Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI Program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA’s STI. The NASA STI Program provides access to the NTRS Registered and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- **TECHNICAL PUBLICATION.** Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- **TECHNICAL MEMORANDUM.** Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- **CONTRACTOR REPORT.** Scientific and technical findings by NASA-sponsored contractors and grantees.

- **CONFERENCE PUBLICATION.** Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- **SPECIAL PUBLICATION.** Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- **TECHNICAL TRANSLATION.** English-language translations of foreign scientific and technical material pertinent to NASA’s mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI Program, see the following:

- Access the NASA STI program home page at [http://www.sti.nasa.gov](http://www.sti.nasa.gov)
- E-mail your question to help@sti.nasa.gov
- Phone the NASA STI Information Desk at 757-864-9658
- Write to: NASA STI Information Desk Mail Stop 148 NASA Langley Research Center Hampton, VA 23681-2199
The Simple Assurance Argument Interchange Format (SAAIF) Manual

Patrick J. Graydon
Langley Research Center, Hampton, VA
Acknowledgments

We thank the branch and directorate reviewers for their feedback on this work.

The use of trademarks or names of manufacturers in this report is for accurate reporting and does not constitute an official endorsement, either expressed or implied, of such products or manufacturers by the National Aeronautics and Space Administration.

Available from:

NASA STI Program / Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199
Fax: 757-864-6500
Abstract

This document describes the Simple Assurance Argument Interchange Format, a proposed meta-model for describing structured assurance arguments. We describe the syntax and semantics of the model elements, compare the meta-model to existing argument formats, and give an example to illustrate its use.

1 Introduction

Assurance cases are variously defined. One prototypical definition of a safety case—an assurance case specifically focused on safety—states that a "Safety Case consists of a structured argument, supported by a body of evidence that provides a compelling, comprehensible and valid\(^1\) case that a system is safe for a given application in a given environment" [1]. Writers have recorded assurance arguments in a number of different formats:

- Structured and freeform prose text [2, 3]
- Tables [3]
- Graphical argument formats such as Goal Structuring Notation (GSN) and Claims-Argument-Evidence models (CAE) [4, 5]
- Combinations of symbolic and natural-language logic [6–8]

The related philosophy literature includes yet more formats, including Toulmin structures and Wigmore diagrams among many others [9–11].

In 2013, the Object Management Group (OMG) published the first version of the Structured Assurance Case Metamodel (SACM) [12]. Some observers cited the metamodel’s complexity as a hindrance to widespread adoption. Work then began on a replacement version that was meant to be simpler. A beta of SACM version 2.0 was released in 2016 and later refined [13]. The final version of SACM version 2.0 was released in March 2018 [14].

Taking inspiration from automobile designer Colin Chapman’s exhortation to “simplify, then add lightness,” we wondered whether a metamodel could be simpler than the SACM yet still serve the purposes of assurance argumentation. This document presents a candidate answer inspired by the SACM and, to a lesser degree, by GSN and Toulmin’s model [5, 9, 14]. We call our metamodel the Simple Assurance Argument Interchange Format (SAAIF).

1.1 Design goals

Our design goals for the SAAIF are that it should:

1. Serve the purposes to which assurance arguments are typically put
2. Be simple enough that a class diagram fits legibly on one sheet of paper
3. Have semantics described well enough to permit argument analysis

\(^1\) This restrictive clause yields unexpected effects. For example, C. M. Holloway observes that if an assurance case is reviewed and found not compelling, it ceases to qualify as a case.
It is worth noting with respect to the first of these that there appears to be several different assurance argument schools of thought [15]. We are concerned here primarily with purposes that involve communication from the writer to many safety stakeholders, such as telling the story of how a system or service achieves the properties to be assured. As one regulator put it,

A safety case is a logical and hierarchical set of documents that describes risk in terms of the hazards presented by the facility, site and the modes of operation, including potential faults and accidents, and those reasonably practicable measures that need to be implemented to prevent or minimise harm. It takes account of experience from the past, is written in the present, and sets expectations and guidance for the processes that should operate in the future if the hazards are to be controlled successfully. The safety case clearly sets out the trail from safety claims through arguments to evidence [16].

Achieving this purpose requires clear and preferably precise communication to human readers, who might have different backgrounds and different technical capabilities. Again, quoting the same regulator:

The primary purpose of a safety case is to provide the licensee with the information required to enable safe management of the facility or activity in question. Therefore it should be understandable to and usable by those with direct responsibility for safety [17].

Accordingly, in reducing the SACM to create the SAAIF we have favored clear, human-readable expression over other considerations that other researchers have addressed, such as deductive validity or computable confidence [8,18].

Whether the SAAIF meets the goals identified above or not is not a matter we have addressed in depth. We present no concrete efficacy hypotheses here, much less valid empirical evidence of efficacy. The purpose of this document is merely to present the SAAIF for further discussion and study, not to present an assessment of it.

1.2 Document conventions

Sections 2 and 3 present a definition of the SAAIF. All subsections are normative unless otherwise marked. Appendices A–C present mappings between the SAAIF and GSN, Toulmin’s model, and confidence maps. Appendix D presents an example argument taken from NASA’s Explicate ’78 work in order to illustrate how the SAAIF might be used.

Where terms from the model are used in the text, they are typeset \textit{like this} and hyperlinked to their definitions where possible.
2 Model

Figure 1 depicts a UML model of the Simple Assurance Argument Interchange Format. Briefly:

- Each instance of the AssuranceCase class (Section 2.16) represents an assurance case. Each assurance case comprises an arbitrary number of Glossary, Argument, and Inventory objects.
- Each instance of the Glossary class (Section 2.7) represents a set of term definitions to be used within the assurance case. Each definition is represented by an instance of the Definition class (Section 2.6). These definitions can be cited in a machine-readable manner from within the EscapedString fields used to specify the text of the argument.
- Each instance of the Argument class (Section 2.15) represents a self-contained fragment of argument. Arguments comprise an arbitrary number and mix of Claim, EvidenceReference, and Inference objects.
  - Each Claim object (Section 2.12) represents a single claim.
  - Each EvidenceReference object (Section 2.13) represents a citation of evidence.
  - Each Inference object (Section 2.14) describes how a set of claims or evidence references serves as premises to show the truth of one or more claims or inferences.
- The abstract ArgumentElement class (Section 2.11) provides a common base class for claims, evidence references, and inferences.
- Each instance of the Inventory class (Section 2.10) represents a collection of named entities that can be discussed within the argument (e.g., cited as evidence). The inventory comprises an arbitrary number of Artifact objects (Section 2.8). Relationships between artifacts—e.g., cases where one artifact is derived, at least in part, from another, or when one artifact comprises others—may be documented using instances of the ArtifactRelationship class (Section 2.9).
- The ModelElement class (Section 2.5) provides a common abstract base class for most SAAIF classes.
- The EscapedString class (Section 2.2) encodes text that might reference other model elements in a machine-readable manner.
- Instances of the Identifier class (Section 2.1) serve to identify model elements.
- Instances of the KeyValuePair class (Section 2.4) are used to annotate model elements. Each pair’s value comprises a set of objects derived from the abstract Value class (Section 2.3). The SAAIF metamodel defines the EscapedStringValue (Section 2.3.1), (plain) StringValue (Section 2.3.2), and BooleanValue (Section 2.3.3) types. Users may define other value types as needed.

