DISV: Domain Independent Semantic Validation of Data Files

Ashish Kumar[1], Bill Harris[2], and Gang Tan[1]
[1]Penn State University, State College, PA 16802, USA
[2]Galois, Inc., Portland, OR-97204, USA

Data format specification languages such as PDF or HTML have been used extensively for exchanging structured data over the internet. While receivers of data files (e.g., PDF viewers or web browsers) perform syntax validation of files, validating deep semantic properties has not been systematically explored in practice. However, data files that violate semantic properties may cause unintended effects on receivers, such as causing them to crash or creating security breaches, as recent attacks showed. We present our tool DISV (Domain Independent Semantic Validator). It includes a declarative specification language for users to specify semantic properties of a data format. It also includes a validator that takes a data file together with a property specification and checks if the file follows the specification. We demonstrate a rich variety of properties that can be verified by our tool using eight case studies over three data formats. We also demonstrate that our tool can be used to detect advanced attacks on PDF documents.

***Index Terms*—Semantic Property Validation, Attribute Grammars, PDF Attacks.**

I. INTRODUCTION

Data format specification languages such as PDF or HTML have long been used for exchanging structured data between data producers and receivers. While data receivers typically perform syntactic validation on data files according to format specifications, semantic properties on data files are seldom validated even though many of these properties are also described in format specifications and data files violating those properties can cause unintended consequences.

For example, in an SVG (Scalable Vector Graphics) file, there are a set of objects with the `use` tag and a set of objects with the `symbol` tag. Through an `href` link, an object with the `use` tag can refer to a second object with either the `use` or the `symbol` tag. With the link, the attributes of the second object are inlined into the first object, when it is rendered. If we define a graph in which nodes are objects with `use` or `symbol` tags and links are references between objects using `href` links, one semantic property from the `SVG` specification [1] requires that the graph must be acyclic. This property is called *no use-use circularity*. If this property does not hold for an `SVG` file, it may cause an `SVG` data receiver to crash or enter an infinite recursion. Unfortunately, this property is not verified by an `SVG` data receiver. Examples of such semantic properties abound in data formats including `PDF` and `HTML`; Section VI presents a set of case studies. In the recent past, the lack of validation of such semantic properties led to attacks on `PDF` documents [2], [3], [4], [5].

Compared to syntactic validation, validating semantic properties is more challenging, as semantic properties are often global properties. For instance, validating the property of no `use-use` circularity requires going through all objects in an `SVG` file and checking the global property of no cycles. A general framework for validating such semantic properties must include an expressive language for declaring these properties and a tool for automatically validating an input data file according to a specification.

We present DISV (Domain Independent Semantic Validator) for validating semantic properties on data files. DISV can validate semantic properties of any data format, with the assumption that a data file of the format encodes a set of objects. Each object is assumed to have a set of fields, some of which are links to other objects. There can be different kinds of links. For example, in an `HTML` file, an object represents information in an `HTML` tag and objects can refer to other objects using `href` fields. This kind of references are semantic links across objects in an `HTML` file. It can also have another kind of links, namely syntactic links between objects; i.e. if the tag represented by object A is nested within the tag represented by object B, then A is linked to B by a parent-child link.

With the aforementioned assumption on data formats, DISV is designed to have two major components. The first component is a declarative language for specifying semantic properties. A specification file in this language is comprised of (1) a *graph specification*, which specifies what objects and links from the input file constitute a graph on which semantic properties should be checked, (2) a set of *semantic definitions* that associate attributes to graph nodes or introduce auxiliary concepts, and (3) a set of *semantic properties* for validation. The second component of DISV takes an input data file and a property specification file, and performs automated property validation on the input data file.

We have performed experimental evaluation of DISV using eight semantic properties across three different file formats (namely `PDF`, `HTML` and `SVG`). We evaluated DISV's effectiveness and also its performance for each of these semantic properties. From our evaluation, we conclude that the execution time of DISV is majorly dependent on the size of the graph specification, and a more precise graph specification leads to a much faster execution time.

II. RELATED WORK

There have been a few previous efforts at semantic property validation of the `PDF` format. Li et al. [6] developed a language-theoretic approach to `PDF` parsing and semantic

property validation using ACL2 by defining a new language to define PDF documents. Similar approaches to identifying sementic bugs have been proposed before [7]. Wyatt [8] derived a machine-readable definition of formally defined PDF objects and data integrity relationships called the Arlington PDF Model. There have been several attempts to derive machine-readable definitions of the PDF format in the past. Adobe Dictionary Validation Agent is a machine-readable definition that covers the PDF object model and content streams and itself uses custom PDF extensions [9]. Other attempts to derive machine-readable definitions of the PDF format include the veraPDF model [10], [11] and Levigo's DSL framework [12]. However all the above approaches are specific to PDF documents, and allow for a restricted set of properties to be specified. DISV is domain-independent and validates first order properties in the logic of graphs.

Other formats like HTML5 have constraint validation *forms* included in their receivers [13]. However, such forms can validate only a restricted set of semantic properties, such as restricting the range of values allowed by certain input fields. There have been quite a few tools designed for semantic property validation in IoT [14], [15], [16]. Kolozali et al. described an ontology validation tool designed for W3C SSN [17]. It can be used for validating ontologies in IoT and collect statistics regarding the most commonly used terms in an ontology; however, it cannot be used to check universal or existential quantified properties, as DISV can.

In the past, new frameworks for specifying and validating graph-based properties have also been designed. For example, Navarro et al. [18] defined a logic for defining properties of graphs called *Graph Navigational Logic* (GNL), and gave an algorithm that validates GNL properties, using a tableau construction. However GNL is quite restrictive in the properties that it can specify - a property in GNL for the edge-labelled graph G has the format (upto type of quantifier) $\forall H \subseteq G \Rightarrow \exists H',\ H' \subseteq G$ where H and H' are specific edge-labelled graphs specified by the property. GNL doesn't allow us to reason about values assigned to nodes or treat nodes as a collection of values like in an attribute grammar or in first order logics of graphs.

Our attribute-based semantic definitions are inspired by attribute grammars [19], [20], [21]. Attribute grammars have been demonstrated for validating a wide variety of properties of context free grammars including type checking, type inference, code generation, optimizing database query operations, etc [22], [23]. Silver [24] is an attribute grammar specification system designed for validating highly modular attribute grammars. When compared to DISV, Silver can be used to specify the attribute-based properties but not the properties based on the first order logics of graphs.

We demonstrate another use of DISV—to check for attacks on PDF Documents. We use DISV to validate whether a PDF document can suffer from a faulty permissions attack [5]. In the past, several attacks on PDF documents have been proposed [2], [3], [4], [5], and tools for checking the same have been developed.

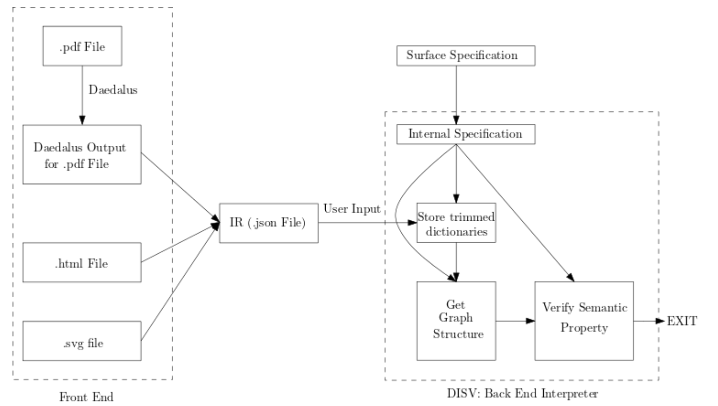


Fig. 1: Internal Architecture of DISV.

III. System Architecture of DISV

The internal architecture of DISV is shown in Fig. 1. First, the front-end of DISV transforms an input file into an Intermediate Representation (IR), represented as a JSON file. We have built the front ends for a number of file formats, including HTML and SVG. For PDF, a PDF parser generated from a definition in DaeDaLus (a format definition language that can express data-dependent grammars) and convert parse results to our IR. More front ends can be incorporated for other data formats.

