The Visual Inheritance Structure to Support the Design of Visual Notations

Nungki Selviandro*†, Tim Kelly*, Richard Hawkins*
*Department of Computer Science, University of York, UK
†School of Compputing, Telkom University, Indonesia
{ns1162, tim.kelly, richard.hawkins}@york.ac.uk

***Abstract*—It is a common practice in modelling languages to provide their users with a set of visual notations as a representation of semantic constructs. The use of visual notation is believed to help communicate complex information, especially when communicating with non-technical users. Therefore, research in the design of visual notation continues to evolve, e.g. research to provide an effective and efficient design approach. There are approaches exist to support the designer in designing the visual notation such as the Cognitive Dimensions and the Physics of Notations. As the current metamodeling approach is widely adopted as a mechanism for improving standardisation and interoperability in modelling languages, it is important to provide a guideline that focus on the design of visual notation for a predefined metamodel. In this paper, we address the visual inheritance structure to support the design of visual notations for a predefined metamodel. This approach emphasises the design coherence between classes and sub-classes. To demonstrate that it is possible to apply our approach, we use part of the OMG Structured Assurance Case Metamodel as a case study.**

I. Introduction

In developing a system model using a particular modelling language, it is essential for the designer to understand the semantic constructs, the compositional rules, and the constructs representation (e.g. in textual or visual forms). It is now a common practice in modelling languages to provide users with a set of visual notations for the representation of the semantic constructs. UML, for example, as a general-purpose modelling language, has a visual notation to help the designer visualise system models.

The visual communication aspect itself in modelling languages is crucial. It can be seen as the interface of the modelling language and users, and helps communicate complex information, especially when communicating with non-technical users. A well-designed visual notation is also easier to learn and to remember than textual syntax [1], [2]. To be more effective, the design of the visual notation should intuitively represent the semantic constructs [3], [4].

In the visual modelling language development process, the semantic constructs and the language rules are typically defined in a metamodel by the language developers together with the end-users, who help to refine the language concepts. Based on the defined metamodel, notation will be developed as a representational form of the language [3]. The design of the notation should be accompanied by a design rationale, that is, explicit reference to theories and empirical evidence for designing the notations [5]. Some approaches exist to support the designer in designing the visual notation such as proposed in [3], [6], [7]. In this paper, we address the visual inheritance to support the design of a visual notation based on a metamodel. We show the applicability of this approach by designing a set of visual notation for the Argumentation Metamodel of the OMG Structured Assurance Case Metamodel (SACM) [8]. This metamodel describes and defines the concepts that are required to model structured arguments (e.g. for safety). However, the current version of the SACM Argumentation specification is not equipped with any visual notation.

Supported by Indonesia Endowment Fund for Education (LPDP).

The remainder of this paper is organised as follows: Section 2 presents related work on the visual notation design theory, visual inheritance design, and the concept of structured arguments in SACM. Section 3 describes the visual inheritance notation design. In Section 4, we discuss the applicability of the proposed approach. In Section 5, we describes the limitations to the current version of the proposed approach. Finally, Section 6 gives conclusions and discusses future work.

II. Related Work

A. *State-of-the-Art of Visual Notation Design Theory*

Visual notations form an integral part of the language of computer science and software engineering. Many of visual notations have been developed to facilitate communicating complex information between technical and non-technical users. They are also used in different stages of software engineering, from requirements engineering through to maintenance. Visual notations are also adopted by practitioners in industry for strategic planning in the design of software systems. In industry, visual notations play a critical role in communicating with internal and external stakeholders [9]. This encourages visual notation researchers to conduct studies as to how to maximise the effectiveness of visual notations.

Research in notation design is closely associated with research in conceptual modelling languages, specifically in evaluating the quality of the conceptual model [10]. Several frameworks have been proposed to evaluate and improve the quality of conceptual models, and some of them, such as SEQUAL [11] and GoM [12], partially paid attention to the visual notation aspect. However, they were not designed exclusively to focus on the visual notation.

Recently, a theory that focuses on visual notation design has been proposed, which is called The Physics of Notations (PoN). This theory aimed to ensure that visual notations are designed to be cognitively effective (i.e. the speed, ease, and accuracy with which a representation can be processed by the human mind) [3]. PoN provides nine principles that can be used either to design or to evaluate the visual notation. There are numbers of existing notations that has been evaluated using the PoN principles such as UML [13], I* [14], SEAM [9], UCM [15], and BPMN [5]. There are also several new notations that are designed using the PoN principles, such as VTML (The Visual Traceability Modelling Language) [16] and Larman's Operation Contracts [17]. However, this framework has received some criticism in the literature, mainly focused on the validation and difficulty in applying the principles since PoN does not prescribe any systematic method for the implementation. Several works, such as in [18]–[20], have been proposed to address this issue.

Besides PoN, the Cognitive Dimensions (CDs) theory is also frequently discussed in notation design literature. CDs framework was introduced by Green, who emphasised the use of this framework as a "discussion tool" to aid non-HCI specialists in evaluating the usability of notational systems and information artifacts [6]. To do this, the CDs provides a vocabulary that can be used by designers when investigating the cognitive implications of their design decisions. Therefore, it does not explicitly provide any guideline or procedure for designing or evaluating a visual notation [21].

