Integrated Project Report: Extending Prova for Architecture Definition Traceability

Fabio Fernandes PG26119 & Telma Correia PG25263

University of Minho, Informatics Department
http://www.di.uminho.pt

Abstract. In nowadays the architecture of software is very important because meets a set of structures needed to reason about the software system, which comprises the software elements, the relation between them, and the properties of both elements. aims at guiding the project implementation and ensures that future requirements can be understood and analysable. Therefore the integration of AADL language in PROVA will be the theme of this project so that it can be a tool with greater capacity for analysis and correction.

1 Introduction

When we think of architecture, a diverse set of attributes come to mint. At a general level, the first thing that reminds us is the development of something big, like buildings. Certainly we immediately think of a physical structure, we think in something tangible. But in reality, the architecture is much more than that, is how various components engage each other in order to obtain a structure as a whole. The architecture is the structure of something, is how something can be produced. As in the development of buildings, on Software the architecture is also present, because the global society is increasingly supported by software-based technology. Wherever we go it exists, from a small register machine in a coffe, to a large hospital centers, where software is fundamental and can't fail, for this reason a good structure,a good architecture is needed.

Citing Pfleeger in 1998, "A software architecture involves the description of elements architectural systems of which will be built, interactions between those elements, patterns that guide their composition, and constraints on these standards. "

Being the planning and architecture of a system is as crucial to the software development role, behold, the PROVA, a tool produced by the company Educed, which aims to help in modeling and behavioral analysis system emerges. It is then through this company that arises the purpose of this project, which aims to develop an extension for PROOF, through the integration of a language of Architectural and Behavioral Analysis System, the AADL.

The project has target of a long process of research and learning so, the following report will specify all the work methodology, technologies used and the respective outcome. To such shall be divided into five main sections:

-In section three, the PROVA is presented.

-In sections four and five the studies concerning the AADL, the whole structure and syntax will be presented.

-In section six, the translation between AADL and PROVA, also the technologies that we use and the semantics actions that we gave to our tool to generate the boilerplates in PROVA.

-Finally section in seven will discuss the final conclusions and future work.

It should also be noted that project was developed within the framework of the Master of Computer Engineering, in Formal Methods in Software Engineering, and supervised by engineer Jose Faria co-founder of educed, and Professor Luis Barbosa.

2 Presentation of Project

2.1 The Proposal

To improve the functionality of PROVA was proposed to make an extension for the same, in order to support the definition of system architectures.

The reference language that will be used for this purpose will be the AADL.

PROVA already supports the graphical modelling of components. So first goal of this project is extend the existing graphical components library with the necessary elements fo AADL Language and implement the back-end support for the storage and management of the architecture components.

The second goal is the support of traceability between requirements and architecture.

it should be mentioned that in the middle of the development of the project, it was discovered that the OSATE (tool support AADL) has been updated and allowed the user to double standards, in other words, allows user to use the code and graphics to develop a model, and moved whenever one of the criteria, the other is automatically updated. So, from there just had to focus on development of the second goal.

3 The PROVA

"The best in requirements engineer captured in one simple tool: Efficient, Enabling, Everywhere".

Prova is an innovative tool for the development of high assurance systems, and aims to be a complete tool of modeling and Behavior analysis of a system. It is capable of analyzing textual requirements and 1. identifying possible conflicts between them, 2. generating simulations of the system behavior 3. Assuring that given properties are met,or, if not 4. generating visual scenarios ilustrating the error.

The PROVA, is currently, in a phase of great expansion for it to be a great asset in support of high assurance systems. Tended to have the most varied features to help in systems of high assurance, without errors, ambiguities inconsistencies and omissions.

3.1 "Architeture" of Prova

The PROVA is a tool that works via the web, and uses web services for integration of various components. So, we can say that this will have two layers: The FrontEnd and the BackEnd.

The Front-End layer is responsible for what is shown to the user. And the Back-End, where is our work, is the layer responsible for dealing with boilerplates, do the translation between the AADL code and PROVA, and then send the respective model.

3.2 Boilerplates

The boilerplates are known as the standardization of a language in order to simplify its structure to make it more efficient. In the case of the PROVA boilerplates are recipes for making common types of requirements by providing them the terms involved in each case.