Recall that we assume the input data file encodes a set of objects. In our JSON IR file, an object is stored as a dictionary (i.e., a key-value store) that maps from object fields to field values. Each object is also given a unique ID. The field value of an object can be the ID of another object, which is how links between objects are encoded.

DISV takes as input the IR (JSON file) and a semantic property specification as a human-understandable, external specification. It first converts the external specification to an internal form, which it then uses along with the IR to store in memory all objects. As an optimization, DISV only stores those objects whose keys occur in the internal specification. Using this list of objects and the internal specification, DISV constructs a graph that represents the given document's relevant components. It then validates the graph regarding the properties declared in the input property specification.

IV. Specification Language

In this section, we define DISV's specification language. We start with an example property that will be used to illustrate the main features of the specification language.

A. An illustrative example

We illustrate our framework by example on the *page-tree inheritance* property of PDF documents. Further background information on the PDF format is given in Appendix A. A PDF file may contain a set of objects that, together, form a so-called *page tree* [25]; an example page tree is depicted in Fig. 2. The page tree's leaves are Page objects; its internal nodes are Pages objects; each Pages node stores common metadata for all leaf Page objects of the subtree rooted with the node.

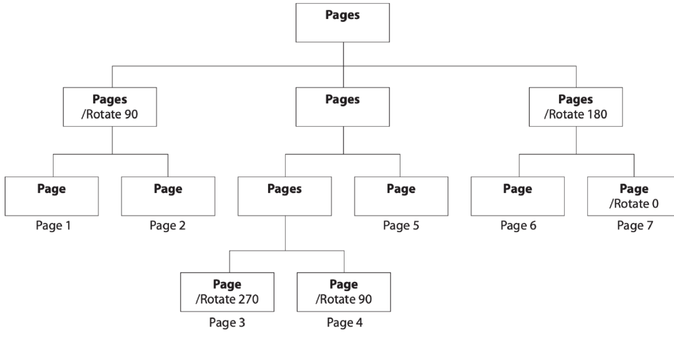


Fig. 2: An example page tree.

The page tree's edges are represented by object references (called *indirect objects*) that are stored in internal nodes; its root is identified in a distinguished `Catalog` object. We use term *page-tree objects* to refer to both `Page` and `Pages` objects.

A page-tree object has a set of key-value pairs, and keys may be designated in the PDF specification as possibly "Required" and/or "Inheritable". A key designated as "Required" but *not* "Inheritable" for an object must have a value in that page-tree object. A key marked as "Required" *and* "Inheritable" must either be defined by the page-tree object or one of its ancestors in the page tree. By the PDF specification, two keys, `Resources` and `MediaBox`, are designated as both "Required" and "Inheritable". These conditions on objects and keys, combined, constitute the page-tree inheritance property.

B. Specification Grammar

We now explain our specification language by presenting the general constructs and illustrating how they are used to formalize the page-tree inheritance property in DISV (Fig. 3). The top-level specification grammar for DISV is defined below. It first requires the specification of a graph structure followed by a specification of a set of semantic definitions and properties.

$$\langle \text{Spec_Grammar} \rangle \quad ::= \quad \langle \text{Graph_Spec} \rangle \langle \text{Semantic_Definition_Spec} \rangle \langle \text{Semantic_Property_Spec} \rangle$$

Graph Specification A graph specification in DISV specifies a graph structure based on the set of objects and their fields in the input IR file. Note that the specified structure may contain only a subset of all objects and fields in the input file. Further, DISV allows specialized graph structures, including trees, ordered trees, and forests.

$$\langle \text{Graph_Spec} \rangle \quad ::= \quad \langle \text{Tree} \rangle \mid \langle \text{Ordered Tree} \rangle \mid \langle \text{Forest} \rangle \mid \langle \text{Graph} \rangle$$

The specification of a tree is comprised of the keyword "Tree" followed by a specification of the root node (using nonterminal 'Source_Def' below), and a specification of how to identify children from nodes (using nonterminal 'Recursive_Child_Def'). In the grammar given below, 'Source_Def'

```
1  Tree:
2  Root is unique d satisfying d.<"Id">
3  = d1.<"Pages"> where unique d1 satisfies
4  d1.<"Type"> = "Catalog".
5
6  forall d in Tree, d0 is Child(d) where
7  d0.<"Id"> in d.<"Kids"> and d0.<"Type">
8  in ["Page", "Pages"].
9
10 Semantic Definitions:
11 forall d in Root, d.<is_Resources_defined>
12 = iskeydefined(d, "Resources").
13
14 forall d in Root, d.<is_MediaBox_defined> =
15 iskeydefined(d, "MediaBox").
16
17 forall d1 in Tree, forall d2 in Child(d1),
18 d2.<is_Resources_defined> =
19 d1.<is_Resources_defined> or
20 iskeydefined(d2, "Resources").
21
22 forall d1 in Tree, forall d2 in Child(d1),
23 d2.<is_MediaBox_defined> =
24 d1.<is_MediaBox_defined> or
25 iskeydefined(d2, "MediaBox").
26
27 Semantic Property:
28 forall d in Leaves, d.<is_Resources_defined>
29 = True.
30
31 forall d in Leaves, d.<is_MediaBox_defined>
32 = True.
```

Fig. 3: The page-tree inheritance property, expressed in DISV.

and 'Recursive_Child_Def' are nonterminals parameterized with strings "Root" and "Tree" respectively.

$$\langle \text{Tree} \rangle \quad ::= \quad \texttt{Tree} : \langle \text{Source_Def (Root)} \rangle \langle \text{Recursive_Child_Def (Tree)} \rangle$$

The grammar for 'Source_Def' is parameterized by a variable x, which is 'Root' or 'Source' depending on whether the graph is a tree or not. It defines a set of objects using a proposition. Since our IR represents objects by dictionaries, the proposition applies to the key-value pairs in dictionaries. The list of propositions that DISV allows is defined in Appendix C. The keyword "unique" is used if a single object is expected to satisfy the required property.

$$\begin{aligned}\langle \text{Source_Def}(x) \rangle \quad ::= \quad & x \ \texttt{is} \ (\texttt{unique})? \ \langle d_0 \rangle \ \texttt{satisfying} \\ & \langle \text{getGraph_Prop} \ (\ \langle \text{d}_\text{i} \rangle_{i=0}^{k} \ ) \rangle \\ & \texttt{where} \ (\texttt{unique})? \ \langle d_1 \rangle \ \texttt{satisfies} \\ & \langle \text{getGraph_Prop} \ (\ \langle \text{d}_\text{i} \rangle_{i=1}^{k} \ ) \rangle \\ & \ldots \\ & \texttt{where} \ (\texttt{unique})? \ \langle d_k \rangle \ \texttt{satisfies} \\ & \langle \text{getGraph_Prop} \rangle (\langle d_k \rangle).\end{aligned}$$

As an examples, lines 2–3 in Fig 3 specifies the root of the page tree. It says that the root is a unique object whose 'Id' equals the value of the 'Page' entry in the unique Catalog object.

As mentioned earlier, nonterminal 'Recursive_Child_Def' is for specifying the children of nodes in the graph structure. It is parameterized by a variable x, which is either 'Tree', 'Forest', 'Graph', or 'Ordered Tree'. For each object d in x,

it recursively defines the child objects of d using a predicate on both the parent and child objects.

⟨Recursive_Child_Def(x)⟩ ::= `forall` ⟨d⟩ `in` x, ⟨d_1⟩ `is Child`(⟨d⟩) `where` ⟨getGraph_Prop⟩(⟨d⟩,⟨d_1⟩).

Lines 5–6 in Fig 3 defines child objects in a page tree. It says that in the tree being defined, d0 is a child of d if the 'Id' value of d0 is present in the 'Kids' array of d and d0's 'Type' is either 'Page' or 'Pages'.

The specification for an ordered tree is similar to that of a tree, with the difference that the recursive child property also assigns a number (order) to the children of an object. Nonterminal 'getGraph_Index' is defined in Appendix C.

