IPSJ Transactions on Programming Vol.7 No.2 1–10 (June 2014)

Regular Paper

Design and Implementation of GSN Patterns: A Step toward Assurance Case Language

Yutaka Matsuno[1,a)]

Received: November 11, 2013, Accepted: March 1, 2014

Abstract: Assurance cases are documented body of evidence that provide valid and convincing argument that the system is adequately dependable in a given application and an environment. Assurance cases are widely required as a regulation for safety-critical systems in EU. There have been several graphical notations for assurance cases. GSN (Goal Structuring Notation) and CAE (Claim, Argument, Evidence) are such two notations. However, these notations have not been defined in a formal way. This paper presents a formal definition of GSN and its pattern extensions. We take the framework of functional programming language as the basis of our study. The implementation has been done on an Eclipse based GSN editor. We report case studies on previous works about GSN and show the applicability of the design and implementation. This is a step toward developing an assurance case language.

Keywords: assurance cases, GSN (Goal Structuring Notation), GSN patterns, functional programming language

1. Introduction

System assurance has become important in many industrial areas, and the notion of *assurance cases* [13] has been getting attention.

Safety cases (assurance cases for system safety) are required to submit to certification bodies for developing and operating safety critical systems, e.g., automotive, railway, defense, nuclear plants and sea oils. There are several standards, e.g., EUROCONTROL [8], Rail Yellow Book [28] and MoD Defense Standard 00-56, which mandate the use of safety cases. The current state of safety cases in the UK is summarized in Ref. [34].

There are several definitions for assurance cases. We show one of such definitions as follows.

> A structured argument, supported by a body of evidence that provides a compelling, comprehensible and valid case that a system is safe for a given application in a given environment [5].

There have been several researches on graphical notations for assurance cases to ease the difficulty of writing and validating them. *GSN* (*Goal Structuring Notation*) [17] and *CAE* (*Claim, Argument, Evidence*) [5] are two such notations. Writing assurance cases and reusing them in a cost effective way is a critical issue for organizations. Patterns, modules, and their supporting constructs are proposed in GSN for the reuse of existing assurance cases, which includes parameterized expressions and reference from a node in a module to another node in other modules. Recently the basic syntax and the extensions for patterns and modules have been defined in the GSN community standard [10].

Assurance cases are getting attention as a framework for assuring systems dependability to various stakeholders including government sectors, certification bodies, and end users. The OPENCOSS project is an European large scale integrating FP7 project dedicated to produce the first European-wide open safety certification platform [25]. In the OPENCOSS project, GSN has been studied as a representation of certification documents. The SAFE (Safe Automotive soFtware architEcture) project [31] aims to enhance methods for defining safety goals and define development processes complying with the new ISO26262 [14] standard for functional safety in automotive electrical and electronic systems. As ISO26262 mandates the use of safety cases, meta-model, patterns, and other topics on safety cases have been studied in the project. The DEOS (Dependable Embedded Operating System) project [36] is a Japanese national project for developing dependable systems. The project aims to use assurance cases in both development and operational phases (specially for failure response action) [9], [23] for assuring the dependability of the systems.

However, as assurance cases are a new research field, the syntax and semantics of assurance case languages are in developing stages. For example, the GSN community standard has some informal definitions for GSN and its extensions, as shown in Ref. [23]. Recently several efforts have been done for formalizing GSN and its extensions [6], [32]. However, there still has not been a well accepted formalization for assurance case language. To facilitate the discussions and development, this paper presents a new formalization of GSN and GSN patterns. Our aim is to develop simple and general framework which can be implemented easily. To do so, we exploit the framework of functional programming languages [27], which is the most basic and formal framework in programming languages.

Our contributions are as follows.

- We give a formal definition and semantics for GSN and its pattern extension.

[1] The University of Electro-Communications, Chofu, Tokyo 182–8585, Japan
[a)] matsuno@is.uec.ac.jp

- We implement GSN and its pattern extension on an open source GSN editor [19].
- We test the applicability of the extension with existing GSN examples [3], [11], [12], [15], [38].

The paper is organized as follows. Section 2 presents background of assurance cases and the graphical notations. Section 3 presents our formalization of GSN and its pattern extensions. Section 4 introduces current implementation of the extension. Section 5 discusses related work. Section 6 concludes the paper.

2. Background

2.1 Assurance Cases

An assurance case is called a safety case when arguing the safety of a system. Similarly, it is called a dependability case, security case, reliability case, or availability case when arguing the dependability, security, reliability, or availability of a system, respectively. The basic structure of assurance cases is shown in **Fig. 1** (slightly modified from the original figure in Ref. [4]).

The scientific background of assurance cases includes Toulmin's Argumentation Model [37], which is the basic model in argumentation theory [26]. Toulmin identifies the key components of information in terms of the roles played within the argument. These components are facts, warrants, backing, rebuttal, and qualified claims. Facts are the basis for the argument. A warrant is the part of the argument that relates facts to qualified claims. Backing is some kind of justification for a warrant. Rebuttal captures the circumstances that would be regarded as exceptions for a warrant. A qualified claim is a conclusion that can be drawn if the warrant holds true and the rebuttal does not hold true. In a sense, the facts plus the warrant imply the claim [26]. Comparing this model with Fig. 1, a qualified claim can correspond to the top goal in Fig. 1, facts to evidence, and the warrant to the argument structure.

Assurance cases have been widely recognized in the U.K. after recent serious disasters, including the Piper Alpha North Sea oil disaster in 1988 (167 people dead) and the Clapham Junction rail crash in 1988 (35 people dead). The term safety case seems to have emerged from a report by Lord Cullen on the Piper Alpha disaster [35]. It has been recognized that not only following a prescribed process and filling in some checklists is required, but system developers and operators must argue why their systems are safe during the period of operation based on evidence.

There are some criticisms of assurance cases themselves. In Ref. [18], Leveson wrote, "Most papers about safety cases express personal opinions or deal with how to prepare a safety case, but not whether it is effective."

2.2 GSN (Goal Structuring Notation) and CAE (Claim, Argument, Evidence)

Goal Structuring Notation (*GSN*) is introduced by Tim Kelly and his colleagues [17]. It is a graphical notation for assurance cases. Some safety cases written in GSN are publicly available [7]. We briefly explain constructs and their meanings in GSN. Arguments in GSN are structured as trees with a few kinds of nodes, including: *goal* nodes for claims to be argued for, *strategy* nodes for reasoning steps that decompose a goal into sub goals, and *evidence* nodes for references to direct evidences that respective goals hold. **Figure 2** is a simple example of GSN. The root of the tree must be a goal node, called top goal, which is the claim to be argued (G_1 in Fig. 2). For G_1, a context node C_1 is attached to complement G_1. Context nodes are used to describe the context (environment) of the goal attached to. A goal node is decomposed through a strategy node (S_1) into sub goal nodes (G_2 and G_3). The strategy node contains an explanation, or reason, for why the goal is achieved when the sub goals are achieved. S_1 explains the way of arguing (argue over each possible fault: A and B). When successive decompositions reach a sub goal (G_2) that has a direct evidence of success, an evidence node (E_1) referring to the evidence is added. Here we use a result of fault tree analysis (FTA) as the evidence. For the sub goal (G_3) that is not decomposed nor supported by evidences, a node (a diamond) of type *undeveloped* is attached to highlight the incomplete status of the case. The assurance case in Fig. 2 is written with *D-Case Editor*, an open source, Eclipse based GSN editor [19].