With the advancement of meta-modelling standards, the semantic constructs and grammatical rules of modelling languages tend to be defined in the metamodel to facilitate communication and tool support [14]. The notation should be defined afterwards referring to the defined metamodel. Therefore, in this paper we focus on providing a guideline to support the design of the notations based on a metamodel.

B. Visual Inheritance

The term *inheritance* is commonly used in object-oriented methodology. Inheritance can be defined as a mechanism which allows new classes to be defined based on existing classes; a new class can be defined as a specialisation of one that has already been defined. In this case, the specialised class inherits the characteristics (attributes and methods) of the class it is created from. This specialised-class is also known as a sub-class or child-class, while the existing class that inherits its characteristics is also known as a general-class, super-class, or parent-class [22].

There are two types of inheritance: single and multiple [22], [23]. Single inheritance can be described as a condition where a child-class has exactly one parent-class. For example, in Fig. 1(a), a child-class "B" has only one parent-class "A". Multiple inheritance can be described as a condition where a child-class has more than one parent-class. However, multiple inheritance tends to be hard to implement due to ambiguity. For example, Java programming language does not support the multiple inheritance mechanism since it could cause ambiguity

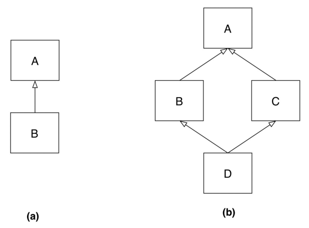

Fig. 1. (a) Single Inheritance; (b) Multiple Inheritance

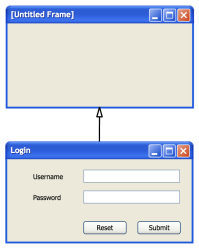

Fig. 2. An Example of Visual Inheritance in UI Design

when compiling the program. A popular case of this ambiguity is known as a 'diamond problem'. The diamond problem can be illustrated as an ambiguity that arises when a parent-class A inherits its attributes and methods to child-classes B and C, while there is a sub-class D that inherits from both class B and C (see Fig. 1(b)). In this case, the problem occurs when there exist methods with the same signature in both parent and child-classes. On calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority [22].

In user interface (UI) object-oriented programming, the concept of inheritance is also being adopted to maximise efficiency. For example, in the case when we have a common (base) design that can be reused further by inheriting this base design [24], [25]. For example, in designing a UI frame, we can create the base design as a rectangle that has common functionalities (methods), e.g. close. This frame base design and functionalities can be reused by more specialised frames (e.g. login frame) by implementing the inheritance mechanism. This process can be called visual inheritance. An illustration of this is provided in Fig. 2.

The concept of inheritance has inspired the idea of visual inheritance design in designing visual notations based on the metamodel. The implementation of this idea only applies for single inheritance scenarios because of the ambiguity in the case of multiple inheritance. The concept of visual inheritance for designing visual notations will be discussed in Section 3.

C. Argumentation Metamodel in SACM

The concept of an assurance case is that it provides a framework for analysing and communicating the assurance arguments and evidence about a particular system in a specific environment (e.g. covering safety requirements of the systems) [8]. In [26], an assurance case is defined as *a reasoned and compelling argument, supported by a body of evidence, that a* *system, service or organisation will operate as intended for a defined application in a defined environment*. These structured arguments, for example in complex systems, sometimes can be very large and complicated. Therefore, it is necessary for them to be clearly documented.

The Structured Assurance Case Metamodel (SACM) is a specification that defines a metamodel for representing structured assurance cases. It is published by The Object Management Group to improve standardisation and interoperability for assurance cases [8]. Part of the SACM specification defines a metamodel for representing structured arguments (see Fig. 3). In SACM, structured arguments are represented explicitly by the Claims, citation of artifacts or ArtifactReferences (e.g. Evidence and Context for claims), and the relationships between these elements [8]. The following relationships are defined:

- AssertedInference for relationship between Claims
- AssertedEvidence for ArtifactReference (as Evidence) and Claims
- AssertedContext for ArtifactReference (as Context) and Claims
- AssertedArtifactSupport for ArtifactReference (as Evidence) supporting ArtifactReference (as Evidence and Context)
- AssertedArtifactContext for ArtifactReference (as Context) supporting ArtifactReference (as Evidence and Context)

In addition to these core elements, it is also possible to provide:

- Description of the ArgumentReasoning associated with AssertedInference or AssertedEvidence
- Counter-Argument and Counter-Evidence (via isCounter: Boolean)
- Modular structured argument (via ArgumentPackage) including the mechanism to organise a specific selection of the ArgumentElements contained within the package (via ArgumentPackageInterface), and a mechanism to bound two or more argument packages (via ArgumentPackageBinding)
- An association of a number of ArgumentElements to a common group (via ArgumentGroup)

The current version of the SACM argumentation specification is not equipped with any visual notation. It is believed that the visual notation is an integral part of modern modelling languages and can help users in communicating their intended message in the form of a model (or models) to the reader [3]. It would therefore be beneficial if we provide the visual notation for the SACM argumentation metamodel, and we shall use this as an example application of our approach.