⟨Ordered Tree⟩ ::= *Ordered Tree:* ⟨Source_Def⟩*(Root)* ⟨Ordered_Recursive_Child_Def⟩

⟨Ordered_Recursive_Child_Def⟩ ::= *forall* ⟨d⟩ *in Ordered Tree,* ⟨d_1⟩ *is ith Child*(⟨d⟩) *where*⟨getGraph_Prop⟩(⟨d⟩,⟨d_1⟩). *where i is* ⟨getGraph_Index⟩(⟨d⟩).

The specification of a forest and a general graph is given below. They are similar in that they both allow one or more source and recursive child definitions, with the difference being that DISV checks that a forest is acyclic, whereas cycles are allowed in general graphs.

⟨Forest⟩ ::= `Forest` :⟨Source_Def (Root)⟩$^+$ ⟨Recursive_Child_Def (Forest)⟩$^+$

⟨Graph⟩ ::= `Graph` :⟨Source_Def (Source)⟩$^+$ ⟨Recursive_Child_Def (Graph)⟩$^+$

Semantic Definition Specification Semantic definitions in DISV introduce additional concepts on top of the graph structure and are used in defining semantic properties. The semantic definition section starts with the string "Semantic Definitions:", followed by one or more semantic definitions.

⟨Semantic_Definition_Spec⟩ ::= `Semantic Definitions` : ⟨Semantic_Def⟩$^+$

A semantic definition can be either an *attribute* or a *quantifier-set* definition:

⟨Semantic_Def⟩ ::= (⟨ Attribute_Def⟩ | ⟨Quantifier_Set_Defs⟩)$^+$

The notion of attributes is inspired by attribute grammars [19], [20]. In an attribute grammar, attributes are associated with nonterminals of the grammar; how attributes are computed is also specified in the grammar. Depending on the order of attribution computation, there are different kinds of attributes. *Synthesized attributes* are computed in a bottom-up fashion: the value of a synthesized attribute of a parent node in a parse tree is computed based on the attribute values of the child nodes. *Inherited attributes* are computed in a top-down fashion: the value of an inherited attribute of a child node is computed based on the attribute values of the parent node.

The grammar for defining an attribute in DISV is given below. An attribute 'attr_name' for object d_k is defined as a function of the key-value pairs of objects d_1 to d_{k-1}. These objects belong to a set of objects, which is defined using the (variadic) nonterminal Attr_Set (that may or may not be parameterized by pre-defined objects). The nonterminals Attr_Set, attr_name and Attr_fun are defined in Appendix C.

⟨Attr_Def⟩ ::= `forall` ⟨d_1⟩ `in` ⟨Attr_Set⟩ , . . . , `forall` ⟨d_k⟩ `in` ⟨Attr_Set(⟨d_1⟩, ... ⟨d_{k-1}⟩)⟩, ⟨d_k⟩.⟨attr_name⟩=⟨Attr_fun⟩(⟨d_1⟩, ⟨d_2⟩, ... ⟨d_{k-1}⟩).

An example attribute definition is given in lines 9–19 of Fig 3. They introduce two boolean attributes for each page-tree object: 'is_Resources_defined' and 'is_MediaBox_defined'. Attribute 'is _Resources _defined' is true for an object if key 'Resources' is defined either directly in the object or in one of its ancestors; attribute 'is_MediaBox_defined' works similarly. Note that DISV allows defining attributes for only acyclic graphs (trees and forests). Also, DISV allows the specification of both inherited and synthesized attributes, as the quantifier set (Attr_Set) for specifying attributes can be either the set of all children, the parent, or even all of the nodes in the tree as discussed in Appendix C.

The second kind of semantic definitions is called *quantifier sets*. A quantifier-set introduces a set of nodes or edges from the graph. Once defined, a quantifier set can be quantified over when defining semantic properties.

⟨Quantifier_Set_Defs⟩ ::= ⟨Quantifier_Node_Set⟩ | ⟨Quantifier_Edge_Set⟩

The specification for a set of nodes is given below. The set of nodes (objects) is defined by a Boolean proposition over the key-value entries of an object. The nonterminal 'Graph_Prop' is defined in Appendix C.

⟨Quantifier_Node_Set⟩ ::= ⟨Node_name⟩ `is` ⟨d⟩ `satisfying` ⟨*Graph_Prop*⟩(⟨d⟩).

An example of a quantifier set definition over a set of nodes is given at line 8 of Fig 4, copied here: `Head is d satisfying d.<"Name"> = "head"`. It defines a quantifier set named 'Head', which contains a set of nodes whose 'Name' value is 'head'. With the definition of this quantifier set, a later semantic property quantifies over all nodes in 'Head' at lines 14–15 of Fig 4.

The specification for a set of edges is given below. An edge is written as a pair of objects, and each object is defined by a Boolean proposition over the key-value entries of the object. Nonterminal 'Graph_Edge_Set' is defined in Appendix C.

⟨Quantifier_Edge_Set⟩ ::= ⟨Edge_name⟩ `is` (⟨d_1⟩,⟨d_2⟩) `satisfying` ⟨*Graph_Prop*⟩(⟨d_1⟩,⟨d_2⟩) `where` ⟨d_1⟩ `in` ⟨*Graph_Edge_Set*⟩ `where` ⟨d_2⟩ `in` ⟨*Graph_Edge_Set*⟩.

An example of a quantifier set definition over a set of edges is given at lines 10-11 of Fig 4, copied below:

```
Ref is (d1, d2) satisfying d1.<"href"> =
REFSTRING(d2.<"id">) where d1 in [ Tree ]
where d2 in [ Tree ] .
```

It defines a set of edges labelled 'Ref' where (d1,d2) is an edge in 'Ref' if d1's 'href' value equals "#" plus d2's 'id' value.

Semantic Property Specification It comprises of the string "Semantic Property:" followed by one or more semantic properties.

$\langle$Semantic_Property_Spec$\rangle$ ::= *Semantic Property:* $\langle$Semantic_Property$\rangle^+$

The grammar for a semantic property is given below. It starts with zero or more quantifiers over nodes or edges, followed by a predicate. The symbol o below is used to denote either a node or an edge (represented by a pair of nodes).

$\langle$Semantic_Property$\rangle$ ::= $\langle$Quantifier$\rangle(o, [])\ldots$
$\langle$Quantifier$\rangle(o_k, [o, o_1, \ldots o_k-1])$
$\langle$Sem_Prop_Prop$\rangle$($\langle d_1\rangle$, $\langle d_2\rangle$, ... $\langle d_k\rangle$).

The grammar for quantifiers is given below. A quantifier can quantify over either nodes or edges. Also the quantifier set for the i-th quantifier can use quantified objects defined by the previous quantifiers.

$\langle$Quantifier(o ,[$\langle d_1\rangle$, $\langle d_2\rangle$, ... $\langle d_k\rangle$])$\rangle$ ::= `forall` $\langle d\rangle$ `in`
$\langle$Sem_Prop_Set($\langle d_1\rangle$, $\langle d_2\rangle$, ... $\langle d_k\rangle$)$\rangle$,
| `forall` $(\langle d\rangle, \langle d'\rangle)$ `in`
$\langle$Sem_Prop_Set($\langle d_1\rangle$, $\langle d_2\rangle$, ... $\langle d_k\rangle$)$\rangle$,

An example of the page-tree inheritance property is given in lines 22–24 of Fig 3. It checks if attributes 'is_Resources_Defined' and 'is_MedixBox_Defined' hold for all leaf nodes in the page tree.

Note that the specification logic of DISV is mostly the first order logic of graphs and it allows for both universal and existential quantification. However, it additionally allows for quantifying over a 'PATH(src, set_edges)' predicate, which represents the set of nodes reachable from 'src' using edges from 'set_edges'. This allows us to specify graph properties such as connectivity, which ordinarily are not specifiable by a first order graph logic.