The following sections explain each of the model’s classes in turn.
Figure 1: UML model of the Simple Assurance Argument Interchange Format.
2.1 The Identifier class

An Identifier is a human-readable identifier for AssuranceCase extensions and objects of the Definition, KeyValuePair, ModelElement, and derived classes.

Derivation. The Identifier class is not derived from any other classes. No classes derive from Identifier.

Fields and relationships. This class’s fields and relationships are:

- domain : Base::String — An optional qualifier for identifiers that permits disambiguating between two identifiers with equal name fields. The domain string comprises a string of arbitrary length. The domain ‘SAAIF’ is reserved.
- name : Base::String — A human-readable name for an identified object comprising a string of arbitrary length.

Semantics. An Identifier and any ModelElement or KeyValuePair it identifies is said to be anonymous if the name field of the Identifier is empty. An Identifier is said to be unqualified if its domain is empty but its name is not. A reference Identifier $i_R$ matches an identified object’s Identifier $i_T$ iff both their name and domain fields are equal.

Invariants. All instances must satisfy the following invariants:

1. The domain and name strings must comprise only characters from the set \{‘a’,…,’z’,’A’,…,’Z’,’0’,…,’9’,’_’\}.

(See Section 3.1 for invariants related to the Strict extension.)

Remarks (non-normative). For compactness, some user agents might choose to display name fields but not domain fields, especially for objects with a name field is unique in the model.

2.2 The EscapedString class

An EscapedString is a string in which special character sequences represent machine-readable references to instances in a model of classes derived from ModelElement (e.g., Definition).

Derivation. The EscapedString class is a special form of a plain Unicode string (represented as Base::String in this document). No classes derive from EscapedString.

Fields and relationships. None.
Semantics. String values should be interpreted literally with the exceptions given in Table 1. Machine-readable references to objects of types derived from ModelElement are given by specifying their identifiers enclosed within dollar signs (’$’). User agents might render these as hyperlinks to the referenced ModelElement. If the reference includes specific display text, user agents should render a reference as that display text. Otherwise, user agents should render references as though they had been replaced by the name of the matching ModelElement.

Table 1: Escape sequences for EscapedString objects.

<table>
<thead>
<tr>
<th>String</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘\’</td>
<td>‘\’</td>
</tr>
<tr>
<td>‘$’</td>
<td>‘$’</td>
</tr>
<tr>
<td>‘$\langle \text{domain}\rangle.\langle \text{name}\rangle$’</td>
<td>The name of the ModelElement with the matching id. User agents might hyperlink this text.</td>
</tr>
<tr>
<td>‘$\langle \text{domain}\rangle.\langle \text{name}\rangle:\langle \text{display}\rangle$’</td>
<td>‘\langle \text{display}\rangle’. Useful where it is necessary to change the referenced text’s number, tense, etc. User agents might hyperlink this text.</td>
</tr>
</tbody>
</table>

Invariants. None. (But see Section 3.1 for invariants related to the Strict extension.)

Remarks (non-normative). None.

2.3 The Value class and its subclasses

An object of a type derived from the abstract Value class represents a specified value—e.g., the value of an annotation to a model element represented by a KeyValuePair.

Derivation. The Value type is not derived from any other classes. The SAAIF metamodel defines three sub-classes of Value: StringValue, EscapedStringValue, and BooleanValue. Users may derive other classes as needed.

Fields and relationships. The Value class’s fields are:

- value — All subclasses of Value have a value field of the appropriate type to specify their value.

Semantics. Represents the given value.

Invariants. None.

Remarks (non-normative). None.
2.3.1 The EscapedStringValue class

The EscapedStringValue class embodies a Value in the specific case where its value field is a EscapedString.

2.3.2 The StringValue class

The StringValue class embodies a Value in the specific case where its value field is a simple Unicode string.

2.3.3 The BooleanValue class

The BooleanValue class embodies a Value in the specific case where its value field is a Boolean.

2.4 The KeyValuePair class

Instances of the KeyValuePair class represent named sets of objects derived from Value. KeyValuePair objects are used to annotate objects of types derived from ModelElement.

Derivation. The KeyValuePair type is not derived from any other classes. No other classes are derived from KeyValuePair.

Fields and relationships. This class’s fields and relationships are:

- key : Identifier – Identifies the KeyValuePair.
- value : Value — Gives the KeyValuePair’s values (if any).

Semantics. The presence of a KeyValuePair a in the annotation set of a ModelElement e indicates that e has a property with a name given by a’s key and a value given by a’s value. Some extensions may define special meanings for given key identifiers (see Section 3). Table 2 defines the pre-defined key values for KeyValuePair objects and the corresponding meanings.

Invariants. All instances must satisfy the following invariants:

1. The key’s name must not be empty.

(See Section 3.1 for invariants related to the Strict extension.)

Remarks (non-normative). The IntantiationNote key might seem duplicative of the Note key since both permit decorating ModelElements with arbitrary human-readable text notes. The purpose of distinguishing the former from the latter is to allow a form of remove-before-operation checking for pattern instantiation. See the description of the strict extension in Section 3.1.
Table 2: Pre-defined kinds of KeyValuePair instances.

<table>
<thead>
<tr>
<th>key (domain.name)</th>
<th>value type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAAIF.InstantiationNote</td>
<td>EscapedStringValue [0..*]</td>
<td>Arbitrary human-readable text comments about how to instantiate an argument pattern.</td>
</tr>
<tr>
<td>SAAIF.IsPattern</td>
<td>BooleanValue [1]</td>
<td>If true, indicates that the decorated Argument or Glossary is or is associated with a pattern.</td>
</tr>
<tr>
<td>SAAIF.Note</td>
<td>EscapedStringValue [0..*]</td>
<td>Arbitrary human-readable text comments about the ModelElement.</td>
</tr>
</tbody>
</table>

2.5 The ModelElement class

The abstract ModelElement class is a base class providing common features for most of the classes used to model assurance arguments and assurance cases.

Derivation. The ModelElement class is not derived from any other classes. Several classes derive from ModelElement: Argument, ArgumentElement (and, indirectly, its derived classes Claim, EvidenceReference, and Inference), Artifact, ArtifactRelationship, AssuranceCase, Definition, Glossary, and Inventory.

Fields and relationships. This class’s fields and relationships are:

- annotation : KeyValuePair [0..*] — A set of a KeyValuePair objects that provide more information about the ModelElement. For example, writers might attach freeform comments to a ModelElement using a KeyValuePair with key=SAAIF.Note (see Section 2.4).
- description : EscapedString — A human-readable description of the ModelElement. Given as a EscapedString so as to encode machine-readable references to objects of the Definition class or a class derived from ModelElement. May be empty unless prohibited by the subclass.
- id : Identifier — A machine-readable Identifier for the ModelElement. May be anonymous (see Section 2.1) unless prohibited by the subclass.
- name : Base::String — A human-readable identifier for the ModelElement. May be blank unless prohibited by the subclass.