III. Visual Inheritance as Part of Notation Design

In this section, we describe the visual inheritance process as part for designing a visual notation based on a predefined metamodel. This approach emphasises the design coherence between parent and child-classes, where the basic design characteristics of a parent-class must be inherited to its child-classes. We hypothesise that by inheriting the base design of a parent-class to its child-classes, this could help the user of the notation inferring the semantic meaning of the classes and reduce the cognitive overload [3] in memorising the number of the notations.

As an input to start the visual inheritance process, we need a predefined metamodel as our main reference for designing the visual notation. This process can be applied only for single inheritance relationship scenarios, where a child-class in a metamodel only has exactly one parent-class. This condition is important to avoid ambiguity such as a "diamond problem" in programming multiple inheritance. The output of the visual inheritance process is the visual notation that consists of visual representations, the visual semantics, and the compositional rules based on the metamodel.

Before designing the visual notation, it is important to study all concepts of the metamodel to capture the core aspects and elements in the metamodel. We also suggest to observe the following elements in the metamodel:

- Type and attributes of each class.
  A class in a metamodel could be defined either as an abstract- or concrete-class. Unlike the concrete-class, an abstract-class cannot be instantiated. A certain class can have one or more attributes. An attribute is a feature within a class that describes a range of values that the instances of the class may hold. The attribute may be grouped by visibility. A visibility keyword can be given once for multiple features with the same visibility.
- Type of each relationship.
  A relationship is a semantic connection among model elements. For example, in UML there are several types of relationships such as association, generalisation, aggregation, and composition.

After studying the overall concept of the metamodel, we can begin the visual inheritance process. At first, we need to choose a root class. In selecting the root class, we suggest considering the scope of the concept that needs to be visually modelled. Note that we only create visual representation for the root class and its successors. We then need to see if the root class has any attribute that must be inherited from its predecessors. An attribute that exists in a class is always inherited to its successors.

Because we use visual inheritance approach, we need to start the design process from the root and inherit any base design to its child-classes by using depth first traversal. An abstract-class does not have a visual representation, but it still need to be analysed to see if we can define any design constraint from its semantic. In addition, an abstract class could have an attribute which must be inherited to its child-classes. A visual representation of attributes in an abstract-class might as well be defined based on the semantic of each attribute. Hence, in analysing an abstract-class we might get a design constraint or a base design for attributes to be inherited to its child-classes.

On the other hand, a concrete-class must have a visual representation. In creating a visual representation of a class, we need to consider a base design or a design constraint that

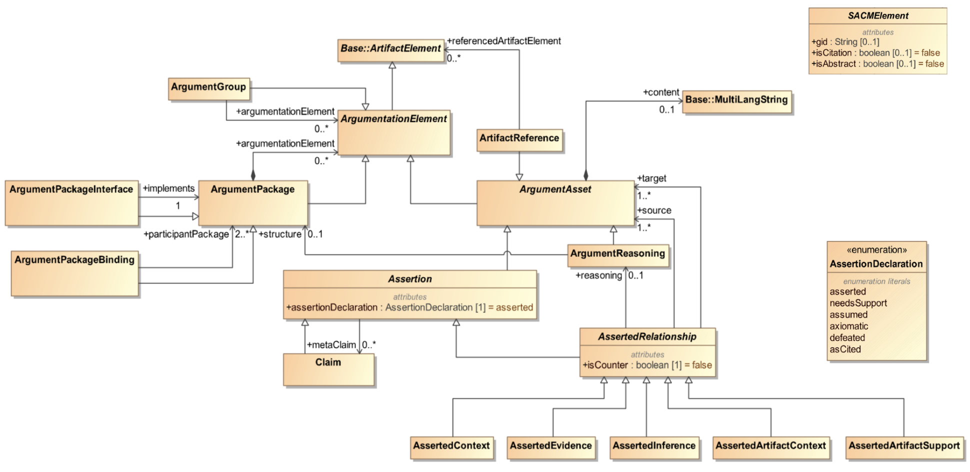

Fig. 3. UML Class Diagram of SACM Argumentation Metamodel [8]

is inherited from its parent-class (if any). Then in creating a visual representation of an attribute in a concrete-class, if there exists a base design that is inherited from a parent-class, we need to combine the base design with the base design of the concrete-class. If there is no base design of the attribute, then we need to create a base design from its semantic. The detail of the process can be seen in Fig. 4.