V. Property Validation

The property validation algorithm of DISV is given in Algorithm 1. It takes as input the property specification file and the data file (IR), and outputs a detailed log information about whether the properties in the specification are satisfied; if a property is not satisfied, it also outputs where it is violated in the input data.

The validation first finds all keys used in the user specification with GET_KEYS. It then stores all objects in the data file using READ_OBJECTS. It only stores those key-value pairs in a dictionary whose keys were found by the earlier GET_KEYS step. DISV then finds the graph type (i.e. tree, graph, forest or ordered tree) and stores this information separately. It then constructs the graph using GET_GRAPH and also validates the graph type. DISV then repeatedly reads and computes semantic definitions, and then reads and verifies semantic properties. We next describe the major components of validation in more detail.

GET_GRAPH. This function takes the user specification and the graph type, computes the graph, verifies if its structure matches the graph type, and outputs a detailed log of information pertaining to the graph structure. When computing the graph, it first finds a set of source nodes using the specification. Then for each node in the graph, the algorithm goes through all objects to find the node's children, according to the specification. This component of DISV works in O(nD) time, where D is the total number of objects in the input file and n is the number of nodes in the graph defined in the user specification.

Algorithm 1: DISV ALGORITHM

```
Input: User_spec_file.txt , Data_file.txt
Output: Output.log
1  Keys ← GET_KEYS(User_spec_file.txt)
2  objects ← READ_OBJECTS(Data_file.txt, Keys)
3  Graph_Type ← GET_GRAPH_TYPE(User_spec_file.txt)
4  Graph ← GET_GRAPH(User_spec_file.txt, Graph_Type)
5  READ( User_spec_file.txt, "Semantic Definitions:")
6  while ! READ(User_spec_file.txt, "Semantic Property:") do
7      Semantic_Def
         ←READ_DEFINITION(User_spec_file.txt)
8      if IS_ATTRIBUTE(Semantic_Def) then
9          COMPUTE_ATTRIBUTE(Graph, Semantic_Def, Graph_Type)
10     else
11         COMPUTE_QUANTIFIER_SET(Graph, Semantic_Def)
12 while ! EOF(User_spec_file.txt) do
13     Semantic_Prop ←
         READ_DEFINITION(User_spec_file.txt)
14     VERIFY_PROPERTY(Semantic_Prop, ϕ)
```

COMPUTE_ATTRIBUTE. This function computes the attributes specified by users. DISV allows for defining both inherited and synthesized attributes, and we associate attributes with the nodes of the graph computed early. Our validator assumes that attributes defined by users are well defined (a similar requirement is there in attribute grammars), which means that, if we construct a dependence graph between attributes, the graph has a topological order. This implies that attribute values can be computed using previously defined attributes, following the topological order. Assuming there is a O(n) number of attributes, where n is the number of nodes in the graph, this component of DISV works in O(n) time, since it follows the topological order to compute attribute values.

COMPUTE_QUANTIFIER_SET. This function computes the required quantifier sets of nodes or edges. It goes through each object in the graph against the property stated to find the quantifier set. A quantifier set may use previously defined quantifier sets - thus the order of specifying quantifier sets by user must be topologically sorted w.r.t to their dependencies. The specification for quantifier nodes sets allows universal quantifiers using the 'where' keyword, wheras the specification for quantifier edge sets is essentially a double universal quantification over nodes. This component of DISV works in O($n^q + n^2$) time, where n is the number of nodes in the graph and q is the node quantifier depth, as for each quantifier we make a single pass over the quantifier set, which has size O(n).

VERIFY_PROPERTY. This function validates a user defined semantic property and outputs the result of validation. We

describe the algorithm for validating semantic properties in Algorithm 2. DISV uses a map between object names to values called 'Quantifier_Map' to track the value of variables in a quantified property. DISV first checks if a property is a zeroth order property (i.e. it has no quantifiers), and if so it computes its (boolean) truth value by calling the VERIFY_PROPOSITIONAL_PROPERTY subroutine. If not, it extracts the top-level quantifier, computes its corresponding quantifier set and type and for each node or edge in the quantifier set, it recursively calls VERIFY_PROPERTY. DISV stores the truth value for the computation in the variable 'Truth_Value'. The quantifier map is restored to its earlier value at the end of each loop iteration, as it is passed by value during recursive calls.

Algorithm 2: VERIFY_PROPERTY ALGORITHM

```
Input: Semantic_Prop, Quantifier_Map
Output: Truth_Value
1  if IS_ZEROTH_ORDER_PROPERTY(Semantic_Prop) then
2      Truth_Value ←
3        VERIFY_PROPOSITIONAL_PROPERTY(Semantic_Prop,
       Quantifier_Map)
4      return Truth_Value
5  (Quantifier, Remaining_Property)
6      ← EXTRACT_TOP_QUANTIFIER (Semantic_Prop)
7  Quantifier_Name←
     GET_QUANTIFIER_NAME(Quantifier)
8  Quantifier_Set ← GET_QUANTIFIER_SET (Quantifier)
9  Quantifier_Type ← GET_QUANTIFIER_TYPE
     (Quantifier)
10 if IS_UNIVERSAL_QUANTIFIER (Quantifier_Type) then
11     Truth_Value ← True; ⟨operator⟩ ← ∧
12 else
13     Truth_Value ← False; ⟨operator⟩ ← ∨
14 if IS_NODE_SET(Quantifier_Set) then
15     for n ∈ Quantifier_Set do
16         Truth_Value ← Truth_Value ⟨operator⟩
17             VERIFY_PROPERTY (Remaining_Property,
18                 Quantifier_Map ∪ [(Quantifier_Name, n)])
19 else
20     for (n1,n2) ∈ Quantifier_Set do
21         Truth_Value ← Truth_Value ⟨operator⟩
22             VERIFY_PROPERTY (Remaining_Property,
23                 Quantifier_Map ∪ [(Quantifier_Name.first,
             n1); (Quantifier_Name.second, n2)])
24 return Truth_Value
```

This component's time complexity in O($n^q + e^q$), where n is the number of nodes, e the number of edges in the graph, and q is the quantifier depth. We note that the time complexity for this component is near-optimal, as the best known algorithm to verify a first-order graph property (which quantifies over only node sets) with at least 3 quantifiers has time complexity $\widetilde{O}(n^{q-0.63})$ [26], [27].

Property Name	Graph Type	Semantic Definition Type	Quantifier Type
PDF page-tree inheritance	Tree	Inherited Attributes	Universal
PDF faulty permissions attack	Graph	Node Quantifiers + Edge Quantifiers	Universal
HTML head element referenced at most once	Tree	Node Quantifiers + Edge Quantifiers	Universal
HTML unique refstrings	Tree	Synthesized Attributes	Universal
HTML nesting order of table elements	Tree	Node Quantifiers	Universal + Existential
HTML paragraphs cannot be nested	Forest	–	Universal
SVG 'Title' is first child	Ordered Tree	Node Quantifers	Universal
SVG references in 'defs'	Tree	Edge Quantifiers	Universal
SVG no use-use circularity	Tree	Node Quantifiers + Edge Quantifiers	Universal

TABLE I: Summary of Properties Validated.

VI. Case Studies

We demonstrate the use of our tool on 9 case studies across 3 file formats: PDF, HTML and SVG. We summarize the type of properties specified by each property in Table I. We already presented a detailed description of the PDF page-tree inheritance property earlier. We now give the specifications for 3 other case studies and the rest are mentioned in Appendix C.

HTML Head Elements Referenced at most once HTML files consist of start and end tags of elements that are nested within each other. We consider an HTML file to have a tree structure where an element is a child of another element if the child element's start and end tags are nested within the start and end tags of the parent element. The root of this tree is the 'html' element, which has two children: 'head' and 'body'. Often start tags have attributes that can be treated as key-value pairs. In HTML files, element A references element B if A's href attribute value equals B's id attribute value (called a refstring) pre-pended with a '#' symbol. The HTML Specification [28] recommends that all descendants of the 'head' element should be referenced at most once. This property is specified in Fig 4.