CAE (Claim, Argument, Evidence) [5] is another well known graphical notation. In OMG (Object Management Group), a standardization effort has been done for a common meta-model for graphical notations of assurance cases including GSN and CAE, and the OMG SACM (Structured Assurance Case Metamodel) [24] has been published. GSN and CAE are compatible

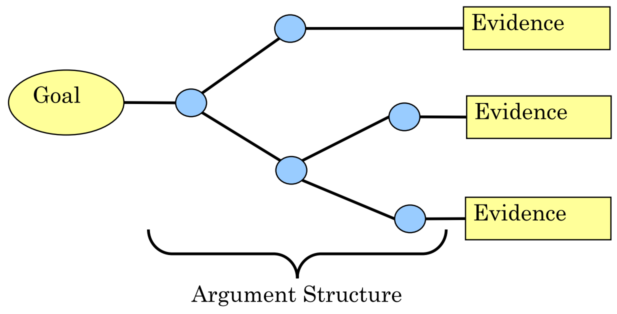

Fig. 1 Argument structure.

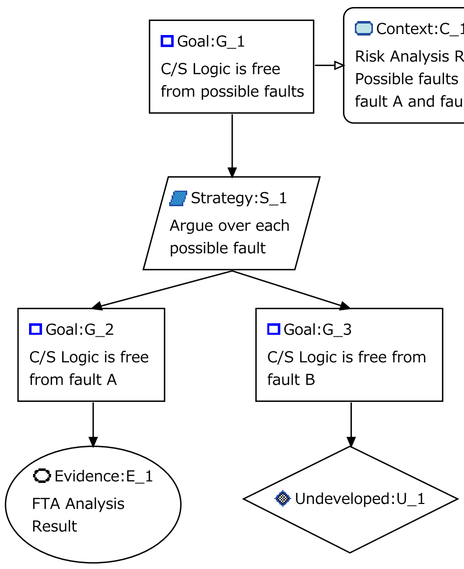
Goal:G_1
C/S Logic is free from possible faults
Context:C_1
Risk Analysis Result Possible faults are fault A and fault B
Strategy:S_1
Argue over each possible fault
Goal:G_2
C/S Logic is free from fault A
Goal:G_3
C/S Logic is free from fault B
Evidence:E_1
FTA Analysis Result
Undeveloped:U_1

Fig. 2 A simple GSN.

to each other via SACM, and our formalism can be adapted to CAE easily.

2.3 GSN Patterns

There have been several publicly available GSN patterns [3], [16], [38]. **Figure 3** is a simple example of GSN patterns in Ref. [3]. The top-level goal of system safety (G1) is re-expressed as a number of goals of functional safety (G2) as part of the strategy identified by S1. In order to support this strategy, it is necessary to have identified all system functions affecting overall safety (C1) e.g., through Functional Hazard Analysis (FHA). In addition, it is also necessary to put forward (and develop) the claim that either all the identified functions are independent, and therefore have no interactions that could give rise to hazards (G4) or that any interactions that have been identified are non-hazardous (G3). Figure 3 includes main GSN extensions for GSN patterns [10]:

- Parameterized expressions. {System X} and {Function Y} are parametrized expressions. We can instantiate X and Y by appropriate (possibly safety critical) system and function, respectively.
- Uninstantiated. Triangles (△) attached to nodes indicate that the nodes contain uninstantiated parametrized expressions. To instantiate the GSN pattern as an assurance case, we need to instantiate the expressions.

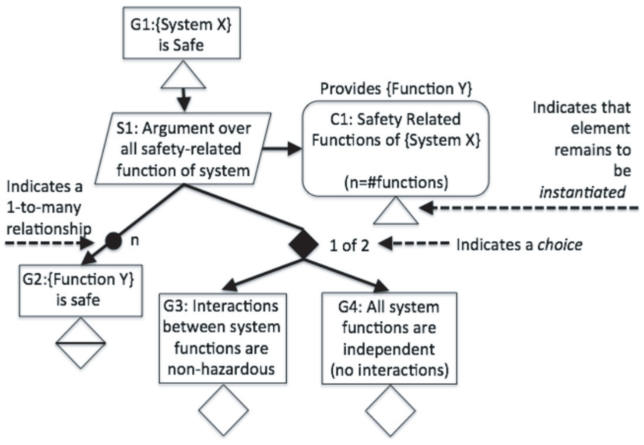
G1:{System X} is Safe
Provides {Function Y}
S1: Argument over all safety-related function of system
C1: Safety Related Functions of {System X} (n=#functions)
Indicates that element remains to be instantiated
Indicates a 1-to-many relationship
n
1 of 2
Indicates a choice
G2:{Function Y} is safe
G3: Interactions between system functions are non-hazardous
G4: All system functions are independent (no interactions)

Fig. 3 An example of GSN patterns [3].

- 1 to many expressions (multiplicity). Number of functions are different by the target system. We can instantiate the number of functions (n) for the target system.
- Choice. The user can select arbitrary sub goals from the set of sub goals depending on the user system.

Loop construct is also introduced. **Figure 4** is a simplified pattern containing loop construct in Ref. [12].

The loop is represented by the back edge from G_2 to S_1. Loop instantiation is done by recursively unfolding the back edge. For example, in Fig. 4, one step unfolding is shown. However, the definition of loop constructs has not been well developed, as the loop counter (n) is given in informal way in Ref. [12].

3. A Formal Definition of GSN and Its Pattern Extensions

3.1 GSN Basic Definition

We first define the GSN term as follows. Let g, e, and st be meta-variables for goals, evidence (solution) and strategies, respectively. For simplicity, we omit other GSN nodes including context, assumption, and justification nodes. Each node contains the description by strings such as "System is dependable." This structured definition of GSN terms first appearing in Ref. [22].

Definition 1 (GSN term T)

$$T \ ::= \Diamond \mid (g, \Diamond) \mid (g, e) \mid (g, st, (T_1, \ldots, T_n))$$

$\Diamond$ implies an empty GSN term. $(g, \Diamond)$ is a GSN term of the top goal g with no supporting argument. (g, e) is a GSN term whose top goal g is supported by a direct evidence e. $(g, st, (T_1, \ldots, T_n))$ is a GSN term with top goal g which is supported by sub trees $T_1, \ldots, T_n$ via strategy st. This definition normalizes current GSN definitions in several way. For example, in the GSN Community Standard [10], strategy nodes can be omitted among goal nodes, and multiple strategies can be connected from the same goal. These node links can be incorporated into our definition by adding a few other nodes (**Fig. 5**). Also, we add sibling order in sub goals of a goal as in Ref. [6].

3.2 GSN Patterns

We formalize the following construct of GSN patterns: parameterized expressions, multiplicity, choice, and loop.