Semantics. None.

Invariants. All instances must satisfy the following invariants:

1. The name and description fields may not contain whitespace other than space characters and may not begin or end with whitespace.
2. The name field may not contain two or more adjacent space characters.

(See Section 3.1 for invariants related to the Strict extension.)

Remarks (non-normative). A ModelElement’s name is typically short: a few words at most. It’s description may be longer: a phrase or sentence. Narrative text explanations are typically relegated to notes attached as an annotation (see Section 2.4).

2.6 The Definition class

A Definition defines a term of art for an assurance case.

Derivation. The Definition class is derived from the ModelElement class. No classes are derived from the Definition class.

Fields and relationships. This class’s fields and relationships are:

- annotation : KeyValuePair [0..*] (inherited from ModelElement) — See Section 2.5.
- description: EscapedString (inherited from ModelElement) — The definition of the term.
- id : Identifier (inherited from ModelElement) — An identifier for the Definition. This can be used to refer unambiguously to the Definition from within an EscapedString.
- name : Base::String (inherited from ModelElement) — The term to be defined.

Semantics. Definitions have scope throughout the AssuranceCase they are defined in. Wherever it is used, the defined term (and derivatives such as plural forms) should be read as though it had the meaning specified by in the description (mutatis mutandis). While definitions may be explicitly referenced in any EscapedString (including the description of a term), the definitions in an AssuranceCase apply to all uses of defined terms whether or not they are referenced explicitly.

Invariants. None. (But see Section 3.1 for invariants related to the Strict extension.)

Remarks (non-normative). None.

2.7 The Glossary class

A Glossary defines a set of terms of art for an assurance case.

Derivation. The Glossary class is derived from the ModelElement class. No classes are derived from the Glossary class.
Fields and relationships. This class’s fields and relationships are:

- annotation : KeyValuePair [0..*] *(inherited from ModelElement)* — See Section 2.5.
- description : EscapedString *(inherited from ModelElement)* — See Section 2.5.
- entry : Definition [0..*] — The set of entries within the Glossary.
- i : Identifier *(inherited from ModelElement)* — See Section 2.5.
- name : Base::String *(inherited from ModelElement)* — See Section 2.5.

Semantics. Defines a set of terms.

Invariants. None. (But see Section 3.1 for invariants related to the Strict extension.)

Remarks (non-normative). None.

2.8 The Artifact class

An Artifact object identifies an artifact of significance to the assurance argument.

Derivation. The Artifact class is derived from the ModelElement class. No classes are derived from the Artifact class.

Fields and relationships. This class’s fields and relationships are:

- annotation : KeyValuePair [0..*] *(inherited from ModelElement)* — See Section 2.5.
- description : EscapedString *(inherited from ModelElement)* — Describes the artifact, answering the question, What kind of thing is it?
- id : Identifier *(inherited from ModelElement)* — See Section 2.5.
- name : Base::String *(inherited from ModelElement)* — Identifies the artifact, answering the question, What is it called?
- reference : EscapedString — Identifies the artifact, answering the question, Which instance of <kind of thing> is it? Might be empty.

Semantics. An Artifact object identifies a thing. Two or more Artifact objects might identify the same thing or parts of the same thing. (A writer might model such relationships using the ArtifactRelationship class.)

Invariants. None. (But see Section 3.1 for invariants related to the Strict extension.)

Remarks (non-normative). None.
2.9 The ArtifactRelationship class

An ArtifactRelationship asserts a relationship between artifacts.

**Derivation.** The ArtifactRelationship class is derived from the ModelElement class. No classes are derived from the ArtifactRelationship class.

**Fields and relationships.** This class’s fields and relationships are:

- **annotation : KeyValuePair [0..*]** (*inherited from* ModelElement) — See Section 2.5.
- **description : EscapedString** (*inherited from* ModelElement) — A description the relationship between the artifacts identified by source and the artifacts identified by target.
- **id : Identifier** (*inherited from* ModelElement) — See Section 2.5.
- **name : Base::String** (*inherited from* ModelElement) — See Section 2.5.
- **source : Artifact [0..*]** (**reference**) — The Artifact objects on one side of the relationship.
- **target : Artifact [0..*]** (**reference**) — The Artifact objects on the other side of the relationship.

**Semantics.** Each source is related to each target as specified by the description.

**Invariants.** None. (But see Section 3.1 for invariants related to the Strict extension.)

**Remarks (non-normative).** The source and target sets are not required to be disjoint. This because some relationships are bi-directional. For example, a writer might document the fact that a set of artifacts were all produced by the same developer(s) using an ArtifactRelationship with source=target and a description reading, “Produced by same developer as.”

We considered adding additional inventory-related classes to record, e.g., the provenance of artifacts. But these relationships can be described to human readers using ArtifactRelationship objects. For example, one might model a process description as an Artifact and use an ArtifactRelationship to record that another artifact was “produced using the process described in” the former artifact. If a user wishes to automate support for re-examining artifacts that might have been impacted by a change, the extension and annotation mechanisms could be used to indicate which ArtifactRelationship objects signify dependence between Artifact objects.

2.10 The Inventory class

An Inventory defines a set of Artifacts and their ArtifactRelationships (if any).
Derivation. The Inventory class is derived from the ModelElement class. No classes are derived from the Inventory class.

Fields and relationships. This class’s fields and relationships are:

- annotation : KeyValuePair [0..*] (inherited from ModelElement) — See Section 2.5.
- artifact : Artifact [0..*] — Defines the set of Artifact objects in this Inventory.
- description : EscapedString (inherited from ModelElement) — See Section 2.5.
- id : Identifier (inherited from ModelElement) — See Section 2.5.
- name : Base::String (inherited from ModelElement) — See Section 2.5.
- relationship : ArtifactRelationship [0..*] — Defines the set of relationships amongst the inventory’s artifacts.

Semantics. An Inventory is a container of Artifact objects and Artifact-Relationship objects documenting the relationships amongst them. No universal meaning applies to the separation of Artifact objects into separate Inventory containers, but those containers’ description fields might indicate such a purpose.

Invariants. All instances must satisfy the following invariants:

1. If an Inventory i’s relationship set contains an ArtifactRelationship r with a source or target containing Artifact a, i’s artifact set must also contain a.

(See Section 3.1 for invariants related to the Strict extension.)

Remarks (non-normative). None.

2.11 The ArgumentElement class

ArgumentElement serves as a common base class for the classes used to model the contents of an Argument.

Derivation. The ArgumentElement class is derived from the ModelElement class. ArgumentElement is the abstract base class from which the Claim, EvidenceReference, and Inference classes are derived.