SVG 'Title' is First Child An ordered tree is defined here. The tree structure is the same as before, but the order on the children of an element is defined by the position of their 'Id' value in their 'Kids' entry of their parent element. Given this order on the children, the SVG specification [1] recommends that the 'Title' element is the first child of its parent. This property is checked by the specification in Fig 5.

DISV for PDF Attack Detection We briefly describe the Faulty Permission Attack [2] and give a specification to show how DISV can be used to detect whether a PDF file can be subject to such an attack.

We first define certified PDFs. A certified PDF defines permissions that allow certain changes to the file. Certifiers choose

```
Tree:
Root is unique d satisfying  d.<"Name">
= "html".

forall d in Tree, d0 is Child(d) where
d0.<"Id"> in d.<"Kids"> and d0.<"Name">
!= "html".

Semantic Definitions:
Head is d satisfying d.<"Name"> = "head".

Ref is (d1, d2) satisfying d1.<"href">
= REFSTRING(d2.<"id">)
where d1 in [Tree] where d2 in [Tree].

Semantic Property:
forall d1 in Head, forall (d2, d3) in Ref,
forall (d4, d5) in Ref,
(d3 = d5 and d3.<"Id"> in
PATH(d1.<"Id">,Tree)) implies d2 = d4.
```

Fig. 4: Head Element Referenced Atmost Once

```
Ordered Tree:
Root is unique d satisfying d.<"Name"> =
"svg".

forall d in Ordered Tree, d0 is ith Child(d)
where d0.<"Id"> in d.<"Kids">  and
d0.<"Name"> !=  "svg" where i is
indexn (d0, d.<"Kids">).

Semantic Definitions:
Title is d satisfying d.<"Name"> = "Title".

Semantic Property:

forall d in Title,
ochild_field(parent_field(d,<"Id">)
,1,<"Id">)=d.<"Id">.
```

Fig. 5: ‘Title’ is First Child Property.

```
Graph:
Source is d satisfying True.

forall d in Tree, d0 is Child(d) where False.

Semantic Definition:
XRef is d satisfying d.<"Type"> = "XRef".

Certificate is d satisfying
iskeydefined(d,<"Reference.0 .
TransformParams.P">).

XRef_Certificate is d satisfying
d.<"Type">="XRef" and (d1>d2
implies d1>=d) where d1 in [XRef]
where d2 in [Certificate].

Digital_Form is d satisfying
d.<"header.Subtype"> = "Form" and
d.<"header.Type"> = "XObject".

Annotation is d satisfying
d.<"Type"> = "Annot".

Semantic Property:
forall d1 in XRef_Certificate,
forall d2 in Annotation,
forall d3 in Certificate,
d3.<"Reference.0 .
TransformParams.P"> < 3
implies d2 < d1.

forall d1 in XRef_Certificate,
forall d2 in Digital_Form,
forall d3 in Certificate,
d3.<"Reference.0 .
TransformParams.P"> < 2
implies d2 < d1.
```

Fig. 6: Specification of a property for preventing faulty permission Level attacks (The graph contains all objects in the input file and no edges are needed for this property).

between 3 different permission levels to allow incremental updates, with P1 being stricter than P2, which is in turn stricter than P3. A PDF document can have at most one certification.

- P1: No modifications allowed.
- P2: Filling out form fields and digitally signing the document are the only modifications allowed.
- P3: All incremental updates are allowed.

Faulty Permissions Attack. If an attacker forces a disallowed modification to a certified document, the certification breaks. Recent studies [2] on certain PDF readers demonstrated attacks that allowed modifying a certified PDF file with changes not within the permission level, and avoided detection by those PDF readers. This can happen if annotations are added to PDF documents with a permission level stricter than P3 or if forms are modified in PDF documents with a permission level stricter than P2.

In a PDF file with a permission level stricter than P3 (i.e., P1 and P2), no annotations should be allowed after the PDF file has been certified, whereas in a PDF file with a permission level stricter than P2 (i.e., P1), additionally no form fields are allowed after the PDF file has been certified. To ensure that there are no annotations or form fields after certification, we need to check if there are any annotations or form fields in the incremental update sections after the document has been certified. This property is specified in Fig 6. To do this, we first locate the XRef section of the original certified document labelled ‘XRef_Certificate’. We then check that all incremental updates after this XRef section are within the set of changed allowed by the file’s certification level. Note that the certification level of a PDF document is stored in the ‘P’ (permissions) attribute of the TransformParams object.

VII. Implementation and Performance Evaluation

We implemented DISV in C++. The implementation has about 4,600 LOC, with finding the graph structure taking 1K LOC and verifying the property taking 2K LOC.

A. Performance evaluation

We present the time vs input size plots for validating the semantic property for the case studies discussed in Section VI

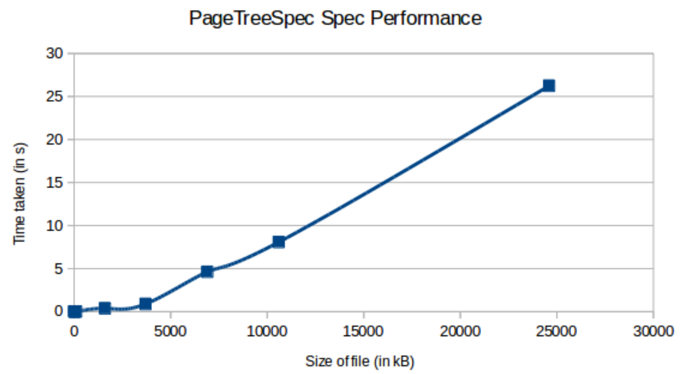
and the remaining performance plots are in Appendix C. For the page-tree inheritance property, the numbers of nodes and edges are much smaller than the number of objects in the input file ($n, e << D$), and our observed time complexity is linear in D. In the property of HTML head elements referenced at most once, we have $n \sim D$, and we observe quadratic behavior in D, whereas the SVG-Ordered Spec shows linear behaviour w.r.t. D; the file size of 350kB is an outlier as it has very few objects in the user defined nodes. This is consistent with our knowledge that the running time of DISV is O($s+nD+n^q+e^q$), where s is the size of the specification file, D is the total number of objects in the input file, n and e are the numbers of nodes and edges in the graph respectively, and q is the quantifier depth in the specification file.

We also conducted additional experiments to compare the running time of graph construction and the running time of semantic property checking for the property of HTML nesting order of table elements. We plot the results in Fig 8(a). From this, we infer that the majority of the execution time for DISV lies in searching for the graph structure.

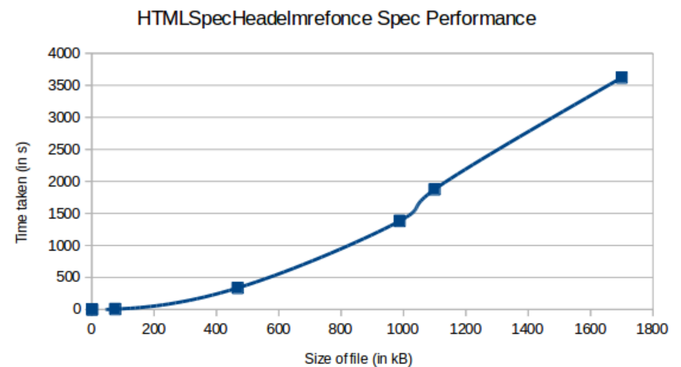
We note that as graph construction contributes to the bulk of the execution time for DISV, a more precise definition of an input graph would result in a much smaller running time. We depict this for the HTML-paragraph-cannot-nest property in Fig 8(b). The red line is the performance for the specification given in Fig 10, whereas the blue line is the performance for when the get graph specification of Fig 10 is replaced with the get graph specification in Fig 9.