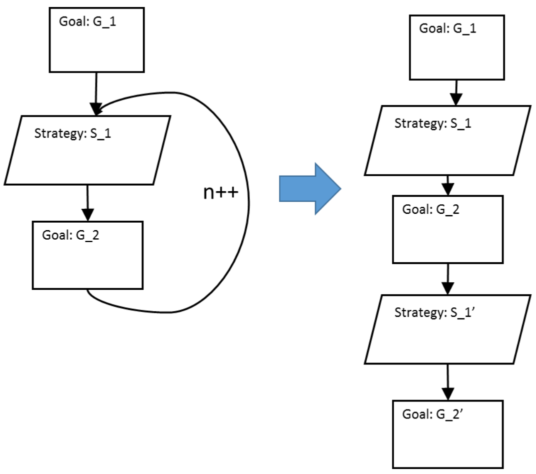

Fig. 4 A GSN pattern with loop.

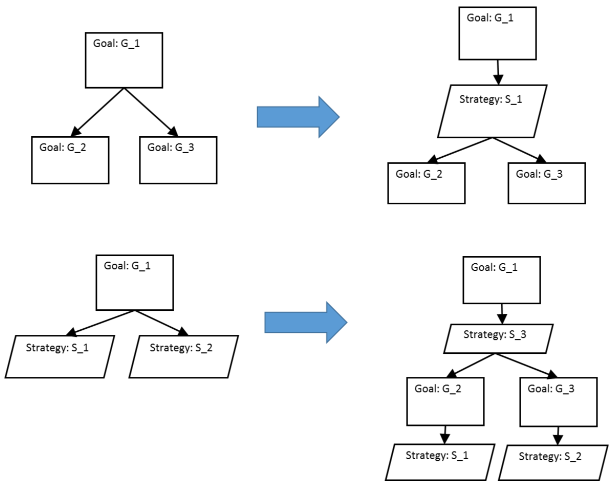

Fig. 5 Normalization of GSN link.

In Ref. [22], Matsuno and Taguchi introduced *types* and define the *scope of variables* appeared in expressions. These two are not new and are fairly basic notions in programming languages. However, the current GSN [3] neither incorporates types nor provides the precise account of the scope of variables. As explained in Section 2.3, the intended meaning of the parameterized expression {System X} in Fig. 3 is to instantiate the variable X by some particular instance which belongs to the *System* class (or type). We believe that introducing types and giving a precise account of the scope of variables will contribute to avoid misuses of parameterized expressions and to detect errors in early stages. For example, we can automatically avoid mis-placement of variables by type checking. In Fig. 3, if a user instantiates X with e.g., "Railway hazards," then the argument does not make sense. It is fairly obvious that type checking prevents such a mis-placement. If the scoping rules are not precisely defined, we cannot figure out where variables in a node are declared in the first place.

We introduce parameter context as a sub-class of context node of the form $[x : \tau = v]$ where x is a parameter of type τ to which a value v is assigned (we use $x, y, z, \ldots$ for parameters and v for values). Parameter context is attached to a goal. In **Fig. 6**, a parameter x is defined in parameter context C_1. x can be used in the goal G_1 and its sub-trees. "x" is defined as a parameter of type string, and assigned a value "car." In current implementation, a parameter can be used in GSN nodes as "[x] is dependable" where parameters are enclosed by "[]." If a parameter is assigned a value, then the occurrence of the parameter in the scope is replaced with the value such as "[car] is dependable." Currently, types τ is defined as follows.

$$\tau ::= int \mid double \mid string \mid enum \mid raw,$$

where raw types mean other than int, double, string, and enum types. Also, the set of values includes $\perp$ for unassigned parameters. In Ref. [10], node "uninstantiated" is attached to a goal node to indicate a parameter is unassigned in the goal node. In our formalization, we use $\perp$ for unassigned parameters. Next, we define choice constructs. Following Ref. [6], we regard the semantics of the choice construct as follows. Given an integer k within the range, a choice construct is instantiated with k sub GSN terms (we use $i, j, k, \ldots$ for integers). For example, if a choice construct has 4 sub GSN terms, and the user chooses 2 for k, then the choice construct is instantiated with the first and second sub GSN terms.

Third, we define multiplicity constructs. Given an integer k within the range, a multiplicity construct is instantiated with k copies of a GSN term.

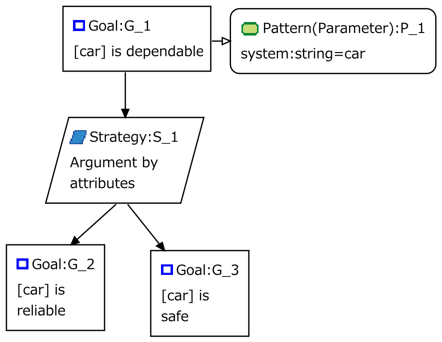

Fig. 6 An example of parameter context.

Definition 2 states the syntax of GSN pattern.

Definition 2 (GSN pattern P)

$$\begin{aligned} d\ &::= \epsilon \mid [x : \tau = v] \\ P\ &::= \alpha \mid \diamond \mid (g, \diamond, d) \\ &\quad \mid (g, e, d) \mid (g, st, (P_1, \ldots, P_n), d) \\ &\quad \mid (g, st, \mathsf{c}[i, j](P_1, \ldots, P_n), d) \\ &\quad \mid (g, st, \mathsf{m}[i, j](P), d) \mid \mu\alpha.P \end{aligned}$$

d is parameter context. Without loss of generality, we assume that at most one parameter can be defined in one parameter context. We omit d if d is ϵ, i.e., no parameter is defined in the goal. α is variable for patterns which is used for loop constructs. $(g, st, \mathsf{c}[i, j](P_1, \ldots, P_n), d)$ is choice construct where $[i, j]$ is the range of pattern instantiation. The user can choose k $(1 \leq i \leq k \leq j \leq n)$ patterns from $P_1, \ldots, P_n$. Multiplicity construct is represented by $(g, st, \mathsf{m}[i, j](P), d)$, where $[i, j]$ is the range of pattern instantiation. The user selects the number of multiplicity k $(1 \leq i \leq k \leq j)$, and the construct is instantiated with k copies of P. $\mu\alpha.P$ represents loop construct. α is a binding variable within the body P. α possibly appears as sub terms of P such as $P_1 = (g_1, st_1, (\alpha, (g_1, e_1)))$. We say P is closed if there is no free occurrence of α within the body of P (note that P_1 is not closed). During instantiation, the user can substitute α with P itself. This represents unfolding of loop construct as shown in Fig. 4.

Pattern Instantiation is defined as a binary relation of the form $P_1 \longrightarrow P_2$ (**Fig. 7**).

The first three are instantiation relations by parameter assignment. For example, in $(g, \diamond, [x : \tau = \perp])$, if the user selects value v for x (v should have the same type of x. Current D-Case Editor only accepts a value of the same type for the parameter), then the pattern is instantiated with $(g[v/x], \diamond, [x : \tau = v])$, with all occurrences of x in g is replaced with v. These first three relations that explicitly determine the scope of the parameter, i.e., the subtree. This is different from the GSN community standard [10] in which there is no formal description of parameter scope and the work by Denney and Pai, which only considers global parameters of a GSN tree [6].