Fields and relationships. This class’s fields and relationships are:

- annotation : KeyValuePair [0..*] (inherited from ModelElement) — See Section 2.5.
• description : EscapedString (inherited from ModelElement) — A description of the ArgumentElement. User agents might display this as the element’s text content.
• id : Identifier (inherited from ModelElement) — See Section 2.5.
• name : Base::String (inherited from ModelElement) — The name of the ArgumentElement. User agents might display this as an identifier.
• toBeModified : Boolean = false — Indicates that the element requires further modification. For example, a pattern might set this field to true in one of its Claim objects to indicate that the user should modify the description to suit the application. A writer producing a draft assurance case might set this field to true to indicate that an Argument-Element might require revisiting later.
• toBeSupported : Boolean = false — Indicates that the element requires further support. For example, a pattern might set this field to true in one of its Claim objects to indicate that the user instantiating the element should supply premises that are appropriate for the application and link these via one or more Inference objects. A writer producing a draft assurance case might mark an Inference this way to indicate an intent to later supply a warrant and backing.

Semantics. The semantics of an ArgumentElement vary according to whether it is an Inference, EvidenceReference, or Claim.

Invariants. None. (But see Section 3.1 for invariants related to the Strict extension.)

Remarks (non-normative). None.

2.12 The Claim class

A Claim represents a proposition that the argument writer asserts is true. A claim might be assumed (i.e., deliberately not supported by evidence either directly or indirectly), a conclusion, or a premise.

Derivation. The Claim class is derived from the ArgumentElement class. No classes are derived from the Claim class.

Fields and relationships. This class’s fields and relationships are:
• annotation : KeyValuePair [0..*] (inherited from ModelElement via ArgumentElement) — See Section 2.5.
• description : EscapedString (inherited from ModelElement via ArgumentElement) — Specifies the proposition to be asserted.
• id : Identifier (inherited from ModelElement via ArgumentElement) — See Section 2.5.
• name : Base::String \((inherited \ from \ ModelElement \ via \ ArgumentElement)\) — See Section 2.5.

• toBeModified : Boolean = false \((inherited \ from \ ArgumentElement)\) — See Section 2.11.

• toBeSupported : Boolean = false \((inherited \ from ArgumentElement)\) — See Section 2.11.

**Semantics.** The existence of a Claim in an Argument indicates the writer’s contention that the proposition recorded in the description field of that Claim is true. A Claim is assumed (at least for the moment) if it does not appear in the target of at least one Inference with against=false.

**Invariants.** None. (But see Section 3.1 for invariants related to the Strict extension.)

**Remarks (non-normative).** None.

### 2.13 The EvidenceReference class

An EvidenceReference represents the citation of an evidence-related item as a premise.

**Derivation.** The EvidenceReference class is derived from the ArgumentElement class. No classes are derived from the EvidenceReference class.

**Fields and relationships.** This class’s fields and relationships are:

• annotation : KeyValuePair \([0..\ast]\) \((inherited \ from \ ModelElement \ via \ ArgumentElement)\) — See Section 2.5.

• description : EscapedString \((inherited \ from \ ModelElement \ via \ ArgumentElement)\) — A description of the evidence-related item. If this is blank and reference matches the id of a ModelElement, the description of that ModelElement can be taken as the item’s description.

• id : Identifier \((inherited \ from \ ModelElement \ via \ ArgumentElement)\) — See Section 2.5.

• reference : Identifier — The (optional) id of the evidence-related item in question—e.g., an Artifact or another Argument (if the item is represented in the model). Empty name and domain fields signify that referenced item is not modeled.

• name : Base::String \((inherited \ from \ ModelElement \ via \ ArgumentElement)\) — See Section 2.5.

• toBeModified : Boolean = false \((inherited \ from \ ArgumentElement)\) — See Section 2.11.

• toBeSupported : Boolean = false \((inherited \ from \ ArgumentElement)\) — See Section 2.11.
Semantics. Identifies an evidence-related item. The Inference in which the EvidenceReference appears as a source defines how the item serves as evidence [19].

Invariants. None. (But see Section 3.1 for invariants related to the Strict extension.)

Remarks (non-normative). It is tempting to require that if reference is empty, toBeSupported must be true. We deliberately do not make this restriction. Writers are free to simply describe the evidence item using the description field. Writers are also free to use the description field in addition to the reference relationship in order to cite only part of a modeled item.

We considered allowing EvidenceReference objects to refer to specific elements contained by other Argument objects. This is disallowed in the Strict extension (see Section 3.1) so as to force readers to consider the context of both arguments when determining whether one supports a claim made in the other.

2.14 The Inference class

An Inference represents an inference from premises, which might comprise evidence, assumptions, or claims backed by further argument.

Fields and relationships. This class’s fields and relationships are:

- against : Boolean = false — Indicates that the premises tend to show that all claims in the target set are false and any inferences in the target set do not hold.
- annotation : KeyValuePair [0..*] (inherited from ModelElement via ArgumentElement) — See Section 2.5.
- description : EscapedString (inherited from ModelElement via ArgumentElement) — Describes the nature of the inference (i.e., how the premises support the claims and inferences identified by target. For example, in an inference from one or more EvidenceReference objects to a Claim, the description might identify or describe an evidence scheme [19].
- id : Identifier (inherited from ModelElement via ArgumentElement) — See Section 2.5.
- name : Base::String (inherited from ModelElement via ArgumentElement) — See Section 2.5.
- toBeModified : Boolean = false (inherited from ArgumentElement) — See Section 2.11.
- toBeSupported : Boolean = false (inherited from ArgumentElement) — See Section 2.11.
**Semantics.** The assertion of an *Inference* from the elements in `source` to the elements in `target` indicates the writer’s contention that the sources collaboratively support (or weigh against) each `target` in the same way. Premises that support a `target` or targets in similar fashion should do so through separate *Inference* objects, even if the support is not fully independent. That is, the removal of a premise from the `source` set should change what is known about the nature of the inference, not merely alter the confidence it lends (or takes from) the targets. For a comparison to Toulmin’s model of argumentation, see Appendix B.

**Invariants.** All instances must satisfy the following invariants:

1. The `source` set must not contain any `Inference`s.
2. The `target` set must not contain any `EvidenceReference`s.
3. If `source` is empty, `toBeSupported` must be true.
4. If `target` is empty, `toBeModified` must be true.

(See Section 3.1 for invariants related to the Strict extension.)

**Remarks (non-normative).** It is tempting to assert that the `source` set and `target` set should be homogenous (e.g., contain only `Claim` objects) on the grounds that inference from or two a heterogeneous set seems likely to be the result of an error or misuse. We make no such prohibition. However, readers and reviewers are reminded that *Inference* objects are meant to record coherent, collaborative support (or challenge). We cannot conclude that there are no inferences from mixed sources that would qualify, but if there are, we would not preclude recording them.

### 2.15 The Argument class

The *Argument* class represents a self-contained argument or argument pattern. *Argument* objects may reference artifacts and terms defined in the same *AssuranceCase*.

**Derivation.** The *Argument* class is derived from the *ModelElement* class. No classes derive from the *Argument* class.