VIII. Future Work

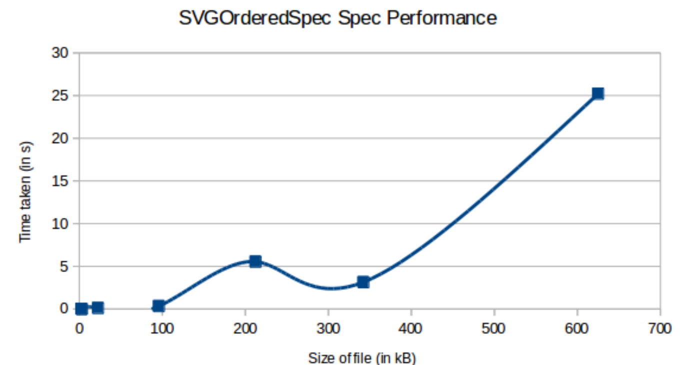
There are a few directions for future work for our work. One direction is to extend the graph logic specification used by DISV to allow for second order graph logic properties (MSO1 or MSO2). Another direction would be to allow for properties to be specified in a topologically unsorted (w.r.t dependencies) fashion. Another direction in this project would be to expand the database of atomic predicates that DISV allows. A completely different direction in the field of security would be to use DISV to detect security attacks such as EAA and SSA attacks on PDF documents, which break the integrity of a PDF document [2].



(a) PDF: Page Tree Inheritance



(b) HTML: Head elements referenced atmost once



(c) SVG: 'Title' is first child

Fig. 7: Performance plots for Specifications

Acknowledgement

The authors would like to thank anonymous reviewers for their insightful comments. This work was supported by DARPA research grant HR0011-19-C-0073.

References

[1] W3C, "Scalable vector graphics (svg) 2," https://www.w3.org/TR/SVG2/, 2018.

[2] C. Mainka, V. Mladenov, and S. Rohlmann, "Shadow attacks: Hiding and replacing content in signed pdfs," 2021.

[3] V. Mladenov, C. Mainka, K. Meyer zu Selhausen, M. Grothe, and J. Schwenk, "1 trillion dollar refund: How to spoof pdf signatures," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1–14.

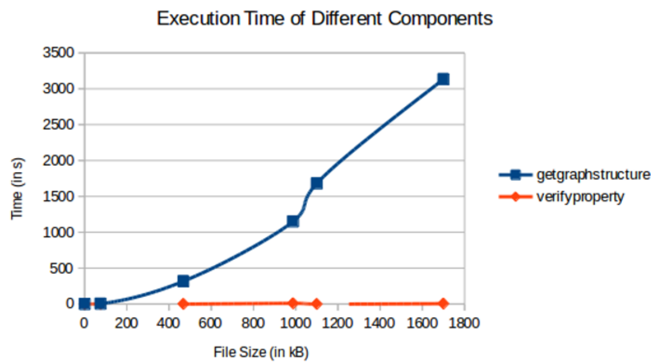
[4] J. Müller, F. Ising, V. Mladenov, C. Mainka, S. Schinzel, and J. Schwenk, "Practical decryption exfiltration: Breaking pdf encryption," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 15–29.

[5] S. Rohlmann, V. Mladenov, C. Mainka, and J. Schwenk, "Breaking the specification: Pdf certification."

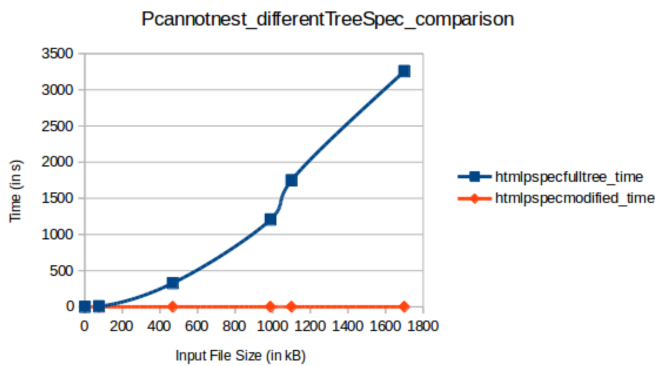
[6] L. W. Li, G. Eakman, E. J. Garcia, and S. Atman, "Accessible formal methods for verified parser development."

[7] F. Momot, S. Bratus, S. M. Hallberg, and M. L. Patterson, "The seven turrets of babel: A taxonomy of langsec errors and how to expunge them," in *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 2016, pp. 45–52.

[8] P. Wyatt, "Work in progress: Demystifying pdf through a machine-readable definition," 2021.



(a) Running time breakdown for the property of for HTML nesting order of table elements.



(b) Graph construction times for HTML-paragraph-cannot-nest property.

Fig. 8: Performance Comparison plots

[9] L. Rosenthol, "Pdf validation," *PDF Technical Conference 2012, Basel, Switzerland. Unpublished.*

[10] "Verapdf model," https://github.com/veraPDF/veraPDF-model.

[11] "Verapdf model syntax," https://github.com/veraPDF/veraPDFmodel-syntax.

[12] "Pdf runtime validation and beyond..., françois fernandès (levigo solutions gmbh), pdf technical conference 2013, königswinter, germany," https://github.com/levigo/pdf-formal-representation.

[13] "Constraint validation api," https://developer.mozilla.org/en-US/docs/Web/API/Constraint_validation.

[14] U. Şimşek, E. Kärle, O. Holzknecht, and D. Fensel, "Domain specific semantic validation of schema. org annotations," in *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer, 2017, pp. 417–429.

[15] O. Panasiuk, E. Kärle, U. Simsek, and D. Fensel, "Defining tourism domains for semantic annotation of web content," *arXiv preprint arXiv:1711.03425*, 2017.

[16] I. Boneva, J. E. L. Gayo, and E. G. Prud'hommeaux, "Semantics and validation of shapes schemas for rdf," in *International Semantic Web Conference*. Springer, 2017, pp. 104–120.

[17] S. Kolozali, T. Elsaleh, and P. M. Barnaghi, "A validation tool for the w3c ssn ontology based sensory semantic knowledge," in *Joint Proceedings of the 6th International Workshop on the Foundations, Technologies and Applications of the Geospatial Web and 7th International Workshop on Semantic Sensor Networks*, 2014, pp. 83–88.

[18] M. Navarro, F. Orejas, E. Pino, and L. Lambers, "A navigational logic for reasoning about graph properties," *Journal of Logical and Algebraic Methods in Programming*, vol. 118, p. 100616, 2021.

[19] D. E. Knuth, "Semantics of context-free languages," *Mathematical systems theory*, vol. 2, no. 2, pp. 127–145, 1968.

[20] H. Alblas, "Introduction to attribute grammars," in *International Summer School on Attribute Grammars, Applications, and Systems*. Springer, 1991, pp. 1–15.

[21] S. Karol, "An introduction to attribute grammars," *Department of Computer Science. Technische Universitat Dresden, Germany*, 2006.

[22] K. Thirunarayan, "Attribute grammars and their applications," in *Encyclopedia of Information Science and Technology, Second Edition*. IGI Global, 2009, pp. 268–273.

[23] P. Deransart and M. Jourdan, "Attribute grammars and their applications," *Lecture Notes in Computer Science*, vol. 461, 1990.

[24] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan, "Silver: An extensible attribute grammar system," *Science of Computer Programming*, vol. 75, no. 1-2, pp. 39–54, 2010.

[25] "Document management—portable document 493 format—part 1: Pdf 1.7," 2008.

[26] R. Williams, "Faster decision of first-order graph properties," in *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2014, pp. 1–6.

[27] J. Alman and V. V. Williams, "A refined laser method and faster matrix multiplication," in *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2021, pp. 522–539.

[28] W3C, "Html 5.2," https://www.w3.org/TR/html52/, 2021.

[29] G. Endignoux, O. Levillain, and J.-Y. Migeon, "Caradoc: A pragmatic approach to pdf parsing and validation," in *2016 IEEE Security and Privacy Workshops (SPW)*. Ieee, 2016, pp. 126–139.

Appendix

We next briefly describe the structure of Portable Document Format (PDF) [29], [25].

A. PDF File Structure

In its basic form, a PDF file has four elements:

- *Header* stores the PDF version number.
- *Body* contains an uncompressed list of *indirect objects*, which are explained later.
- *Xref Table* stores the byte position of each object in the file. An Xref table is a collection of xref sections, where each xref section is a continuous list of entries that store information about a new incremental update. We will briefly explain incremental updates later.
- *Trailer* stores information about the root object of the PDF document. It is demarcated by a End of File (EOF) marker.