In designing the visual representations, we can adopt Bertin's visual variables that can be used to graphically encode information: shape, colour, size, texture, orientation, brightness, position (horizontal and vertical). These visual variables can be used to develop numbers of graphical symbols by combining the variables together in different ways [27]. Besides that, we suggest to apply the following design principles:

- *Semantic transparency*
  In creating the visual representation, we suggest creating the design of visual representation that conveys the meaning of the semantic definition of the identified class (Semantic Transparency) [3], [4]. If no visual representation directly conveys the meaning of the semantic definition, then create the visual representation design that can provide a clue to the (majority) of targeted users. If there is still no visual representation that can provide a clue to the majority of the targeted users, create the visual representation design that consensually (as a norm) provides a closer meaning to the domain of the language. Peirce's theory of signs [28] influences the idea of this guideline. Peirce stated that the form of a sign could be classified as one of three types:
  – An *Icon* has similar characteristics to the object that is being referred, e.g., a photograph as it certainly resembles whatever it depicts.
  – An *Index* shows factual connection or clue to its object, e.g., using an image of smoke to indicate a fire.
  – A *Symbol* provides interpretive habit or norm of reference to its object. Therefore, it needs to be learned, e.g., numbers. There is nothing inherent in the number 7 to indicate what it represents. So, it must be culturally learned.
- *Graphic Economy*
  It is also essential to keep the number of the visual representations as minimal as possible (Graphic Economy) [3]. This is to make sure not to overload the total number of the visual representations that could cause difficulty for the users in remembering all the notations [3]. We suggest to keep the visual representations simple by reusing the same visual representation for sub-classes that has close semantic meaning (e.g. relationship-types class); meanwhile, we can still distinguish their meaning while implementing them in the diagram by utilising the compositional rules of the visual representation which is can be defined based on the relationship between classes in the metamodel.

IV. Applicability

In this section, we present a case study to apply the visual inheritance. In this case, we adopt the SACM Argumentation Metamodel since currently there is no available visual notation for this metamodel. A metamodel of SACM Argumentation (see Fig. 3), that is documented in SACM manual [8], is used as an input to conduct the visual inheritance process. This argumentation metamodel does not have any multiple inheritance in it, so this approach is applicable for this metamodel.

To begin the visual inheritance process, we need to choose a root class. Here, we choose the *ArgumentAsset* as the root class because it contains the core concept of the structured argument. The *ArgumentAsset* is inherited public attributes from the *SACMElement* class (in the Structured Assurance Case Base Classes): *gid* (an element unique ID), *isCitation* (indicate whether an element cites another element), *isAbstract* (indicate an abstract element). The *ArgumentAsset* itself is an abstract-class, and there is no design constraint that we can define from its semantic. However, we still need to see the semantic of its attributes to see if we can have a base design for each attribute.

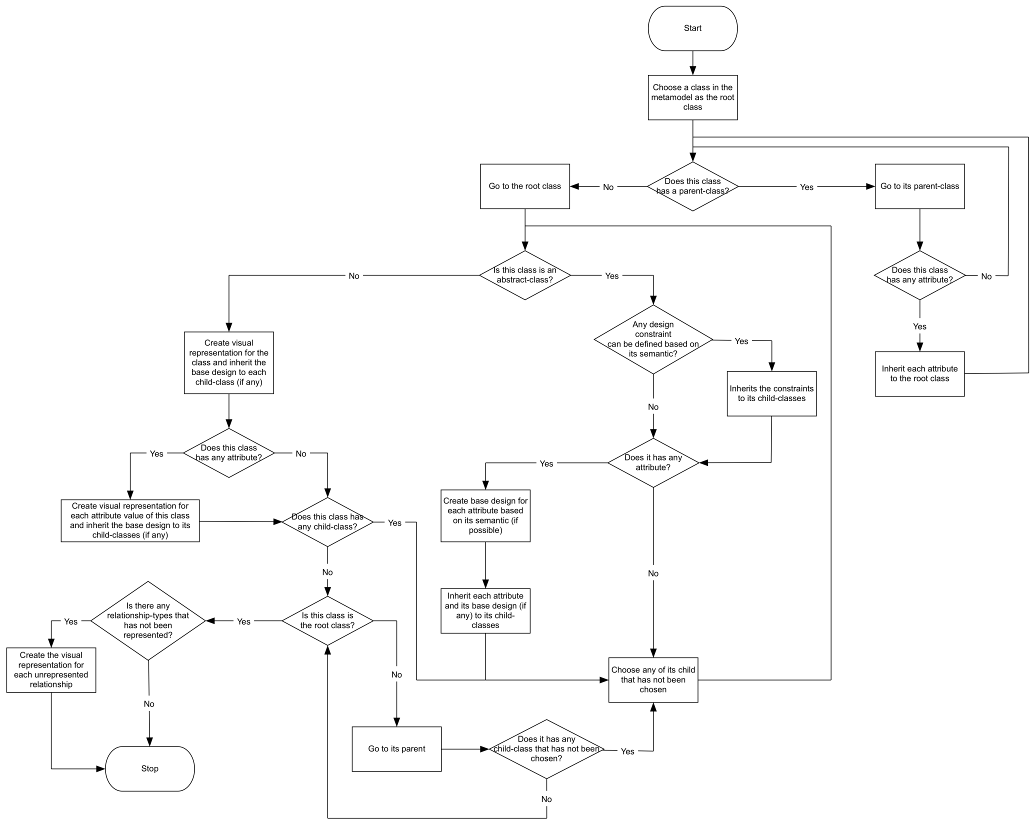
Start
Choose a class in the metamodel as the root class
Does this class has a parent-class?
No
Yes
Go to the root class
Go to its parent-class
Does this class has any attribute?
No
Yes
Inherit each attribute to the root class
Is this class is an abstract-class?
No
Yes
Any design constraint can be defined based on its semantic?
Yes
Inherits the constraints to its child-classes
No
Does it has any attribute?
Yes
Create base design for each attribute based on its semantic (if possible)
Inherit each attribute and its base design (if any) to its child-classes
No
Create visual representation for the class and inherit the base design to each child-class (if any)
Does this class has any attribute?
Yes
No
Create visual representation for each attribute value of this class and inherit the base design to its child-classes (if any)
Does this class has any child-class?
Yes
No
Is this class is the root class?
Yes
No
Is there any relationship-types that has not been represented?
Yes
No
Create the visual representation for each unrepresented relationship
Stop
Go to its parent
Does it has any child-class that has not been chosen?
Yes
No
Choose any of its child that has not been chosen