**Fields and relationships.** This class’s fields and relationships are:

- `annotation : KeyValuePair [0..*]` (*inherited from* `ModelElement`) — See Section 2.5.
- `background : Claim [0..*]` (*reference*) — The set of claims—assumed or supported—to be taken as true throughout the argument.
- `content : ArgumentElement [0..*]` — The set of claims, evidence references, and instances that the argument comprises.
- `description : EscapedString` (*inherited from* `ModelElement`) — A description of the argument or argument pattern.
Semantics. The argument should be interpreted as if the background claims were true. If the assumed claims—either in the background or asserted in the argument—are true, the Argument should be taken as support for its non-assumed claims. The nature and degree of the support for a given claim depends on the truth or assumptions and the strength of the evidence and inferences that directly or indirectly support that claim.

Invariants. All instances must satisfy the following invariants:

1. background must be a subset of content.

Remarks (non-normative). The Argument class lacks identification for 'top-level' Claim objects. If it is important to identify a particular Claim as having particular significance within a system safety assurance or certification effort, this can be done with its id or description or a Note annotation (if present).

The Argument class also lacks any means of distinguishing between 'public' (i.e., can be referenced from other arguments) and 'private' (i.e., cannot be referenced) elements. This distinction is not meaningful since arguments can only cite other arguments in toto.

2.16 The AssuranceCase class

AssuranceCase is the main container of the Simple Assurance Argument Interchange Format. Each AssuranceCase represents a collection of arguments and evidence.

Derivation. The AssuranceCase class is derived from the ModelElement class. No classes derive from the AssuranceCase class.

Fields and relationships. This class’s fields and relationships are:

- annotation : KeyValuePair [0..*] (inherited from ModelElement) — See Section 2.5.
- description : EscapedString (inherited from ModelElement) — See Section 2.5.
- extension : Identifier [0..*] — Identifies any extensions that the model implements. See Section 3 for the definition of extensions.
- glossary : Glossary [0..*] — The glossary or glossaries for terms used in the assurance case. When multiple glossaries are supplied, all definitions are available for use in all arguments within the case.
- id : Identifier (inherited from ModelElement) — An optional Identifier for the assurance case as a whole. Used to refer to cases defined in other models. May have empty domain and name fields.
• inventory : Inventory [0..*] — The collection(s) of Artifact objects that may be referred to by the case’s argument(s). Where multiple inventories are supplied, all artifacts are available for reference from all arguments within the case.

• locale : Base::String — The ISO 15897 locale for the assurance case (e.g., “en_US”). May not be empty.

• name : Base::String (inherited from ModelElement) — See Section 2.5.

Semantics. Each AssuranceCase represents an assurance case, complete with definitions of any terms of art and references to any cited evidence. The scope of an assurance case is not limited to any particular stakeholder’s perspective, the whole or part of any system or service, or any particular dependability properties. The description and any notes (see Section 2.5) may be used to describe the scope of the assurance case for the reader’s benefit.

Invariants. None. (But see Section 3.1 for invariants related to the Strict extension.)

Remarks (non-normative). Having only one locale for the entire assurance case implies that all text within it is treated by the rules of a single language. This does not preclude including content from other languages. For example, an argument written in English might include German artifact names.

This restriction effectively precludes including translations of a single case within the same model. This is by design. Were cases to include content in multiple languages, the cost would be at least two forms of additional complexity: First, the model elements themselves would require locale fields. Second, and more importantly, the model would require a mechanism for guiding arbitration of meaning by identifying which version of the content was canonical and which versions are translations of that canonical content.

3 Extensions

Extensions are declarations that the writer can add to a model to indicate the adoption of additional invariants or model elements. Coupled with the ability to define arbitrary KeyValuePair annotations, extensions allow users to customize and extend the format. For example, one could define an extension for argument reviewing that used annotations to record which argument steps had been reviewed and record any issues noted during review. The SAAIF has one built-in extension: SAAIF.Strict.

3.1 SAAIF.Strict

The SAAIF model described in Section 2 deliberately permits models with detectable structural flaws such as circular arguments. This permissiveness serves an important design aim, namely allowing the writer to record work
in progress. The SAAIF.Strict extension includes additional invariants that preclude these structural flaws.

An AssuranceCase whose extension set includes an Identifier matching SAAIF.Strict must satisfy the following additional invariants:

1. The AssuranceCase must have at least one Argument, at least one Glossary, and at least one Inventory, but each of these may be empty.
2. The Identifier of each identified object must be unique. That is, no two identifiers used as (i) a Definition’s key or (ii) a ModelElement’s id may match.
3. Term definitions must be unique and well-defined. That is:
   (a) A Definition object’s name and description fields must not be empty.
   (b) No two Definition objects in the same Glossary may have the same name.
   (c) A Definition object’s description may not refer to another term that is directly or indirectly defined in terms of that definition. (Mechanical checking of strict conformance should interpret the appearance of an Identifier matching term a in term b’s Definition as b being defined in terms of a.)
4. Inferences must be valid. That is:
   (a) An Inference object’s source set must not be empty.
   (b) An Inference object’s target set must not be empty.
   (c) No two ArgumentElement objects may participate in a circular reference. More formally, \( \neg \text{supports}(a, a) \) where \( \text{supports}(a, b) \) indicates that either (a) there exists an Inference with \( a \in \text{source}, b \in \text{target}, \text{and against}=false \), or (b) there exists a \( c_1..c_n \) (for \( n \geq 1 \)) such that \( \text{supports}(a, c_1) \land \text{supports}(c_i, c_{i+1}) \land \ldots \land \text{supports}(c_n, b) \).
5. Concrete, specific evidence references must identify specific, real Artifact objects or Argument objects for support. That is, if an EvidenceReference is not toBeSupported and has a non-empty reference, its reference must match the id of an Artifact or Argument.
6. Artifact relationships must be well-specified. That is, an ArtifactRelationship object’s source and target sets must not be empty.
7. All patterns must be fully instantiated. That is:
   (a) The SAAIF.IsPattern annotation may only be attached to Argument and Glossary objects.
   (b) No ModelElement may have an annotation with a key matching SAAIF.InstantiationNote unless one of the following is true:
      i. The ModelElement is a Glossary and it is also annotated with SAAIF.IsPattern = \{true\}.
      ii. The ModelElement is a Definition and its Glossary is also annotated with SAAIF.IsPattern = \{true\}.
iii. The ModelElement is an Argument and it is also annotated with SAAIF.IsPattern = {true}.

iv. The ModelElement is an ArgumentElement and its Argument is also annotated with SAAIF.IsPattern = {true}.

8. The value fields of KeyValuePair objects with pre-defined keys should have the correct types. That is:

(a) The value field of any KeyValuePair with a key matching SAAIF.InstantiationNote must not contain nothing other than EscapeStringValue objects.

(b) The value field of any KeyValuePair with a key matching SAAIF.IsPattern must contain exactly one BooleanValue.

(c) The value field of any KeyValuePair with a key matching SAAIF.Note must not contain nothing other than EscapeStringValue objects.

If an AssuranceCase object’s extension set includes an Identifier matching SAAIF.Strict, it should also meet the following goals where practicable:

1. Define terms of art clearly. In addition to avoiding circular definitions, writers should (a) avoid defining words in terms of words the reader is even less likely to understand, (b) refrain from using non-predictive constructions such as defining terms using examples or using hedges such as ‘usually’ or ‘generally’ [20].