B. Basic PDF Objects

We briefly describe the basic forms of PDF objects: direct objects, indirect objects, and streams.

Direct Objects PDF objects have a few basic object types:

- a *null* object is represented by the keyword 'null'
- *boolean* objects are 'true' and 'false'

- *integers*; e.g. 123, 45
- *reals*; e.g. -2.8 , 3.0
- *strings* are enclosed in parenthesis; e.g. (man)
- *names* are preceded by a slash; e.g. /Date
- *arrays* are heterogeneous collections of any of the above objects enclosed in square brackets. For example, [/Name (John) /Year 1975] is an array object where the first object '/Name' is a name object, the second object '(John)' is a string object which is followed by another name object '/Year' and then integer 1975.
- *dictionaries* are collections of key-value pairs enclosed in double angle brackets, where the key is a name object and the value could be any direct object including another dictionary. For example, ¡¡ /Name (John) /Date [1 1 1975] ¿¿ is a dictionary with 2-key value pairs, the first being '/Name (John)' where '/Name' is the key and the string object '(John)' is the value; the second being '/Date [1 1 1975]' where '/Date' is the key and the array object '[1 1 1975]' is the value.

Indirect Objects Indirect Objects are objects enclosed between keywords 'obj' and 'endobj'. An example indirect object is given in lines 3–8 of Fig 12. An indirect object is identified using two integers called the object number and the generation number respectively. The object number is unique for each object. The generation number is used to keep track of updates on the object; hence a new object has 0 as its generation number. For instance, the first object in Fig 12 has object number 1, generation number 0, and contains a dictionary with 4 keys: /Lang, /OpenAction , /Pages, and /Type. An indirect object is referred to by using a reference to its object and generation numbers; for example, the example object can be referred to as '1 0 R'.

Stream Objects A stream object is a special kind of indirect object and contains a byte sequence enclosed within the keywords 'stream' and 'endstream'. The object is used to store metadata and allows to use compression filters. An example stream object is given in lines 23–27 of Fig 12. This stream object has object number 5, generation number 0, contains a dictionary with 2 keys (/Filter and /Length), and has a bytestream on line 26.

C. PDF Document Structure

From a semantic point of view, indirect objects in the PDF body form a directed graph, with nodes being the indirect objects and edges being references between indirect objects. A few important entities in the graph are mentioned below:

- *Pages* are indirect objects that store graphical content for pages of a PDF file. Each PDF page has its own separate indirect object. An example of a page object is given in lines 14–21 of Fig 12.
- *Trailer* is the root of the digraph. It contains metadata for the PDF file and references the PDF Catalog indirect object. An example of a trailer object is given in lines 39–43 of Fig 12.
- *Catalog* references the remainder of the document. It references the root of the Page Tree Structure which in-turn references the entirety of the Page Tree. It's location is stored by the /Root key in the Trailer object. An example of a Catalog object is given in lines 3-8 of Fig 12.

Incremental Updates An incremental update allows to extend a PDF by appending new information at the end of the file. Incremental updates are of two types:

- *Forms* allow for user input in predefined form fields and cannot change other content in the PDF.
- *Annotations* are not restricted to predefined places and can be applied anywhere in the document.

We first describe the nonterminals used in the graph specification. The nonterminal getgraph_Prop can either be the infix functions 'or' , 'and' , 'in' , '=' or the prefix functions 'not', 'inn' , 'ins' , 'eqs', 'eqn'. The function 'in' is a polymorphic function that accepts 2 arguments and checks if the first argument is contained in the second. The functions 'inn' and 'ins' are type-dependent implementations of 'in', where n stands for number and s for string. The same holds for 'eqs' and 'eqn'. The nonterminal 'getGraph_Index' can be either indexn or indexs, where n and s stand for numbers and strings respectively. indexn inputs a number and a list of numbers and returns the position of the number in the list; indexs is defined similarly.

⟨getgraph_Prop⟩ ::= *or* | *and* | *not* | *in* | *inn* | ins | = | *eqs* | *eqn* | *True* | *False*
⟨getGraph_Index⟩ ::= *indexn* | *indexs*

We now describe the nonterminals used in semantic definition specification. The nonterminal 'Attr_Set' can be either 'Root', 'Leaves' or 'Tree'. The nonterminal 'Attr_func' can either be 'make_singleton_array', 'dictval' , 'append_all_children_attributes' , 'union' , 'or' , 'and', 'not', 'is_empty_array' or 'ancestor'. 'make_singleton_array' takes a single value and converts it to a singleton array. 'dictval' takes 2 arguments, an object and a key string, and returns the value of that key of the object - the object may not have an explict value and may be written in terms of other functions. 'append_all_children_attributes' takes a object and an attribute name, and returns a list of the attribute values of all children of that object.'union' is union of two list, 'is_empty_array' checks if a list is empty or not. Similarly the nonterminals 'Attr_Prop' and 'Graph_Prop' are defined below. 'Graph_Edge_Set' is an array of either 'Quantifer_Edge_Set' or 'Graph_Spec"s. The nonterminals 'attr_name', 'Node_name' and 'Edge_name' can be any alphanumeric string. 'attr_name' cannot be enclosed in quotes, which are reserved for key names of objects.

⟨Attr_Set⟩ ::= *Root* | *Leaves* | *Tree*
⟨Attr_fun⟩ ::= *make_singleton_array* | *dictval* | *union* | *append_all_children_attributes* | *or* | *and* | *not*
⟨Attr_Prop⟩ ::= *is_set* | *is_empty_array* | *True* | *False* | *and* | *or* | *not* | *in* | *inn* | *ins* | *eq* | *eqs* | *eqn* | *implies*
⟨attr_name⟩ ::= $[a - zA - Z0 - 9]^+$
⟨Graph_Prop⟩ ::= *is_empty_arr* | *True* | *False* | *or* | *and* | *not* | *in* | *implies* | *inn* | ins | *eq* | *eqs* | *eqn*
⟨Graph_Edge_Set⟩ ::= [⟨Quantifier_Edge_Set⟩$^+$ | ⟨Graph_Spec⟩]
⟨Node_name⟩ ::= $[a - zA - Z0 - 9]^*$
⟨Edge_name⟩ ::= $[a - zA - Z0 - 9]^*$

We now give the non terminals used in the semantic property specification.

⟨Sem_Prop_Prop⟩ ::= | *or* | *and* | *not* | *in*
| *inn* | *ins* | *eq* | *eqs* | *eqn* | *implies*
is_set | *parent_field* | *grandparent_field*
| *ancestor* | $<$ | *ochild_field* | *PATH*
| *True* | *False*

⟨Sem_Prop_Set⟩ ::= ⟨Quantifier_Node_Set⟩ | ⟨Quantifier_Edge_Set⟩
| ⟨Graph_Spec⟩ | *Root* | *Leaves* | *PATH*

HTML: Unique Refstrings Property The tree structure is the same as the one in the property that ensures HTML head elements are referenced at most once. Each refstring corresponding to a reference not to a 'cite' object must be unique [28]. This property is specified by using a synthesized attribute 'id_def', which stores all the refstrings defined on a path from that node to some leaf node. We finally check that all refstrings are unique by checking for double occurrences of a refstring in the 'id_def' value of the root node. This property is specified as follows:

```
Tree:
Root is unique d satisfying  d.<"Name"> = "html".
forall d in Tree, d0 is Child(d) where
d0.<"Id"> in d.<"Kids">
  and d0.<"Name"> !=  "cite" .

Semantic Definitions:
forall d in Leaves, d.<id_def> =
make_singleton_array( d.<"id"> ).
forall d in Tree,
d.<id_def>=append_all_children_attributes(d.<"Id">,
          <id_def>) union make_singleton_array
          (d.<"id"> ).

Semantic Property:
forall d in Root, is_set(d.<id_def>).
```