Fig. 4. The Visual Inheritance Process

Based on the semantic definition of *gid* attribute, it is a unique identifier for the SACM element. The type of the *gid* attribute is string, therefore we decided to represent this attribute as a string that indicate the ID of the element. Then for the attribute *+isAbstract: True*, because we could not find any visual representation that can convey directly the meaning of abstract, we decided to use an arbitrary design choice. In this case, we will consistently use a dashed line to represent an abstract meaning. Meanwhile, for *+isCitation*, we consistently use a closed square bracket to represent a citation, as is commonly used in academic writing.

The *ArgumentAsset* class has three child-classes: *Assertion*, *ArtifactReference*, and *ArgumentReasoning*. Hence, we inherit the attributes and their base design to the child-classes.

Assertion is an abstract-class, so we do not need to create a visual representation for this class. However, in addition to attributes that are inherited from its parent-class, this class has an attribute, *+assertionDeclaration* with six enumeration literals: *Asserted*, *Axiomatic*, *Defeated*, *Assumed*, *NeedsSupport*, and *AsCited*. From their semantics, we only can create a base design for *Defeated* and *AsCited*. For *defeated*, it is "indicating that the *Assertion* is defeated by counter-evidence and/or argumentation". Hence, we use a cross to convey this meaning. Then for *AsCited*, it is "indicating that because the Assertion is cited, the AssertionDeclaration should be transitively derived from the value of the AssertionDeclaration of the cited Assertion". Here, we use the same representation as *+isCitation* as both have related semantic meaning. All attributes in *Assertion* then must be inherited to all its child-classes: *Claim* and *AssertedRelationship*.

Claim is a concrete-class, so we need to create a visual representation for this class. In creating the visual representation we should consider its semantic definition so we can visually convey its meaning: "*Claims* are used to record the propositions of any structured argument contained in an *ArgumentPackage*. Propositions are instances of statements that could be true or false, but cannot be true and false simultaneously" [8]. Based on the semantic definition of a *Claim*, we could not find any visual representation that can directly infer the meaning of the semantic. Therefore, we adopt the visual representation that might provide a cue to the majority of targeted users that are already familiar with an existing assurance case notation. In this case, we reuse a visual representation of a *Goal* from a widely used assurance case notation (i.e. The Goal Structuring Notation (GSN) [26]), that has similar semantics with the *Claim*. Hence, we decided the visual representation for a *Claim* is a rectangle where the user can write down the propositional statement in it and the ID as its attribute (*gid*) in the top left. Fig. 5 illustrates the visual representation for a *Claim*. Then for the attribute *+isAbstract: True*, the *AbstractClaim* is visually represented as a dashed line rectangle. Meanwhile, for *+isCitation*, we place the rectangle

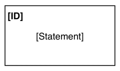

Fig. 5. The visual representation of a *Claim*

in a closed square bracket.

There is also an attribute (i.e. *assertionDeclaration*) of the *Assertion* class that is inherited to the *Claim* class. This attribute has six enumeration literals (*Asserted*, *Axiomatic*, *Defeated*, *Assumed*, *NeedsSupport*, and *AsCited*) that the visual representations of each attribute values need to be created as instances in the *Claim* class. In this case, the instances of the *Claim* class are: *Asserted Claim*, *Axiomatic Claim*, *Defeated Claim*, *Assumed Claim*, *NeedsSupport Claim*, and *AsCited Claim*. As mentioned in the visual inheritance approach guideline process in the previous section, the base design for these attribute values need to be inherited from the base design of the class, in this case is the *Claim* class. Then, we need to modify their base design to convey the semantic definition of these attribute values. As follows we describe the semantic definition of each attribute value (based on [8]) along with the proposed design of the visual representation and the rational behind the design.