In this project we only use the Boilerplates of Entity Model,which describe the static behavior between the entities and relations.

For definition of AADL boilerplates is only necessary five types of them, which are:

Mult - Cardinality of an entity/relation, there are m rel
Assoc - relation between entities every A shall have m fixed? r B
Gen - relation between two relations every r1 is a r2
Abstract - an entity is abstract, r is abstract
Extends - inheritance, r extends s

4 The MBE - Model-Based Engineering

The Model-Based Engeering involves the creation and analyse of models of systems from which can predict and understand its capabilities and operational qualities, like as Performance, Reliability/Confidence and Safety.

In other words the MBE is a Model who provides increased in predictability in systems integration through the life-cycle analysis. You can predict failures at the system level and avoid costly rework in development and maintenance.

Previously, different parts of the system were handled separately, which could lead to various problems. But with the approach of MBE have an architecture centered on a single system.

Architecture-centric approaches address system-level issues and maintain a self-consistent set of analytical views of a system such that individual analyses retain their validity amidst architectural changes within the set.

5 The AADL - Architecture Analysis & Design Language

The AADL is an language of architecture description santandardized by SAE, and is a unifying framework for MBE.

This Language provides formal methods concepts for the analysis and description of application systems architecture. This is a language textual and graphical and is used to model Software and Hardware architecture of an embedded, real-time system.

The AADL define a pattern to describe system components, interface, and relations of components, facilitates the automation of code generation, system build, and other development activities; and significantly reduces design and implementation defects.

In developing an AADL model, you represent the architecture of your system as a hierarchy of interacting components. You organize interface specifications

and implementation blueprints of software,hardware, and physical components into packages to support large-scale and team-based development.

5.1 AADL Sintax Summary

In AADL, components are defined through type and implementation declarations. A Component Type declaration defines a component interface elements and externally observable attributes. A Component Implementation declaration defines a components internal structure in terms of subcomponents, subcomponent connections, subprogram call sequences, modes,flow implementations, and properties. Components are grouped into application software, execution platform, and composite categories. Packages enable the organization of AADL elements into named groups.

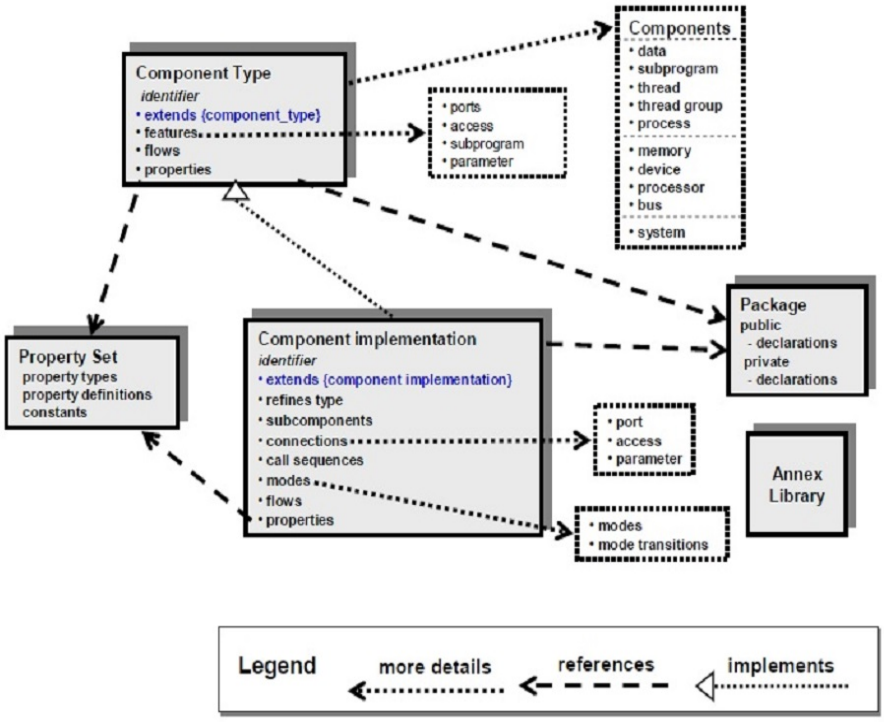

Fig. 1. Summary of AADL Elements

Components are the core of modeling vocabulary for the AADL. Components has a name (unique identity) and are declared as a type and implementation

within a particular component category. A component category defines the runtime essence of a component. There are three distinct sets of component categories:

1. **application software:**
   a. **thread**: a schedulable unit of concurrent execution;
   b. **thread group**: a compositional unit for organizing threads;
   c. **process**: a protected address space;
   d. **data**: data types and static data in source text
   e. **subprogram**: callable sequentially executable code

2. **execution platform:**
   a. **processor**: components that execute threads
   b. **memory**: components that store data and code
   c. **device**: components that interface with and represent the external environment
   d. **bus**: components that provide access among execution platform components

3. **composite:**
   a. **system**: a composite of software, execution platform, or system components

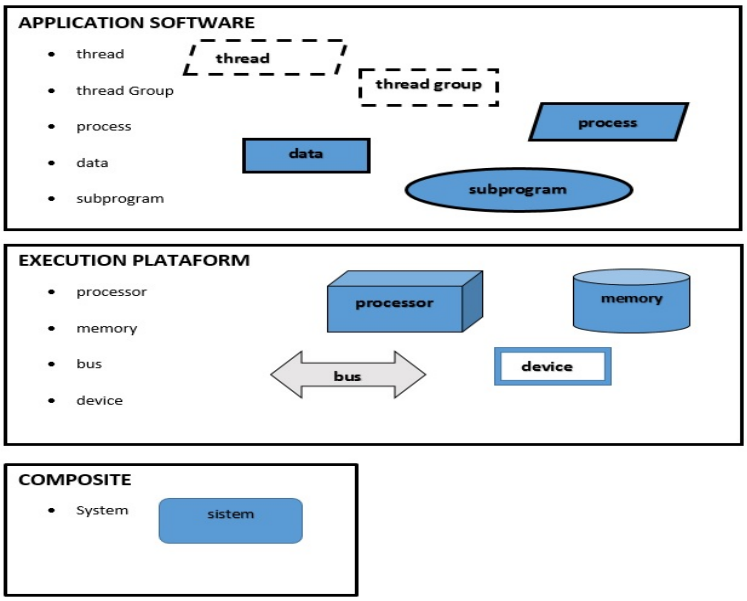

Fig. 2. Representation of AADL components

Components Types An AADL component type declaration establishes a component externally visible characteristics. This caracteristics can have the content:

. extends
. prototypes
. features
. flows
. modes
. properties
. annex subclauses

Components Implementations specifies an internal structure in terms of subcomponents, interactions (calls and connections) among the features of those subcomponents, flows across a sequence of subcomponents, modes that represent operational states, and properties. May have the following content:

. extends
. prototypes
. subcomponents
. connections
. calls
. flows
. modes
. properties
. annex subclauses

AADL Syntax Examples In this setion we present an example of applying AADL.

Declaration of Package

```
package demo
end demo;
```

Declaration of a Component

```
system example_system
end example_system;
```

Declaration of a Component implementation

```
system implementation example_system.impl
end example_system.impl;
```

Declaration of a Component implementation with content

```
system implementation example_system.impl
    subcomponents
      net : system democomponents::navigation_system_ext;
      pp : system democomponents::position_processor;
      ui : system democomponents::user_interface.impl;
      switch : device democomponents::ethernet_switch;
      cbl1 : bus democomponents::ethernet_bus;
      cbl2 : bus democomponents::ethernet_bus;
      cbl3 : bus democomponents::ethernet_bus;
    connections
      eth_cbl_cbl1 : bus access net.eth_cbl -> cbl1;
      cbl1_switch : bus access cbl1 -> switch.e1;
      eth_clb_cbl2 : bus access pp.eth_cbl -> cbl2;
      cbl2_switch : bus access cbl2 -> switch.e2;
      eth_clb_cbl3 : bus access ui.eth_cbl -> cbl3;
      cbl3_switch : bus access cbl3 -> switch.e3;
      net_to_pp : feature net.iface -> pp.in_iface;
      pp_tp_ui : feature pp.out_iface -> ui.iface;
    flows
      position_flow : end to end flow net.position_source ->
            net_to_pp -> pp.position_flow -> pp_tp_ui
                  -> ui.position_sink;
  end example_system.impl;
```