The user-selected value v is indicated as an annotation to the arrow as $\xrightarrow{v}$. This corresponds to non-deterministic evaluation in programming languages. The fourth relation is for choice con-

$$(g, \diamond, [x : \tau = \perp]) \xrightarrow{v} (g[v/x], \diamond, [x : \tau = v])$$
$$(g, e, [x : \tau = \perp]) \xrightarrow{v} (g[v/x], e[v/x], [x : \tau = v])$$
$$(g, st, (P_1, \ldots, P_n), [x : \tau = \perp]) \xrightarrow{v}$$
$$(g[v/x], st[v/x], (P_1[v/x], \ldots, P_n[v/x]), [x : \tau = v])$$
$$(g, st, \mathsf{c}[i, j](P_1, \ldots, P_n), d) \xrightarrow{k} (g, st, (P_1, \ldots, P_k), d)$$
$$(g, st, \mathsf{m}[i, j](P), d) \xrightarrow{k} (g, st, (P, \ldots, P), d)$$
$$(P \text{ repeats } k \text{ times})$$
$$\mu\alpha.P \xrightarrow{\mu} P[\mu\alpha.P/\alpha]$$
$$\mu\alpha.P \xrightarrow{\diamond} \diamond$$

Fig. 7 Pattern instantiation relation $P_1 \rightarrow P_2$.

struct instantiation. The user chooses the number of sub GSN terms k within the range, and the construct is instantiated by k sub GSN terms. Similarly, the fifth relation is for multiplicity construct instantiation. If the user selects the multiplicity number k, the construct is instantiated with k copies of P as sub GSN terms of the goal g. The last two relations are for loop constructs. If the user wants to unfold the loop constructs, the occurrence of α within the body P is replaced with $\mu\alpha.P$ it self. Otherwise (the last rule), the loop construct is replaced with $\Diamond$. Note that a loop construct can be unfolded many times as the user wants.

Next, we generalize the pattern instantiation relation by *environmental context* E. An environmental context is a pattern with possibly multiple *holes* []. For example, if $E = (g_1, st, ([], P_2), d)$, then $E[P_1] = (g_1, st, (P_1, P_2), d)$.

Definition 3 (Environmental Context E)

$$\begin{aligned} E \ ::= & \ [] \mid \alpha \mid \Diamond \mid (g, \Diamond, d) \\ & \mid (g, e, d) \mid (g, st, (E_1, \ldots, E_n), d) \\ & \mid (g, st, \mathsf{c}[i, j](E_1, \ldots, E_n), d) \\ & \mid (g, st, \mathsf{m}[i, j](E), d) \mid \mu\alpha.E \end{aligned}$$

Using E, the pattern instantiation rules also include:

$$\frac{P_1 \longrightarrow P_2}{E[P_1] \longrightarrow E[P_2].}$$

The following definitions state the relation between a pattern and its instances.

Definition 4 ($\mathsf{elim}(P)$) $\mathsf{elim}(P)$ is a function that returns P' in which all parameter contexts are eliminated from P. For example, if $P = (g, e, [x : \tau = v])$, then $\mathsf{elim}(P) = (g, e)$.

Definition 5 (Normal Form) A pattern P is said to be normal form if and only if there does not exit P_1 such that $P \longrightarrow P_1$.

Definition 6 (Instances of a Pattern) Let P be a pattern in which all parameters are unassigned. If

$$P \longrightarrow_* I$$

and I is a normal form, then $\mathsf{elim}(I)$ is an instance of P.

Thanks to the functional programming language formalization, pattern instantiation algorithm can be defined in a straight and recursive way. We denote the algorithm as $\Pi(P)$. In the algorithm, $u = v, k, \mu, \Diamond$ are user input. The algorithm is shown in **Fig. 8**. Note that v is other than $\perp$ in the algorithm.

```
$\Pi(P) =$
case $P$ of
$(g, \Diamond, \epsilon) \Longrightarrow (g, \Diamond, \epsilon)$
$(g, e, \epsilon) \Longrightarrow (g, e, \epsilon)$
$(g, st, (P_1, \ldots, P_n), \epsilon) \Longrightarrow (g, st, (\Pi(P_1), \ldots, \Pi(P_n)), \epsilon)$
$(g, \Diamond, [x : \tau = \perp]) \Longrightarrow (g[v/x], \Diamond, [x : \tau = v])$
$(g, e, [x : \tau = \perp]) \Longrightarrow (g[v/x], e[v/x], [x : \tau = v])$
$(g, st, (P_1, \ldots, P_n), [x : \tau = \perp]) \Longrightarrow$
  $(g[v/x], st[v/x], (\Pi(P_1[v/x]), \ldots, \Pi(P_n[v/x])), [x : \tau = v])$
$(g, st, \mathsf{c}[i, j](P_1, \ldots, P_n), d) \Longrightarrow \Pi((g, st, (P_1, \ldots, P_k), d))$
$(g, st, \mathsf{m}[i, j](P), d) \Longrightarrow \Pi((g, st, (P, \ldots, P), d))$
$\mu\alpha.P \Longrightarrow \Pi(P[\mu\alpha.P/\alpha])$ if $u = \mu$
$\mu\alpha.P \Longrightarrow \Diamond$ if $u = \Diamond$
```

Fig. 8 Pattern instantiation algorithm.

The correctness of the algorithm is stated as follows.

Theorem 1 Let P be a closed GSN pattern. If

$$\Pi(P) = I,$$

then $\mathsf{elim}(I)$ is an instance of P.

4. Implementation

We have implemented the GSN pattern extensions using the open source code of D-Case Editor [19].

Matsuno and Yamamoto [23] reported a preliminary implementation of parameter instantiation function in D-Case Editor. In this paper, we have implemented all pattern constructs. The pattern instantiation takes the following steps.

(1) The user selects an appropriate pattern from the pattern library.
(2) The user recursively chooses values for parameters, number of multiplicity and choice, and whether to unfold loop structure once or not on the GSN tree structure.
(3) The editor automatically places the instantiated pattern in the canvas.

The implementation has been done using Eclipse GMF (Graphical Modeling Framework) [1]. A user can easily install D-Case Editor as a plug-in of Eclipse. The download page is located in Ref. [19]. The source codes for GSN pattern have already been incorporated in the D-Case Editor open source repository [2]. A screenshot of D-Case Editor is shown in **Fig. 9**. Using D-Case Editor, a user can draw a GSN diagram by putting GSN nodes and links in the canvas.

We show some fragments of the source code for pattern instantiation.

Putting a pattern into the canvas

A user can put an instance of a pattern by right clicking the canvas, and select a pattern from "Add Pattern" menu. This function is implemented by method `widgetSelected()` in `AddPatternSelectionAdapter` class. The source code is located in Ref. [20].

The procedure of method `widgetSelected()` is as follows.