2. Avoid the use of loaded language [21]. That is, writers should not use defined terms in such a way that knowledge of their plain-language meaning is likely to mislead readers.

3. Where practical\(^2\), an argument’s references to propositions and artifacts should be recorded using Inference objects and EvidenceReference objects rather than through the EscapedString mechanism.

4 Encoding

The SAAIF model presented in Section 2 might be encoded in stream or file formats not specified in this document. For the purpose of facilitating interchange, we define one standard encoding, SAAIF_TEXT. This encoding is defined in Appendix E.

---

\(^2\) There are some instances where it might be desirable to use an EscapedString reference rather than an Inference or EvidenceReference. For example, one might use a Note to observe that one Claim is analogous to another Claim in the same argument or in another argument.
References


Appendix A

Mapping to the Goal Structuring Notation

The Goal Structuring Notation (GSN) is a popular graphical notation for recording assurance arguments [5]. While it is not possible to convert arguments between SAAIF and GSN without some loss of information or refactoring, it is useful for pedagogical purposes to explain which concepts in GSN align with which concepts in the SAAIF and vice-versa. Table A1 sketches such a mapping.

<table>
<thead>
<tr>
<th>GSN Goal</th>
<th>SAAIF</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strategy</td>
<td>description of Inference</td>
<td></td>
</tr>
<tr>
<td>Solution</td>
<td>Evidence-Reference</td>
<td></td>
</tr>
<tr>
<td>Context (as definition [21])</td>
<td>Definition</td>
<td>Definitions apply case-wide.</td>
</tr>
<tr>
<td>Context (as background)</td>
<td>Claim in the background of an Argument</td>
<td></td>
</tr>
<tr>
<td>Context (as artifact)</td>
<td>?</td>
<td>Depends on the purpose of asserting the artifact [21].</td>
</tr>
<tr>
<td>Justification as context for strategy</td>
<td>Inference from Claim to Inference</td>
<td>The Claim gives the justification text; the target Inference represents the strategy.</td>
</tr>
<tr>
<td>Assumption</td>
<td>Claim</td>
<td>Claim without support.</td>
</tr>
<tr>
<td>Undeveloped entity</td>
<td>toBeSupported</td>
<td></td>
</tr>
<tr>
<td>SupportedBy</td>
<td>Inference</td>
<td>Except that the goal-supported-by-strategy-supported-by-goal construct is represented as a described Inference from the first Claim to the second.</td>
</tr>
<tr>
<td>InContextOf</td>
<td>No direct equivalent</td>
<td>See mappings for Context and Away Goals.</td>
</tr>
<tr>
<td>Uninstantiated entity</td>
<td>toBeModified</td>
<td></td>
</tr>
<tr>
<td>Number/choice decorations</td>
<td>Instantiation-Note</td>
<td>(See Section 2.4.)</td>
</tr>
</tbody>
</table>
Table A1: Mapping between the GSN and the SAAIF (continued).

<table>
<thead>
<tr>
<th>GSN</th>
<th>SAAIF</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bracketed text</td>
<td>Definition in Glossary that IsPattern</td>
<td>Placeholder text in patterns can refer to associated definitions.</td>
</tr>
<tr>
<td>Away goal</td>
<td>Inference from other Argument to Claim</td>
<td>Claims in other arguments can’t be referenced directly; the reader must determine whether and how the referenced Argument supports the Claim.</td>
</tr>
<tr>
<td>Module reference</td>
<td>Evidence-Reference to Argument</td>
<td></td>
</tr>
<tr>
<td>Away solution</td>
<td>Evidence-Reference to same Artifact</td>
<td></td>
</tr>
<tr>
<td>Away context</td>
<td>No direct equivalent</td>
<td>Can be duplicated depending on how the context is being used.</td>
</tr>
<tr>
<td>Public entity</td>
<td>No direct equivalent</td>
<td>A Claim in one Argument cannot be directly referenced from another Argument.</td>
</tr>
</tbody>
</table>
Appendix B

Mapping to Toulmin’s model

Toulmin’s classic text defines a model for informal argumentation that is still used as a model today [9]. Figure B1 depicts Toulmin’s model. While it is not possible to convert arguments between SAAIF and Toulmin’s model without some loss of information or refactoring, it is useful for pedagogical purposes to explain which concepts in Toulmin’s model align with which concepts in the SAAIF and vice-versa. Table B1 sketches such a mapping.

![Toulmin's model of informal argumentation.](image)

**Figure B1:** Toulmin’s model of informal argumentation.

<table>
<thead>
<tr>
<th>Toulmin’s model</th>
<th>SAAIF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Claim or EvidenceReference</td>
</tr>
<tr>
<td>Warrant</td>
<td>Inference from Claim to Inference</td>
</tr>
<tr>
<td>Backing</td>
<td>Inference to the Claim presenting the warrant</td>
</tr>
<tr>
<td>Qualifier</td>
<td>Text in the description</td>
</tr>
<tr>
<td>Rebuttal</td>
<td>Inference with against=true to the Claim</td>
</tr>
<tr>
<td>Claim</td>
<td>Claim</td>
</tr>
</tbody>
</table>

**Table B1:** Mapping between Toulmin’s model and the SAAIF.
Appendix C

Mapping to Confidence Maps

Some argument notations, including the Goal Structuring Notation (GSN), lack an explicit mechanism for presenting counter-evidence [5]. To address this lack, Goodenough et al. propose eliminative argumentation in the form of confidence maps [22]. Figure C1 depicts a confidence map, which introduces to familiar GSN symbols representing inference rules, the rebutting, undercutting, and undermining defeaters familiar to students of informal argumentation, and the writer’s assertion that something is either deductively OK or assumed OK. Table C1 sketches a mapping from confidence maps to the SAAIF.

Table C1: Mapping between confidence maps and the SAAIF.

<table>
<thead>
<tr>
<th>Confidence map</th>
<th>SAAIF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inference rule</td>
<td>Inference from Claim (expressing the rule) to Inference</td>
</tr>
<tr>
<td>Rebutting defeater</td>
<td>Inference with against=true from Claim expressing the reason to Claim expressing the rebutted claim.</td>
</tr>
<tr>
<td>Undercutting defeater</td>
<td>Inference with against=true from Claim expressing the reason to the rebutted Inference.</td>
</tr>
<tr>
<td>Undermining defeater</td>
<td>Either (1) an Inference with against=true from Claim expressing the reason to the rebutted EvidenceReference or (2) an Inference with against=true from Claim expressing the reason to the Inference from the rebutted evidence.</td>
</tr>
<tr>
<td>Assumed/is OK</td>
<td>No direct equivalent. A writer might mark an ArgumentElement as toBeSupported to indicate that it should be revisited. Readers may use their own annotations to record their acceptance of or confidence in claims, evidence references, and inferences.</td>
</tr>
</tbody>
</table>
The system is acceptably reliable if $pft < 0.001$ (with 99% statistical certainty).