6 AADL to PROVA

Before attacking the problem, the translation between AADL and PROVA, it was necessary to take some decisions, such as, in PROVA we have Entities, in AADL we have components and each component have its own type, so you see the problem here, it was necessary to find a solution to represent component types in Prova and find a way to distinguish them, so to do that, our solution was to use the type of each component and concat it with its own identifier, for instance, we can declare a system in AADL this way "***system*** *exemple1*", and you can guess the result in terms of its representation in PROVA, that is *SYSTEM_exemplo1*, we choose capitalize the type of the component, so this way it is more easy to identify and distinguish component types in requirements of the PROVA. In Prova we also have relations between Entities, in AADL we have some kind of connections between the components, like *flows*, *features*, *connections* and more, so again we have to find a way to represent those connections in Prova, and distinguish them, for instance if we have "***system*** *exemple1*" and if the system that we declare have some features, like this "*features identify : exemple2*" we can say that the *system example1* have a relation *FEATURES_identify* to *example2*, we can notest that again we use the type of the connection, in this case we declare a *feature* and concat its with is own identifier, again, doing this way it is

more easy to distinguish in PROVA the many connections that an AADL model can have, so in resume, we have components in AADL that are Entities in Prova, and connections in AADL that are relations in PROVA, you can see with more details, the translation between AADL and PROVA e the next sections.

6.1 Technologies

To do the translation between AADL and PROVA, we need to build a parser for AADL Language, and add the semantics rules to generate the boilerplates[1], to do that we use ANTLR[2], *ANother Tool for Language Recognition*, is a parser generator that uses LL(*)[3] parsing.

ANTLR takes as input a grammar that specifies a language and generates as output source code for a recognizer for that language. A language is specified using a context-free grammar which is expressed using Extended BackusNaur Form (EBNF[4]). ANTLR allows generating lexers, parsers, tree parsers, and combined lexer-parsers.

We build a context-free Grammar, that we will show you in the next section, and with the semantics rules that we add to our grammar we generate the boilerplates.

6.2 Antlr Grammar

In this section we will show our Grammar, without semantics rules. we decide do not put all the productions of our grammar here, so we put an excerpt with just the most important rules, the complete grammar will be added to the annexes.

```
aadl : 'package' IDENTIFICADOR ('public' | 'private') components
            'end' IDENTIFICADOR ';'
     ;
components : (component_category | component_implementation)*
           ;
components_category : (component_category)+
           ;
component_category : component_type IDENTIFICADOR
                  | component_type IDENTIFICADOR 'extends' IDENTIFICADOR
                      (internal_components)* 'end' IDENTIFICADOR ';'
                  ;
```

[1] templates to specify requisites in Prova
[2] http://www.antlr.org/
[3] top-down parser
[4] http://en.wikipedia.org/wiki/Extended_BackusNaur_Form

```
internal_components : component_features
                    | component_modes
                    | component_flows
                    ;
components_implementation : (component_implementation)+
                          ;
component_implementation : component_implementation_type IDENTIFICADOR
                              (internal_implementation)* 'end' IDENTIFICADOR ';'
                         ;
internal_implementation : impl_subcomponents
                        | impl_flows
                        | impl_connections
                        | impl_modes
                        ;
```

We can observe that our language start with a *package* that can be *public* or *private* and we can have components, and each component can be a *component_category* ,here we refer to the component types like *system, bus, process, etc*, or *component_implementation*, here we refer to the implementation of the components, its important to say that we can not have components implementation without have its respective component type defined. Inside *component_category* we can have *component_features, component_modes, component_flows* that are **features, modes, flows** respectively, and inside *component_implementation* we can have *impl_subcomponents, impl_flows, impl_connections, impl_modes* that are **subcomponents, flows, connections, modes** respectively. With our grammar it is easy to add more rules, for instance, we can easely add **calls, properties** to the component implementation declaration.