- *asserted*, "the default enumeration literal, indicating that an Assertion is asserted". Based on this definition, the default *Claim* that being asserted is called as as *asserted Claim*, the visual representation of the *asserted Claim* is the visual representation of the *Claim*.
- *axiomatic*, "indicating the *Assertion* being made by the author is axiomatically true, so that no further argumentation is needed". Here, we use a thick line below the rectangle to indicate a sign of "no further argumentation is needed" (i.e. declaring axiomatically true).
- *defeated*, we already have a base design for this attribute, which is a cross. Hence, we add a cross on top of the rectangle to convey the meaning of *defeated Claim*.
- *assumed*, "indicating that the *Assertion* being made is declared by the author as being assumed to be true rather than being supported by further argumentation". Here, we add a gap in the bottom part of the rectangle to represent that there is no supporting evidence or argumentation.
- *needsSupport*, "indicating that further argumentation has yet to be provided to support the *Assertion*". For this, we add three dots in the bottom-centre part of the rectangle to represent that further evidence or argumentation is required.
- *asCited*, we already have a base design for this attribute, which is a closed square bracket. Therefore, we use the same representation as +*isCitation*.

Fig. 6 shows the summary of all *Claim* types.

After designing all *Claim* types, we continue to the other child-class of the *Assertion* class: the *AssertedRelationship* class. The semantic definition of this class is: "the abstract association class that enables the ArgumentAssets of any structured argument to be linked together" [8]. Based on this definition, this class is an abstract-class, so we do not need to create a visual representation for this class. Besides that, the semantic definition describes that this class is used to declare a type of association, which then became the design constraint to be inherited to its child-classes. There are several visual representations that can be used to represents an association e.g. colours and lines. We decided to adopt a 'line' to represents the *AssertedRelationship* since it can give information about the *source* and *target* of the *AssertedRelationship*. In this case, the types of the line can vary based on the semantic definition of the *AssertedRelationship* concrete child-classes.

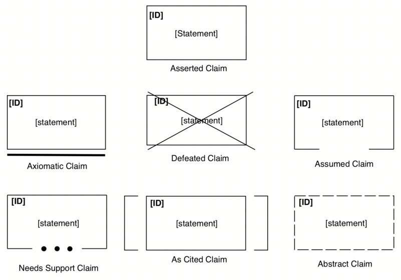

Fig. 6. Types of *Claim*

AssertedRelationship class has attributes that need to be inherited to its child-classes: *isCounter*, *assertionDeclaration*, *gid*, *isAbstract*, and *isCitation*. From all these attributes, only *isCounter* that has no base design yet. However, from its semantic, we can not define its base design without having a base design of the class, so we can not create it in an abstract-class. The *AssertedRelationship* class has five concrete child-classes for which visual representations need to be created including visual representations as instances of the inherited attributes. We proposed design, based on the semantic definition (on [8]) of all the *AssertedRelationship*'s concrete child-classes. In each case, we could not find any visual representation that intuitively conveys the semantic definition, so we used either visual representations from existing notations or proposed an arbitrary design. As follows we describe the result of the design process:

AssertedInference

- Definition:"records the inference that a user declares to exist between one or more Assertion (premise) and another Assertion (conclusion)".
- Visual representation: We use a solid arrowhead line to visualise an *AssertedInference*. This visual representation is influenced by a *SupportedBy* visual representation in GSN that indicating an inferential relationship. Then for the attributes, we already have a base design for each attribute except for *isCounter*, which has semantic definition "indicating that the AssertedRelationship counters its declared purposes". We use a hollow-head line to indicate the counter purpose of the *AssertedInference* and we use this consistently to represent *isCounter* attribute for all types of relationship. Then to create the visual representation for each attribute, we only need to combine the

defined base design of each attribute with the base design of *AssertedInference*. Note that the other child-classes of *AssertedRelationship* also have the same attributes as this class, so the process of creating the visual representation of the attributes is similar to this.

AssertedEvidence

- Definition: "records the declaration that one or more artifacts of Evidence (cited by *ArtifactReference*) provide information that helps establish the truth of a Claim".
- Visual representation: We adopted the visual representation from GSN, that is a solid arrowhead which is representing the evidential relationships. In this case, the visual representation is similar to the visual representation for *AssertedInference*, but we can distinguish them by identifying the source of the relationship (line). The source is an Evidence for *AssertedEvidence* (cited by *ArtifactReference*), and a *Claim* for *AssertedInference*.

AssertedContext

- Definition:"can be used to declare that the artifact cited by an ArtifactReference(s) provides the context for the interpretation and scoping of a Claim or ArgumentReasoning element".
- Visual representation: We cannot use the existing visual representation from GSN visual representation that conveys similar meaning (i.e. hollow arrowhead line for *inContextOf*) because we already used this type of visual representation. Therefore, we decided to create our own design for *AssertedContext*, with requirements: the design needs to be a line-type to represents a relationship type; and the type of the line-head can be designed as a solid or hollow. As the result, we use solid diamondhead line for *AssertedContext* and hollow diamondhead line for representing *Counter AssertedContext*.

AssertedArtifactSupport

- Definition: "records the assertion that one or more artifacts support another artifact".
- Visual representation: Here, we reuse the visual representation of *AssertedInference* since the main idea from its semantic meaning also supporting the targeted elements. We believed reusing the same visual representation to represents different classes that have a closer semantic meaning can help to minimise the number of the visual representations. To distinguish them, we identify the source and target elements. For *AssertedArtifactSupport*, the source and target elements must be of type ArtifactReference (as described in [8] for *AssertedArtifactSupport* constraints).