- Get the selected pattern by the user.
- Test whether the top goal of the pattern exists or not. If not, report the error.
- Copy the pattern into an object `copyModel`.
- Call method `ModuleUtil.processPatterns`, which takes `copyModel` as an argument, and returns the

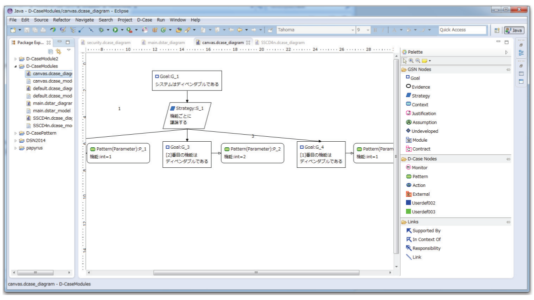

Fig. 9 A screenshot of D-Case Editor.

instantiated pattern. The procedure of method `ModuleUtil.processPatterns` will be explained next.
- Put the instantiated pattern into the canvas.

Processing Loop, Choice, and Multiplicity instantiation

The main part of the instantiation algorithm (Fig. 8) is implemented in method `ModuleUtil.processPatterns` (used in method `widgetSelected()`) from 1067 line of `common/util/ModuleUtil.java`, which can be obtained from [21].

In the implementation, GSN terms are represented by lists of nodes and links:
`copyArgument.getRootBasicNode()` and
`copyArgument.getRootBasicLink()`.

Method `ModuleUtil.processPatterns` traverses the GSN term and instantiates patterns from the top goal according to the tree structure.

A part of the source code is shown below. For readability, some parts of the code is omitted (indicated as `(...SKIP...)`). Among loop, choice, and multiplicity constructs, only the code of choice construct is shown. For detail, please refer to the source code [21].

```
public static boolean processPatterns
(Argument copyArgument) {

// Get the top goal (rootNode).
BasicNode rootNode =
ModuleUtil.getRootNode(copyArgument);

(...SKIP...)

// This while loop
// corresponds to PI(P) in Fig.8.
// (Parameter instantiation
// is in other method.)

while (true) {
// Search pattern nodes
// from the top goal

(...SKIP...)

// Process pattern nodes.
// cNode is the current Pattern node.
// subType is the type of the pattern node,
// either loop, choice, or multiplicity.
// Each pattern construct is
// instantiated according to the subType.

// k is the user input.
// k is the iteration number
// for loop construct.
// For choice and multiplicity,
// k is the number of sub goals
// to be instantiated.

System scNode = (System)cNode;
String subType = scNode.getSubType();
int k = getPatternNumber(scNode);

(...SKIP...)


// Case for Loop pattern construct //

if (PatternUtil.isLoop(subType)) {

// Unfolding the loop pattern
// construct k-1 times.
(...SKIP...)}

// Case for Choice pattern construct //

if (PatternUtil.isChoice(subType)) {

// Get the children nodes
List<BasicNode> childList =
PatternUtil.getChildren(cNode, copyArgument);

// Remove from k+1 th to n th sub goals
// from the pattern,
// and remain the first k sub goals.

for (int i = k; i < childList.size(); i++) {
ArrayList<BasicNode>pnodeList =
new ArrayList<BasicNode>();
ArrayList<BasicLink>plinkList =
new ArrayList<BasicLink>();
HashSet<BasicNode>checkedSet =
new HashSet<BasicNode>();
BasicNode pnode = childList.get(i);
PatternUtil.getSubtree
(pnode, copyArgument, pnodeList
, plinkList, checkedSet);
copyArgument.getRootBasicNode()
.removeAll(pnodeList);
copyArgument.getRootBasicLink()
.removeAll(plinkList);
// Remove links of deleted nodes.
for (BasicNode dnode : pnodeList) {
PatternUtil.removeLinks
(dnode, copyArgument.getRootBasicLink());
}}}

// Case for Multiplicity pattern construct //

if (PatternUtil.isMultiplicity(subType)) {

// Add k-1 times of the pattern construct.
// The implementation is mostly similar to
// that of choice construct
(...SKIP...)}

// remove the current Pattern node.
(...SKIP...)
// Process all Parameter nodes.
// Continue until no Parameter nodes found.
(...SKIP...)}
return true;}
```

4.1 Pattern Instantiation Examples

We show a few pattern examples in our implementation.

A simple example

Let $P_1 = (g_1, st_1, \mathsf{m}[1,3]((g_2, e_2, [FunctionName : String = \bot])))$, where g_1 = "System is dependable," st_1 = "Argument over functions," g_2 = "[⊥] is dependable," e_2 = "Evidence for [⊥]." Note that a parameter is enclosed by [] in the node description. P_1 can be written as in **Fig. 10**. In our implementation, parameter, multiplicity, choice, and loop constructs are generalized as "Pattern" node. **Figure 11** shows an instantion of P_1. The number of multiplicity is set as 2, and parameter "FunctionName" is instantiated with "Function1" and "Function2" in each instantiated

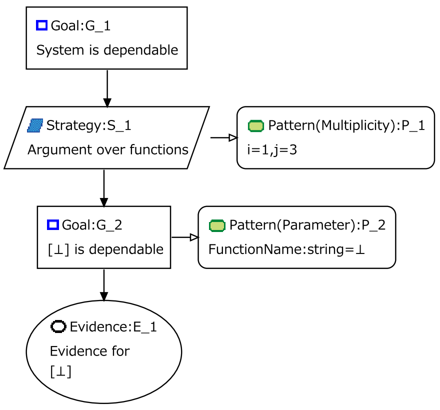

Fig. 10 A simple example of pattern.

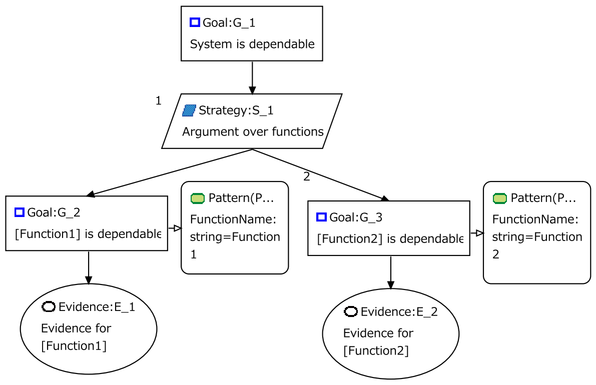

Fig. 11 An instantiation of Fig. 10.

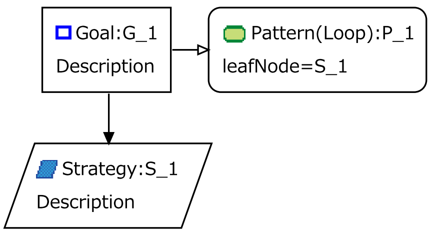

Fig. 12 A simple example of loop pattern.

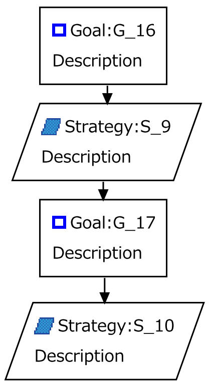

Fig. 13 An instantiation of Fig. 12.

sub tree, respectively.