6.3 Semantics Actions

In this section we will present all the translations between AADL and PROVA that our semantics actions, that we add in our grammar, do.

First we focus only in the AADL declaration types and the respectives boilerplates that were generated.

Component category

In the next example you can see an example of a component declaration in AADL that will give an Entity in PROVA.

```
component_category name_of_component
end name_of_component ;
```

As you can see we have a *component_category* that can be, *device, process, processor, memory* and etc, and we also have the *name_of_component*, as we explain before this will give us only one entity in PROVA. The name of entity will

be the concatenation between the component category and the name of component, as explained before, doing this way it is more easy to an user distinguish the many components that we can have in PROVA.

Now we will show the boilerplate that is generated by the example:

– there is 1 *COMPONENT-CATEGORY_name-of-component*

Component features

Now we will show you the translation when we have *components* with *features*, as you can see in the next example and we will describe how to represent *components* with *features* in PROVA. A feature describes a functional interface of a component through which control and data may be exchanged with other components [1]

```
component_category name_of_component
  features
    feature_name : type_of_feature elements_identifier
end name_of_component ;
```

In the example you can see that we have declarated a component and inside of component we can add features, in the example we only added one. Now we describe how we can represent that in PROVA.

In the example we have a component, so we have to create an entity in PROVA. The reservated word *features* tell us that we have features inside the component, the features can be: list of ports or access to other elements. In Prova we can say that exists a relation between the component and those elements or ports. For an easy understanding, in PROVA, the name of the relation will be the concatenation between the reservated word *features* and its name *feature_name*, so this way, when you saw in PROVA a relation *FEATURES_feature-name* you know that the relation represent a feature of a component.

Now we will show the boilerplates that are generated by the example:

– there are 1 *COMPONENT-CATEGORY_name-of-component*
– every *COMPONENT-CATEGORY_name-of-component* shall have exactly 1 *FEATURES_feature-name TYPE-OF-FEATURE_elements-identifier*

Component flows

Now we will show you the translation when we have *components* with *flows*, as you can see in the next example, and we will describe how to represent *components* with *flows* in PROVA. Flow specifications can represent: flow sources that are flows originating from within a component; flow sinks that are flows ending within a component; flow paths that are flows through a component from its incoming ports to its outgoing ports[1].

```
component_category name_of_component
  flows
```

```
    flow_name : flow_spec flow_identifier1 ->
         flow_identifier2 -> flow_identifier3 ;
end name_of_component ;
```

The example show that we have flows inside a component, flows convert to relations between entities and we can assume that the extremities are the entities and the others are relations. Again to an easy understanding, in PROVA, the name of the relations will be the concatenation between the reservated word *flows* and its name *flow_name*, so this way, when you saw in PROVA a relation *FLOWS_flow-name* you know that the relation represent a flow of a component. We use the same algorithm to represent the entities that are flows, but instead of use *FLOWS* we use *FLOW*, so when you see an entitie that the name is *FLOW_identifier* we know that the *identifier* is an entitie that are a flow in AADL.

Now we will show the boilerplates that are generated by the example:

- there are 1 *COMPONENT-CATEGORY_name-of-component*
- every *COMPONENT-CATEGORY_name-of-component* shall have exactly 1 *FLOWS_flow-name flow-spec_flow-identifier1*
- every *FLOW-SPEC_flow-identifier1* shall have exactly 1 *FLOWS_flow-identifier2 FLOW_flow-identifier3*
- there are 1 *FLOW_flow-identifier1*
- there are 1 *FLOW_flow-identifier3*

Components that extends other Components

In AADL we can have components that extends other Components, these should be *Abstract* components. In the next example you can see an example of this

```
component_category name_of_component1
end name_of_component1 ;

component_category name_of_component2 extends name_of_component1
end name_of_component2;
```

The example show two components, the first(*name_of_component1*) should be a component of Abstract type, the second can have other type.

In Prova this can easily been represented, because in PROVA we also have Abstract Entities.