Unless at least one failure is observed in a sequence of 4,603 operationally random test executions, then the system is acceptably reliable.

Unless at least one failure is observed in a sequence of 4,100 operationally random executions, then the system is acceptably reliable.

Unless at least one failure is observed in a sequence of 503 additional operationally random test executions, then the system is acceptably reliable.

Unless an error exists in the system, then the system is acceptably reliable.

Unless statically detectable coding errors exist, then the system is acceptably reliable.

Unless other kinds of errors exist, then the system is acceptably reliable.

Static analysis results showing no statically detectable coding errors.

But the static analysis overlooked some statically detectable errors because...

Analysis tools used don't detect some errors.

Not all code has been analyzed.
Appendix D

Example: Explicate ’78 revisited

Researchers have developed an assurance case representing the logic of RTCA’s DO-178C standard for airborne software [23, 24]. In this section, we present a re-imagining of part of that argument in SAAIF as an example of how SAAIF might be used.

Figure D1 reproduces Figure 3 (“Level D primary argument in GSN”) from a paper reporting a draft of that argument [23]. The figure is presented in a variant of the Goal Structuring Notation used by a specific argument editing tool [5, 25]. The argument is not a traditional safety argument, but rather an attempt to capture a standard’s logic, and so it differs from traditional software safety arguments in, e.g., referring to requirements and evidence in the abstract.

Figures D2–D4 give the argument as modeled using the SAAIF. Because SAAIF does not have a canonical graphical form, we illustrate the SAAIF model using a UML instance diagram. While this graphical form is somewhat bulky and impenetrable, we use it here to directly illustrate how the SAAIF’s fields and relationships might be used in modeling.

Figures D2–D4 are not a direct translation of Figure D1. Several differences are worthy of special mention:

- The GSN argument uses context elements to define terms of art. The SAAIF argument uses explicitly modeled definitions instead (permitting a user agent to render the defined terms as, e.g., hyperlinks).
- The GSN argument records as contextual information the assertion that “the software has been assigned to level D.” The SAAIF argument instead records this as a background assumption. If this assumption is not true, the argument becomes irrelevant.
- The (modified) GSN argument uses an assurance claim point (rendered as a box decorating the strategy) to link to a confidence argument (not shown). The SAAIF argument instead introduces the confidence claim as background supported by the confidence argument. This background knowledge should shape the reader’s interpretation of this argument, including the strength of inferences from evidence.
- The GSN argument renders objectives requiring the development of artifacts such as the high-level requirements as contextual information. The SAAIF argument records these objectives instead as inferences to the argument’s main inference. If high-level requirements do not exist, the main inference makes no sense.
1.1: SwAcceptableLevD
Software performs its intended function at acceptable level of safety for level D

1.2: IntFun
Description of intended function of the software

1.3: DefAwReg
Definition of acceptable level of safety from airworthiness regulations

1.4: LevelDAssign
The software has been assigned to level D

2.1: MeaningLevel D
Description of the meaning of level D. "Software whose anomalous behavior, as shown by the system assessment process, would cause or contribute to a failure of system function resulting in a minor failure for the aircraft." (2.3.3.d).

2.1: MeaningAnomBeh
"Anomalous behavior: Behavior that is inconsistent with specified requirements" (Glossary, p. 109.)

2.3: ArgByCorrectness
Argue by correctness of the software relative to allocated system requirements and derived requirements

2.4: ReqAllocValidStuff
System requirements allocated to software augmented by any derived requirements are valid and sufficient to define intended function and ensure acceptable level of safety

2.5: HLRDev
"High-level requirements are developed" (A-2.1)

2.5: DerHLProv
"Derived high-level requirements are defined and provided to the system processes, including the system safety assessment process" (A-2.2)

3.1: References
“The relationship of the requirements development process to the safety process [is] defined to ensure that the safety analysis [is] not compromised by either the improper implementation of safety-related requirements or the introduction of new behavior (that is, derived requirements) that was not envisioned in the original safety analysis” DO-248C 5.4 bullet 6

3.2: References
5.1.1.a; Activities 5.1.2.a, 5.1.2.b, 5.1.2.c, 5.1.2.d, 5.1.2.e, 5.1.2.f, 5.1.2.g, 5.1.2.j, 5.5.a; Glossary; 248C 5.5.1

3.4: Module HLRSatLevD
High-level requirements are a satisfactory (for level D) refinement of the allocated system requirements

3.5: Module EOCSatLevD
Executable Object Code is a satisfactory (for level D) refinement of the high-level requirements

Figure D1: Reproduction of "Level D primary argument", from 2.3 in modified GSN.
Figure D2: Adaptation of “Level D primary argument” from [23] (part 1 of 3).
Figure D3: Adaptation of “Level D primary argument” from [23] (part 2 of 3).
Note: Claims HLR_Dev and DevHLRProv were originally rendered as context. Here, we render them as evidence to show how reference as-yet-undeveloped evidence. Their appearance in two figures is meant only to permit splitting the argument over several pages; they would not be duplicated.

Figure D4: Adaptation of “Level D primary argument” from [23] (part 3 of 3).
Appendix E

Text-based encoding SAAIF

It is useful to have at least one canonical file or stream format for encoding SAAIF arguments. This section presents a text file encoding, SAAIF.

E.1 Character encoding

The characters that comprise a SAAIF file shall be encoded in UTF-8.

E.2 Lexical structure

A SAAIF file comprises the following tokens:

Bareword. A sequence of one or more characters from the same set that identifiers comprise, namely \{'a',\ldots,'z','A',\ldots,'Z','0',\ldots,'9','_'}.

Identifier. An identifier comprises two parts separated by a period ('.'). Each part might be either a bareword or an asterisk ('*'). The first part is the domain field. The second part is the name field. An asterisk represents an empty field.

UID. A sequence of 1–16 uppercase or lowercase hexadecimal digits.UIDs need not be truly unique. (See Section E.4 for details.)

Symbol. Symbols—operators and punctuation—include '='; ';' ‘(’, ‘)’, ',', and '+'.

String literal. A string literal begins with a double-quote character ('"'), contains any characters other than a newline or an unescaped double-quote, and ends with a double-quote. Escape sequences begin with a pipe character ('|'):

- '| |' represents a single pipe character.
- '| "' represents a double-quote character.
- '| n' represents a newline character.
- '| ' followed by one or more hexadecimal digits represents the unicode code point with the given value.

Escaped string literal. A backslash character ('\') followed immediately by what would otherwise be a string literal.

Comment. A comment is a pair of hyphen characters ("--") followed by all non-newline characters up to the next newline.