A simple loop example

Let $P_2 = \mu\alpha.(g_1, st_1, (\alpha))$. P_2 in D-Case Editor is shown in **Fig. 12**. In current implementation, an occurrence of α is defined by choosing the parent node. In Fig. 12, the strategy node S_1 is chosen as the parent node of α. **Figure 13** is an instance of the loop pattern by unfolding of the loop once.

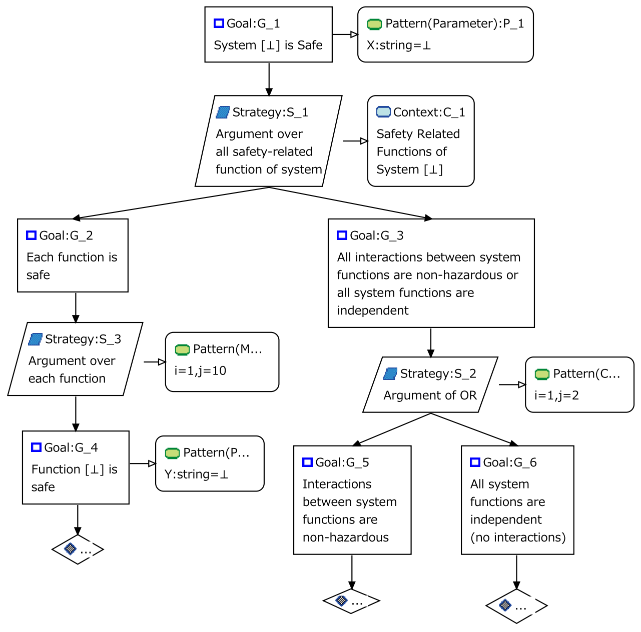
Goal:G_1
System [⊥] is Safe
Pattern(Parameter):P_1
X:string=⊥
Strategy:S_1
Argument over all safety-related function of system
Context:C_1
Safety Related Functions of System [⊥]
Goal:G_2
Each function is safe
Goal:G_3
All interactions between system functions are non-hazardous or all system functions are independent
Strategy:S_3
Argument over each function
Pattern(M...
i=1,j=10
Strategy:S_2
Argument of OR
Pattern(C...
i=1,j=2
Goal:G_4
Function [⊥] is safe
Pattern(P...
Y:string=⊥
Goal:G_5
Interactions between system functions are non-hazardous
Goal:G_6
All system functions are independent (no interactions)
...
...
...

Fig. 14 Pattern example in Fig. 3.

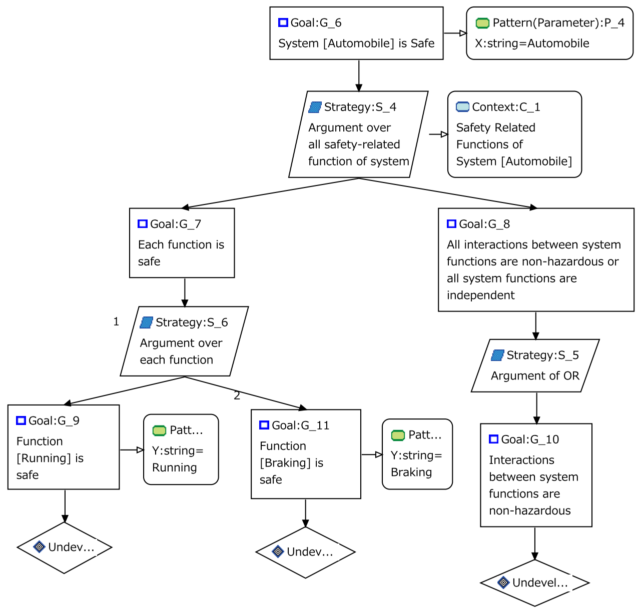
Goal:G_6
System [Automobile] is Safe
Pattern(Parameter):P_4
X:string=Automobile
Strategy:S_4
Argument over all safety-related function of system
Context:C_1
Safety Related Functions of System [Automobile]
Goal:G_7
Each function is safe
Goal:G_8
All interactions between system functions are non-hazardous or all system functions are independent
1
Strategy:S_6
Argument over each function
Strategy:S_5
Argument of OR
2
Goal:G_9
Function [Running] is safe
Patt...
Y:string= Running
Goal:G_11
Function [Braking] is safe
Patt...
Y:string= Braking
Goal:G_10
Interactions between system functions are non-hazardous
Undev...
Undev...
Undevel...

Fig. 15 An instantiation of Fig. 14.

The pattern example in Fig. 3

Figure 14 represents the pattern example in Fig. 3. We normalized the example in Fig. 3 to separate multiplicity and choice constructs. Note that the choice semantics used in the example is different from our semantics. The former is to choose one of two sub trees whereas the latter is to choose one sub tree or two sub trees. It is easy to define the semantics of choice constructs according to that of Fig. 3.

An instantiated pattern of Fig. 14 is shown in **Fig. 15**. For the parameter X, we select "Automobile." We assume that "Automobile" has two functions: "Running" and "Braking," which are instances of parameter Y. For the choice construct, we choose the sub goal "Interactions between system functions are non-hazardous."

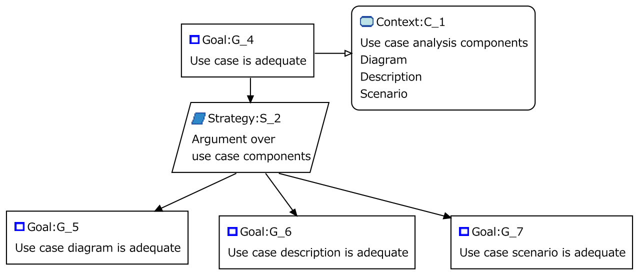

Fig. 16 A GSN fragment for use case analysis for ACC.

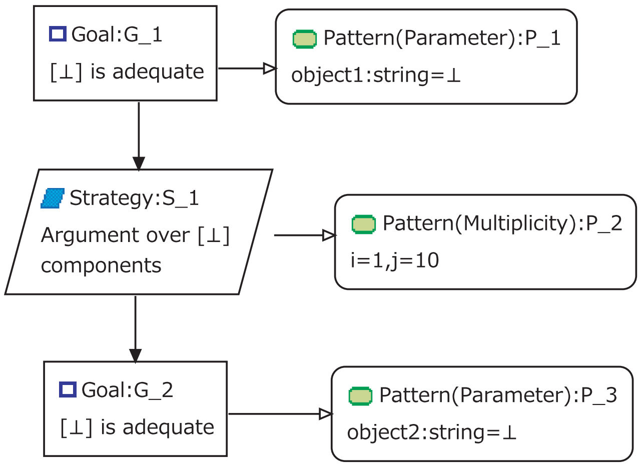

Fig. 17 A GSN pattern for decomposition.