AssertedArtifactContext

- Definition: "records the assertion that one or more artifacts provide context for another artifact".
- Visual representation: Here, we reuse the visual representation of *AssertedContext* because the main idea from their semantic meaning are providing context for the targeted element. To distinguish them, we can identify the source and the targeted elements. For *AssertedArtifactContext* the source and target elements must be of type ArtifactReference (as described in [8] for *AssertedArtifactContext* constraints).

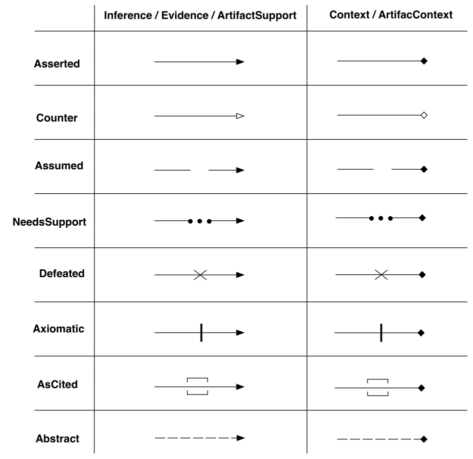

Fig. 7. Types of *Relationship*

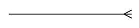

Fig. 8. The visual representation of a *MetaClaim*

As the result of the design combinations and as summary of all types of *AssertedRelationship* can be seen in Fig. 7.

We have finished designing the visual representations for classes that are successors of the root. Next, we see if all relationships have been represented. From all relationship-types only *metaClaim*, which is an association relationship between *Assertion* and *Claim*, has not been represented. The *+metaClaim* association indicates "references Claims concerning the Assertion". Therefore, we need to create a visual representation that can convey the semantic meaning of the *+metaClaim*, in this case as a type of relationship visual representation that the source of the relationship is a *Claim* and the targeted of is the *Assertion* types (i.e. the concrete child-classes of the *Assertion*) [8]. Based on the semantic definition, we decided to use a 'line' to represent the *+metaClaim.* Here, we use the "crow's foot line" since we do not have a constraint such as "counter" *+metaClaim* that need to be visualised in the form of a hollow head-line. The illustration of the *+metaClaim* visual representation can be seen in Fig. 8.

After designing the visual representation of the *Assertion* class, we need to design the visual representation of the other child-classes of the *ArgumentAsset* class: *ArtifactReference* and *ArgumentReasoning*. According to [8], "*ArtifactReference* enables the citation of an artifact as information that relates to the structured argument." This semantic definition inspired us to create the visual representation of *ArtifactReference* in a form of a file or document icon because we think it infers the semantic meaning. Besides that, for the *ArgumentReasoning*, we created a visual representation in a form of an annotation-type icon that can be attached to the instances of *AssertedRelationship* class. This is to convey the semantic meaning of the *ArgumentReasoning*: "can be used to provide additional description or explanation of the asserted relationship". Fig. 9

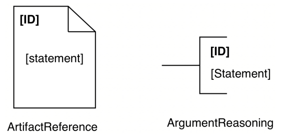

Fig. 9. The visual representations of an *ArtifactReference* and an *ArgumentReasoning*

shows the visual representations of the *ArtifactReference* and the *ArgumentReasoning*.

V. Limitations

There are some limitations to the current version of the proposed approach. *First*, in selecting the root in the metamodel, the scope of the concept is used as the only consideration. This aspect could be explored further to have more specific justification in selecting the root. *Second*, the proposed approach highly depends on the predefined metamodel, and the metamodel is depended on the method and style used by the meta-modeller. *Third*, currently, the proposed approach is being applied in a case study since it is part of work in progress. It is important to explore the applicability of the visual inheritance mechanism in another case study in order to evaluate the validity aspect.

VI. Conclusion and Future Work

We have outlined the visual inheritance as part of visual notation design (considered as an extension of the Semiotic Clarity principle of PoN [3] regarding a guideline about a complete monosemy [27] through the inheritance mechanism). The proposed approach emphasises the design coherence between parent- and child-classes to provide design efficiency (in terms of reusability and extensibility) as well as to help notation users to easily infer the semantic meaning of the visual representations that have a similar base design. We have also described the applicability of the proposed approach by designing the visual notations for SACM Argumentation metamodel. As the future work, we are planning to conduct empirical studies to test the effectiveness [3], [4] of the visual inheritance. In this case, we are interested in observing how effective is the idea of design coherence. We are also interested in testing the semantic transparency [3], [4] of the notation as a result of adopting the visual inheritance process.

References

[1] R. M. A. El-Ghafar, A. M. Ghareeb, and E. S. Nasr, "Designing user comprehensible requirements engineering visual notations: A systematic survey," in *Informatics and Systems (INFOS), 2014 9th International Conference on*. IEEE, 2014, pp. SW–10.

[2] A. El Kouhen, A. Gherbi, C. Dumoulin, and F. Khendek, "On the semantic transparency of visual notations: experiments with uml," in *International SDL Forum*. Springer, 2015, pp. 122–137.

[3] D. Moody, "The physics of notations: Toward a scientific basis for constructing visual notations in software engineering," *IEEE Transactions on Software Engineering*, vol. 35, no. 6, pp. 756–779, 2009.