Whitespace. The whitespace characters are space, horizontal tab, newline, and carriage-return appearing outside of a string literal or escaped string literal. Whitespace is ignored.
E.3 Grammar

We present the SAAIF_TEXT grammar in Backus–Naur Form (BNF) notation with regular expression extensions. Words appearing in all upper case are nonterminal names. SAAIF_TEXT_FILE represents a complete file. Words appearing in all lower case are token names. Token literals appear in quotes and are not case sensitive. IMPORT_STATEMENTs should be treated as though they were replaced by the contents of the file whose path is given by the string literal. A single VALUE appearing in a PARAMETER is interpreted identically to a VALUE_SET with that single value.

```
SAAIF_TEXT_FILE := ASSURANCE_CASE*;
ASSURANCE_CASE := 'AssuranceCase' identifier 'is' PARAMETER+ 'begin' AC_CONTENT* 'end' ';';
AC_CONTENT := IMPORT_STATEMENT | GLOSSARY | INVENTORY | ARGUMENT;
PARAMETER := (bareword | identifier) '=' (VALUE | VALUE_SET) ';';
VALUE_SET := '{' (VALUE (',' VALUE)* )? '}'
VALUE := STRING | ESC_STRING | BOOLEAN | identifier | uid;
STRING := string ('+' string)*;
ESC_STRING := escaped_string ('+' escaped_string)*;
BOOLEAN := 'true' | 'false';
IMPORT_STATEMENT := 'import' string ';';
GLOSSARY := 'Glossary' identifier 'is' PARAMETER+ 'begin' DEFINITION* 'end' ';';
DEFINITION := 'Definition' identifier 'is' PARAMETER+ 'end' ';';
INVENTORY := 'Inventory' identifier 'is' PARAMETER+ 'begin' I_CONTENT* 'end' ';';
I_CONTENT := ARTIFACT | ARTIFACT_RELATIONSHIP;
ARTIFACT := 'Artifact' identifier uid 'is' PARAMETER+ 'end' ';';
ARTIFACT_RELATIONSHIP := 'ArtifactRelationship' identifier 'is' PARAMETER+ 'end' ';';
ARGUMENT := 'Argument' identifier 'is' PARAMETER+ 'begin' ARGUMENT_ELEMENT* 'end' ';';
ARGUMENT_ELEMENT := CLAIM | EVIDENCE_REF | INFERENCE;
CLAIM := 'Claim' identifier uid 'is' PARAMETER+ 'end' ';';
EVIDENCE_REF := 'EvidenceReference' identifier uid 'is' PARAMETER+ 'end' ';';
```
E.4 Semantics

In this section, we describe the relationship between the SAAIF TEXT grammar and the SAAIF model.

**ModelElement.** Each ASSURANCE_CASE, GLOSSARY, INVENTORY, ARTIFACT, ARGUMENT, CLAIM, EVIDENCE_REF, or INFERENCE nonterminal encodes a ModelElement of the kind indicated by its initial token. The fields of a ModelElement are generally given by PARAMETER nonterminals appearing before the next “begin” token. The description and name fields of a ModelElement are given by PARAMETER nonterminals with the bareword tokens “description” and “name”, respectively. Its id field is given by the identifier immediately following its initial token. Its annotation field is given by the set of PARAMETER nonterminals beginning with “identifier” tokens. Unless otherwise specified, the fields of derived classes are given by PARAMETER nonterminals with corresponding bareword tokens.

**AssuranceCase.** The part of each ASSURANCE_CASE nonterminal appearing between its first “begin” token and its last “end” token gives its inventory, argument, and glossary contents in the form of the GLOSSARY, INVENTORY, and ARGUMENT nonterminals that appear there.

**Glossary.** The part of each GLOSSARY appearing between its first “begin” token and its last “end” token gives the entry field for the Glossary it represents.

**Inventory.** The part of each INVENTORY appearing between its first “begin” token and its last “end” token gives the relationship and artifact fields for the Inventory.

**Artifact.** Each ARTIFACT in a given INVENTORY must have a unique uid.

**ArtifactRelationship.** The source and target fields of each ArtifactRelationship will be given as PARAMETER nonterminals with the bareword tokens “source” and “target”. Those PARAMETER nonterminals will have VALUE_SET nonterminals comprising only uids, each giving the uid of an ARTIFACT in the same INVENTORY.

**Argument.** The part of each ARGUMENT appearing between its first “begin” token and its last “end” token gives the content field for the argument Argument it represents. The Argument’s background field will be represented by a PARAMETER with the bareword “background” and a VALUE_SET comprising only uids, each of which identifies an ARG_CONTENT in the same ARGUMENT.
ArgumentElement. Each ARGUMENT_ELEMENT in a given Argument must have a unique uid.

Inference. The source and target fields of each Inference will be given as PARAMETER nonterminals with the bareword tokens "source" and "target". Those PARAMETER nonterminals will have VALUE_SET nonterminals comprising only uid tokens, each giving the uid of an ARGUMENT_ELEMENT in the same ARGUMENT.

E.5 Example

The following example illustrates how one might encode the argument fragment depicted in Figure D2.

```plaintext
AssuranceCase Explicate_78.Explicate_78 is
  extension = { SAAIF.Strict };
  locale = "en_US";
  name = "Explicate '78";
begin
Glossary DO1788.Glossary is
  description = "The $DO178.DO178C$ Glossary";
  name = "DO178 Glossary";
  SAAIF.IsPattern = true;
begin
  Definition DO178.Software is
    description = "Computer programs ...";
    name = "Software";
  end;

  Definition DO178.Intended_Function is
    description = "";
    name = "Indended Function"
    SAAIF.InstantiationNote = { "Replace with " +
                              "a reference to a description of the " +
                              "$DO178.Software:software's$ defined " +
                              "intended function." };
  end;

  Definition DO178.Level_D is
    description = "$DO178.Software$ whose " +
                  "$DO1789.Anomolous_Behavior$, as shown " +
                  "by the system assessment ...";
    name = "Level D";
  end;

  -- Some definitions elided
end;
```
Glossary AWR.Glossary is
description = "Terms from the applicable airworthiness regulations."
name = "AWR Glossary"
begin
  -- Definitions elided
end;

Argument Level_D_Primary.Argument is
description = "Primary argument for $DO178.Level_D$ $DO178.Software$"
name = "Level D Primary"
background = { 1 , 2 };
begin
  Claim Level_D_Primary.Lev_D_Assigned 1 is
description = "The $DO178.Software$ has been assigned to $DO178.Level_D$.
name = "Lev D Assigned"
end;

Claim Level_D_Primary.Justified_Confidence_Lev_D 2 is
description = "The evidence provided is adequate for justifying confidence that the correctness of the $DO178.Software$ has been demonstrated to the extent needed for $DO178.Level_D$.
name = "Justified Confidence Lev D"
end;

Inference Level_D_Primary.Justified_Confidence_Lev_D_I 3 is
description = ""
name = "Justified Confidence Lev D I"
source = { 4 };
target = { 2 };
end;

EvidenceReference Level_D_Primary.Jus_Conf_Lev_D_ER 4 is
description = "$JusConfLevD.Argument"
name = "Jus Conf Lev D ER"
end;

-- Some argument elements elided
end;

-- Some assurance case elements elided
end;
This document describes the Simple Assurance Argument Interchange Format, a proposed meta-model for describing structured assurance arguments. We describe the syntax and semantics of the model elements, compare the meta-model to existing argument formats, and give an example to illustrate its use.