4.2 Preliminary Evaluation

As a preliminary evaluation, we show comparison between writing GSN with and without GSN pattern function by counting user steps. We take an example in development of ACC (Adaptive Cruise Control) system. ACC is a sub system of an automobile for automatically adjusting distance between the user automobile and an automobile ahead of it. **Figure 16** is a GSN top level fragment for UML use case of a ACC system, written by an engineer who is an expert in UML modeling. UML use case consists of three components: use case diagram, use case description, and use case scenario list. This fragment is for assuring that use case analysis for ACC system is adequate by decomposing the goal into sub goals for its components. This decomposition by a construct is very typical in conventional GSN diagrams. For such decomposition, the pattern in **Fig. 17** is useful. This pattern is essentially the same as the simple pattern of Fig. 10. An instance of the pattern is shown in **Fig. 18**. By adding the context C_1 in Fig. 16, this essentially becomes the same fragment.

We compare the user steps for writing the GSN fragment in Fig. 16 and the instance in Fig. 18.

- Input words counting. The fragment in Fig. 16 requires 24 input words, whereas the instance in Fig. 18 requires 10 words (we omit the word counting for the context C_1).
- Processing steps counting. The fragment in Fig. 16 requires 6 and 5 selections for nodes and links, respectively. The instance in Fig. 18 requires 1 pattern selection and 1 instantiation step (for multiplicity).

Therefore, user steps for the fragment in Fig. 16 is $24+6+5 = 35$ and the instance in Fig. 18 is $10 + 1 + 1 = 12$. This result indicates GSN patterns reduce user steps significantly. The difference

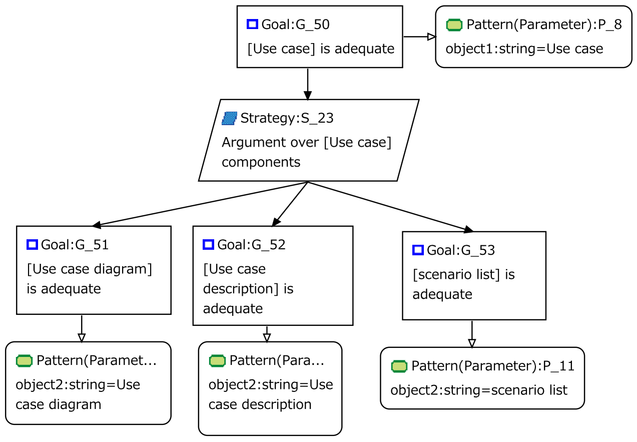

Fig. 18 An instance of the pattern in Fig. 17.

could increase as the size of GSN diagrams become bigger.

4.3 Discussions

Currently, we are representing existing GSN examples in the literatures [3], [11], [12], [15], [38]. We found that most of existing examples can be represented in our implementation with a few minor modifications.

There are several issues to be considered. An issue is to combine instantiation of parameters and other pattern constructs. For example, in loop instantiation, sometimes a parameter should be instantiated automatically according to the loop counter. In Ref. [11], a loop pattern is introduced for each software "tier." In our current implementation, the number of tier need to be instantiated manually for each loop unfolding. Other issues include addition of "list pattern." For example, assume that there is a hazard list. In many existing GSN examples, a goal is divided into sub goals for each item of such lists (the GSN fragment of ACC system in the previous sub section is such one). It seems worth defining list pattern for automatically generating sub goals according to a given list in the context node in the goal.

For simplicity, our implementation of the choice construct is restricted to be instantiated to the first k sub GSN terms. It is easy to generalize the instantiation to arbitrary k sub GSN terms (for example, $(g, st, \mathsf{c}[1, 3](P_1, P_2, P_3))$ can be instantiated to $(g, st, (P_1, P_3)))$. The next version of D-Case Editor will incorporate the generalized choice construct.

We list other interesting issues.

- More types. Current implementation only has basic types for parameters: *int*, *double*, *string*, *enum*, and *raw* types. It is worth adding more types relevant to system assurance such as *system*, *function*, *fault*, *failure*, *SIL*, and so on. Also, it would be beneficial to introduce mechanism for use-defined types.
- Reverse instantiation. In Fig. 7, parameter instantiations leaves the parameter context after the instantiation such as $(g, \Diamond, [x : \tau = \bot]) \xrightarrow{v} (g[v/x], \Diamond, [x : \tau = v])$. As x does not appear in the instantiated pattern, the parameter context is semantically not necessary, so the relation can be written as $(g, \Diamond, [x : \tau = \bot]) \xrightarrow{v} (g[v/x], \Diamond)$. The reason for leaving the parameter context is that it is often the case that a user wants to replace the value of a parameter. For example, assume

that g = "the system satisfies SIL [x]." SIL (Safety integrity level) consists of 1 through 4 levels. A user might select 1 for x for the first time, but the user would want to change the value to 2 later. In such a case, the parameter context should remain. As in this case, reverse instantiation would be needed for practical use of a GSN editor. In current implementation, only reverse instantiation of parameters is allowed. Mechanism for reverse instantiation of other pattern constructs is left as future work.

5. Related Work

The most closely related work is Denney and Pai's work [6]. We follow some part of their paper such as the semantics of choice and multiplicity. Their paper defines GSN as a control flow graph, and introduces pattern constructs on the graph. However, a few subtle issues arise due to the un-structuredness of the control flow graph. For example, patterns are required to satisfy a condition on the back-edges. Also, their instantiation algorithm is sophisticated but contain a few ad-hoc parts, and the notion of scope of parameters has not been considered. In general, in the programming languages field, structured representation of a program is preferred over un-structured representation due to the difficulties in the treatment of un-structured objects. Our formalization is fairly structured: GSN is represented by a simple tree structure, and only structured simple loops are allowed in GSN pattern. Our formalism made several simplifications. We believe that our limitation deserves the benefit of structured-ness.

Takeyama has implemented D-Case/Agda [32], which is an interactive GSN editing and verifying tool. Agda [33] is a dependently typed functional programming language and also a proof assistant. D-Case/Agda lets users write GSN in Agda. This enables more formal and consistent GSN to be written and verified. The definition of GSN in Ref. [32] is structured and we referred to the definition. However, because D-Case/Agda directly uses Agda, it is difficult for ordinary users to write GSN in D-Case/Agda. Also, D-Case/Agda does not explicitly comply with the GSN community standard. If a user wants to write very rigorous GSN, then D-Case/Agda will be a good alternative. How much formalism should assurance case language has is an often discussed question in the community.

Our work is based on Matsuno and Taguchi's work [22] which introduced parameters for pattern. However, the implementation is limited because parameters can only be defined as global parameters of a GSN term. This paper further introduced choice, multiplicity, and loop pattern contracts, and defined pattern instantiation as a binary relation on GSN patterns. In current implementation, all pattern constructs can be defined both globally and locally in a GSN term.

There have been works for verification of assurance cases such as Ref. [29]. However, such works use their own definition of assurance cases representation. Defining an assurance case language will be a base for such verification work, as various type systems have been developed on functional programming languages such as Standard ML and Haskell based on λ calculus, the basic model of functional programming language. It is worth studying how to apply other functional programming concepts such as polymorphism for document generation for dependability assurance.

6. Concluding Remarks

In this paper we have reported on our formalization and implementation of GSN and its extensions, using a functional programming framework. Currently we are developing a module system. Together with the current pattern instantiation function, we plan to release the next version of D-Case Editor soon. This implementation could be a base for developing an assurance case language.