HTML: Nesting Order of Table Elements The tree structure is the same as the one described before. A table in HTML is represented by the 'table' element. 'TR', 'TD' and 'TH' are descendants of a 'table' element and represent a row in a table, a data cell, and a header cell, respectively. A 'TD' and 'TH' element must necessarily be a child of a 'TR' element; however, a 'TD' or 'TH' element cannot be a table itself, implying that a 'TR' element cannot be a child of a 'TD' or 'TR' element. This property is specified as follows:

HTML: Paragraphs cannot be nested For this property we define the graph structure as the one whose sources are all 'p' nodes and kids are as defined before. A paragraph tag (i.e. a 'p' tag) cannot be nested within each other. This property is specified as follows:

SVG: References in 'defs' An SVG file is similar to an HTML file with the difference being that the start element is "svg" instead of "html". The tree structure for an SVG file is defined similar to the tree structure for an HTML file. References between elements in an SVG file are the same as references in an HTML file. The SVG specification [1] recommends that all referenced elements must be children of the 'defs' element, and all referencing elements must be

```
Tree:
Root is unique d satisfying  d.<"Name"> = "html"  .
forall d in Tree, d0 is Child(d) where d0.<"Id">
in d.<"Kids">  and  d0.<"Name"> !=  "html".

Semantic Definitions:
TD is d satisfying d.<"Name"> = "TD".
TR is d satisfying d.<"Name"> = "TR".
TH is d satisfying d.<"Name"> = "TH".

Semantic Property:
forall d in TD, exists d1 in TR, d.<"Id"> in d1.<"Kids">.
forall d in TD, forall d1 in TR,
not( d1.<"Id"> in d.<"Kids">).
forall d in TH, exists d1 in TR, d.<"Id"> in d1.<"Kids">.
forall d in TH, forall d1 in TR,
not(  d1.<"Id"> in d.<"Kids">).
```

Fig. 9: Nesting Order of Table Elements Property.

```
Forest:
Root is d satisfying  d.<"Name"> = "p" .
forall d in Forest, d0 is Child(d) where
d0.<"Id"> in d.<"Kids">
and  d0.<"Name"> !=  "p" .

Semantic Definitions:

Semantic Property:
forall d in Source, forall d1 in Source, not (d in d1).
```

Fig. 10: Paragraph elements cannot be nested.

siblings of the 'defs' element. This property is specified as follows:

```
Tree:
Root is unique d satisfying d.<"Name"> = "svg" .
forall d in Tree, d0 is Child(d) where
d0.<"Id"> in d.<"Kids">
and d0.<"Name"> != "svg"  .

Semantic Definitions:
Ref is (d1, d2) satisfying d1.<"href"> =
REFSTRING(d2.<"Id">)
where d1 in [ Tree ] where d2 in [ Tree ] .

Semantic Property:
forall (d1,d2) in Ref, parent_field(d1 , <"Id">) =
grandparent_field( d2, <"Id"> )
and  parent_field(d1, <"Name">) = "defs"  .
```

SVG: No Use-use Circularity The tree structure is as the one defined in the previous SVG property. A use-reference in an SVG file is a reference between a 'use' and a 'symbol' element or a reference between a 'use' and a 'use' element. A use-reference is denoted by an edge, and the set of these edges is stored in the edge-set 'Ref_use'. The SVG specification [1] requires that there cannot be a cycle of use-reference edges. This property is specified as follows:

```
Tree:
Root is unique d satisfying d.<"Name"> = "svg" .
forall d in Tree, d0 is Child(d) where  d0.<"Id">
in d.<"Kids">  and d0.<"Name"> != "svg"  .

Semantic Definitions:
use is d satisfying d.<"Name"> = "use" .
symbol is d satisfying d.<"Name"> = "symbol" .
Ref_use is (d1, d2) satisfying d1.<"href"> =
REFSTRING(d2.<"Id">)
where d1 in [ use ] where d2 in [ use , symbol ] .

Semantic Property:
forall d3 in use, forall d4 in PATH(d3, Ref_use),
not( ancestor(d3,d4) or d3 = d4 ).
```

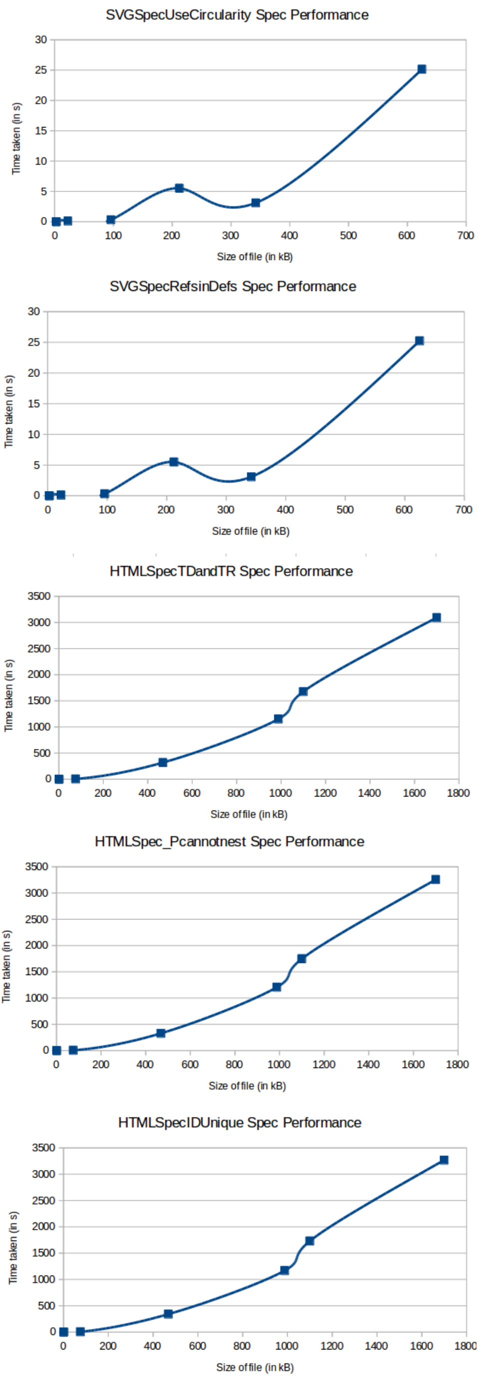


Fig. 11: Performance plots for remainder of Specifications

We present the performance evaluations for the remainder of the case studies in Fig 11.

```
1  %PDF-1.4
2  %M-?M-wM-"M-~
3  1 0 obj
4  << /Lang (en-US)
5      /OpenAction [ 3 0 R /XYZ null null 0 ]
6      /Pages 4 0 R
7      /Type /Catalog >>
8  endobj
9  2 0 obj
10 << /CreationDate (D:20201218162631-05'00')
11     /Creator <feff0057007200690074006500720>
12     /Producer <feff004c006900620072 ... > >>
13 endobj
14 3 0 obj
15 << /Contents 5 0 R
16     /Group << /CS /DeviceRGB /I true /S
17          /Transparency >>
18     /MediaBox [ 0 0 612 792 ]
19     /Parent 4 0 R
20     /Resources 6 0 R
21     /Type /Page >>
22 endobj
23 ...
24 5 0 obj
25 << /Filter /FlateDecode /Length 36 >>
26 stream
27 xM-^\3M-P3T(M-g*T0^@B3C#^EsK#M-^EM-"TM-
28 ... endstream
29 endobj
30 ...
31 xref
32 0 8
33 0000000000 65535 f
34 0000000015 00000 n
35 0000000117 00000 n
36 0000000295 00000 n
37 0000000454 00000 n
38 0000000556 00000 n
39 0000000662 00000 n
40 0000000719 00000 n
41 trailer << /DocChecksum /84244FF450963B148
42            /Info 2 0 R
43            /Root 1 0 R
44            /Size 8
45            /ID [<d41805d1a8b6 ... >] >>
46 startxref
47 740
48 %%EOF
```

Fig. 12: A sample PDF file (with certain content omitted and represented by ellipses).