[4] K. Arning and M. Ziefle, ""it's a bunch of shapes connected by lines": Evaluating the graphical notation system of business process modeling languages," in *Full paper at the 9th International Conference on Work With Computer Systems, WWCS*, 2009.

[5] N. Genon, P. Heymans, and D. Amyot, "Analysing the cognitive effectiveness of the bpmn 2.0 visual notation," in *International Conference on Software Language Engineering*. Springer, 2010, pp. 377–396.

[6] T. R. Green, "Cognitive dimensions of notations," *People and computers V*, pp. 443–460, 1989.

[7] G. Costagliola, A. Delucia, S. Orefice, and G. Polese, "A classification framework to support the design of visual languages," *Journal of Visual Languages & Computing*, vol. 13, no. 6, pp. 573–600, 2002.

[8] O. M. G. (OMG), *Structured assurance case metamodel (SACM)*, Version 2.0, 2018, https://www.omg.org/spec/SACM/About-SACM/.

[9] G. Popescu and A. Wegmann, "Using the physics of notations theory to evaluate the visual notation of seam," in *Business Informatics (CBI), 2014 IEEE 16th Conference on*, vol. 2. IEEE, 2014, pp. 166–173.

[10] D. van der Linden, I. Hadar, and A. Zamansky, "What practitioners really want: requirements for visual notations in conceptual modeling," *Software & Systems Modeling*, pp. 1–19, 2018.

[11] J. Krogstie, G. Sindre, and H. Jørgensen, "Process models representing knowledge for action: a revised quality framework," *European Journal of Information Systems*, vol. 15, no. 1, pp. 91–102, 2006.

[12] R. Schuette and T. Rotthowe, "The guidelines of modeling–an approach to enhance the quality in information models," in *International Conference on Conceptual Modeling*. Springer, 1998, pp. 240–254.

[13] D. Moody and J. van Hillegersberg, "Evaluating the visual syntax of uml: An analysis of the cognitive effectiveness of the uml family of diagrams," in *International Conference on Software Language Engineering*. Springer, 2008, pp. 16–34.

[14] D. L. Moody, P. Heymans, and R. Matulevičius, "Visual syntax does matter: improving the cognitive effectiveness of the i* visual notation," *Requirements Engineering*, vol. 15, no. 2, pp. 141–175, 2010.

[15] N. Genon, D. Amyot, and P. Heymans, "Analysing the cognitive effectiveness of the ucm visual notation," in *International Workshop on System Analysis and Modeling*. Springer, 2010, pp. 221–240.

[16] P. Mäder and J. Cleland-Huang, "A visual traceability modeling language," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2010, pp. 226–240.

[17] A. Algablan, "A visual notation and an improvement for the syntax of larman's operation contracts," Ph.D. dissertation, University of Ottawa, 2016.

[18] M. d. G. da Silva Teixeira, G. K. Quirino, F. Gailly, R. de Almeida Falbo, G. Guizzardi, and M. P. Barcellos, "Pon-s: a systematic approach for applying the physics of notation (pon)," in *Enterprise, Business-Process and Information Systems Modeling*. Springer, 2016, pp. 432–447.

[19] H. Störrle and A. Fish, "Towards an operationalization of the "physics of notations" for the analysis of visual languages," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2013, pp. 104–120.

[20] D. van der Linden, A. Zamansky, and I. Hadar, "A framework for improving the verifiability of visual notation design grounded in the physics of notations," in *Requirements Engineering Conference (RE), 2017 IEEE 25th International*. IEEE, 2017, pp. 41–50.

[21] T. R. Green, A. E. Blandford, L. Church, C. R. Roast, and S. Clarke, "Cognitive dimensions: Achievements, new directions, and open questions," *Journal of Visual Languages & Computing*, vol. 17, no. 4, pp. 328–365, 2006.

[22] R. Ducournau and J. Privat, "Metamodeling semantics of multiple inheritance," *Sci. Comput. Program.*, vol. 76, no. 7, pp. 555–586, 2011.

[23] G. B. Singh, "Single versus multiple inheritance in object oriented programming," *ACM SIGPLAN OOPS Messenger*, vol. 6, no. 1, pp. 30–39, 1995.

[24] R. A. Cain, J. A. De Lu, and R. E. Lemke, "Development system with methods for visual inheritance and improved object reusability," Jul. 22 1997, uS Patent 5,651,108.

[25] C. P. Jazdzewski, "Development system with methods providing visual form inheritance," Dec. 14 1999, uS Patent 6,002,867.

[26] GSN, *Goal Structuring Notation (GSN) Community Standard*, Version 1, 2011, http://www.goalstructuringnotation.info/documents/GSN_Standard.pdf.

[27] B. Jacques, "Semiology of graphics: diagrams, networks, maps," *University of Wisconsin Press, Madison, Wisconsin*, 1983.

[28] A. Atkin, "Peirce's theory of signs," in *The Stanford Encyclopedia of Philosophy*, summer 2013 ed., E. N. Zalta, Ed. Metaphysics Research Lab, Stanford University, 2013.