Assurance cases are becoming important as a framework for dependability assurance. Therefore, an assurance case language should be defined and implemented in a formal way. This will also help automatic verification of assurance case documents, as noted in Ref. [30].

A future step is to show that the assurance case language can be used for generating documents for dependability assurance of a real system. By feedbacks from the use in real systems, the language could be refined and made more practical. We would like to report the results in a near future.

Acknowledgments This work has been supported by JST CREST DEOS project funded by Japan Science and Technology Agency (JST). We would like to thank the reviewer for constructive suggestions.

References

[1] Edipse GMF, available from ⟨http://www.eclipse.org/modeling/gmp/⟩.
[2] D-case Editor GitHub Repository, available from ⟨https://github.com/d-case/d-case_editor⟩.
[3] Alexander, R., Kelly, T., Kurd, Z. and McDermid, J.: Safety cases for advanced control software: Safety case patterns, Technical report, Department of Computer Science, University of York (2007).
[4] Bishop, P. and Bloomfield, R.: A methodology for safety case development, *Safety-critical Systems Symposium* (*SSS 98*) (1998).
[5] Bloomfield, R. and Bishop, P.: Safety and assurance cases: Past, present and possible future - an adelard perspective, *Proc. 18th Safety-Critical Systems Symposium*, Bristol, UK (2010).
[6] Denney, E. and Pai, G.: A formal basis for safety case patterns, *SAFECOMP*, pp.21–32 (2013).
[7] Dependability Research Group, available from ⟨http://dependability.cs.virginia.edu⟩.
[8] European Organisation for the Safety of Air Navigation: Safety case development manual, 2006, *European Air Traffic Management* (2006).
[9] Fujita, H., Matsuno, Y., Hanawa, T., Sato, M., Kato, S. and Ishikawa, Y.: DS-Bench toolset: Tools for dependability benchmarking with simulation and assurance, *Proc. IEEE DSN 2012* (2012).
[10] GSN contributors, GSN community standard version 1.0 (2011).
[11] Hawkins, R. and Kelly, T.: A software safety argument pattern catalogue, Technical report, University of York (2013), available from ⟨http://www-users.cs.york.ac.uk/ rhawkins/pubs.html⟩.
[12] Hawkins, R. and Kelly, T.: A systematic approach for developing software safety arguments, *Proc. 27th International System Safety Conference*, Huntsville, AL (2009).
[13] Howell, C.C., Guerra, S., Pfleeger, S.L. and Stavridou-Coleman, V. (Eds.): Workshop on Assurance Cases: Best Practices, Possible Obstacles, and Future Opportunities, *DSN 2004* (2004).
[14] ISO: ISO 26262 road vehicle – functional safety –, part 1 to part 10 (2011).
[15] Kelly, T. and McDermid, J.: Safety case construction and reuse using patterns, *Proc. 16th International Conference on Computer Safety, Reliability and Security* (*SAFECOMP'97*) (1997).
[16] Kelly, T. and McDermid, J.: Safety case patterns – reusing successful arguments, *IEE Colloquium on Understanding Patterns and Their Application to System Engineering* (1998).
[17] Kelly, T. and Weaver, R.: The goal structuring notation – a safety argument notation, *Proc. Dependable Systems and Networks 2004, Workshop on Assurance Cases* (2004).

[18] Leveson, N.: The use of safety cases in certification and regulation, *ESD Working Paper Series*, Boston: MIT (2011).
[19] Matsuno, Y.: D-case editor homepage, available from ⟨http://www.dependable-os.net/tech/D-CaseEditor/⟩.
[20] Matsuno, Y.: D-case editor source code, available from ⟨https://github.com/d-case/d-case_editor/blob/master/net.dependableos.dcase.diagram.editor/src/net/dependableos/dcase/diagram/editor/command/AddPatternContributionItem.java⟩.
[21] Matsuno, Y.: D-case editor source code, available from ⟨https://github.com/d-case/d-case_editor/blob/master/net.dependableos.dcase.diagram.editor/src/net/dependableos/dcase/diagram/editor/common/util/ModuleUtil.java⟩.
[22] Matsuno, Y. and Taguchi, K.: Parameterised argument structure for GSN patterns, *Proc. IEEE 11th International Conference on Quality Software* (*QSIC 2011*), pp.96–101 (2011).
[23] Matsuno, Y. and Yamamoto, S.: An implementation of gsn community standard, *Proc. 1st International Workshop on Assurance Cases for Software-intensive Systems* (*ASSURE 2013*) (2013).
[24] OMG System Assurance Task Force, 2012, available from ⟨http://sysa.omg.org⟩.
[25] OPENCOSS Project: Opencoss project web page, available from ⟨http://www.opencoss-project.eu⟩ (accessed 2013-12).
[26] Esnard, P. and Hunter, A.: *Elements of Argumentation*, The MIT Press (2008).
[27] Pierce, B.C.: *Types and Programming Languages*, The MIT Press (2002).
[28] Railtrack: Yellow book 3, 2000, Engineering Safety Management Issue3, Vol.1 and Vol.3 (2000).
[29] Rushby, J.: Formalism in safety cases, *Proc. 18th Safety-Critial Systems Symposium*, Bristol, UK, pp.3–17 (2010).
[30] Rushby, J.: Logic and epistemology in safety cases, *SAFECOMP*, pp.1–7 (2013).
[31] SAFE Project: Safe project web page, available from ⟨http://www.safe-project.eu⟩ (accessed 2013-12).
[32] Takeyama, M.: A brief introduction to D-Case/Agda *(draft ver. 0.2), available from ⟨http://wiki.portal.chalmers.se/agda/uploads/D-Case-Agda.D-Case-Agda/D-Case-Agda-HomePage-Data.html⟩.
[33] Agda team: Agda wiki page, available from ⟨http://wiki.portal.chalmers.se/agda/pmwiki.php⟩.
[34] The Health Foundation: Evidence: Using safety cases in industry and healthcare, Technical report (2012). available from ⟨http://www.health.org.uk/publications/using-safety-cases-in-industry-and-healthcare/⟩.
[35] The Hon. Lord Cullen: The public inquiry into the piper alpha disaster, vols. 1 and 2 (report to parliament by the secretary of state for energy by command of her majesty) (1990).
[36] Tokoro, M. (Ed.): *Open Systems Dependability: Dependability Engineering for Ever-Changing Systems*, CRC Press (2012).
[37] Toulmin, S.: *The Use of Argument*, Cambridge University Press (1958).
[38] Weaver, R.A.: *The Safety of Software – Constructing and Assuring Arguments*, PhD thesis, Department of Computer Science, University of York (2003).

Yutaka Matsuno was born in 1977. He received his B.S, M.S., and Ph.D. from the University of Tokyo in 2001, 2003, and 2006, respectively. Currently he is an Assistant Professor at the University of Electro-Communications, Japan. His research interests include programming languages, dependability, system assurance, and assurance cases. He is a member of IPSJ, IEIEC, JSSST, JSQC, and REAJ.