Now we will show the boilerplates that are generated by the example:

- there are 1 *ABSTRACT_name-of-component1*
- *COMPONENT-CATEGORY_name-of-component2* extends *ABSTRACT_name-of-component1*

Component Implementation

When we have component implementations in our model, first we have to declare a component and next we can or not declare the respective component implementation, a component implementation specifies an internal structure of the component, so, in PROVA we have to say that the component implementation *extends* the respective component. The next example show an example of a component implementation, note that the name should be the same, but in the implemetation we have to add *.impl*, the AADL community normally uses the word *impl* so this way they always know that component.*impl* represents the implementation of the component.

```
component_category name_of_component
end name_of_component1;

component_category implementation name_of_component.impl
end name_of_component.impl;
```

In the example you can see a component and the respective implementation, to distinguish a component and a component implementation, in the entity that represents the component implementation we add the keyword *IMPLEMENTATION*.

Now we will show the boilerplates that are generated by the example:

- there are 1 *COMPONENT-CATEGORY_name-of-component*
- there are 1 *COMPONENT-CATEGORY-IMPLEMENTATION_name-of-component.impl*
- *COMPONENT-CATEGORY-IMPLEMENTATION_name-of-component.impl* extends *COMPONENT-CATEGORY_name-of-component*

In component implementation we can have *subcomponents*, *modes*, *connections* and *flows*. The boilerplates that represents *Flows* are generated in the same way that we were generated when we had flows inside a component.

Component implementation with SubComponents

In AADL we can have internal components in component implementation, in other words, subcomponents. In Prova we can say that exists a relation between the component implementation and subcomponents. In the next example show how to declare subcomponents inside a component implementation.

```
component_category implementation name_of_component.impl
  subcomponents
    subcomponent_name : component_category identifier ;
end name_of_component.impl;
```

As you can see, we have a reservated word *subcomponents*, that means the existence of subcomponents inside a component implementation. And we have to say that exists a relation between the component implementation and the subcomponent, again the name of the relation will be the concatenation between

the reservated word *subcomponents* and its name, so, this way when you have a relation that starts with *SUBCOMPONENTS* you know that is a relation between a component implementation and a subcomponent.

No we will show the boilerplates that are generated by the example:

- there are 1 *COMPONENT-CATEGORY-IMPLEMENTATION_name-of-component.impl*
- every *COMPONENT-CATEGORY-IMPLEMENTATION_name-of-component.impl* shall have exactly 1 *SUBCOMPONENTS_subcomponent-name* *COMPONENT-CATEGORY_identifier*

Component implementation with Modes

In AADL we can have modes inside a component implementation, that represents operation states of software, execution platform and compositional components in the model physical system. The next example show how to declare modes inside a component implementation[1].

```
component_category implementation name_of_component.impl
  modes
    mode_name : mode_type identifier ;
end name_of_component.impl;
```

As you can see, we have a reservated word *modes*, that means the existence of modes inside a component implementation. In Prova we need to say that exists a relation between the component and the mode, again to represent this relation, the name will be the concatenation between the reservated word *modes* and its name, again, in PROVA when you see a relation that starts with *MODES* you know that is a relation between a component implementation and a subcomponent.

No we will show the boilerplates that are generated by the example:

- there are 1 *COMPONENT-CATEGORY-IMPLEMENTATION_name-of-component.impl*
- every *COMPONENT-CATEGORY-IMPLEMENTATION_name-of-component.impl* shall have exactly 1 *MODES_mode-name* *MODE-TYPE_identifier*

Component implementation with Connections

In AADL we can have connections inside a component implementation, that specify patterns of control and data flow between individual components at runtime[1]. In the next example show how to declare connections inside a component implementation.

```
component_category implementation name_of_component.impl
  connections
    connection_name : source -> destination ;
end name_of_component.impl;
```

In AADL the connections are directional, and have a source and a destination, as you can notest, there is a ternary relation, but in PROVA we only have binary relations, in prova we represent a ternary relation in two binary relations, for instace if we have *A-¿B-¿C* in PROVA we will have *A-¿B* and *B-¿C*.

Now we will show the boilerplates that are generated by the example:

- there are 1 *COMPONENT-CATEGORY-IMPLEMENTATION_name-of-component.impl*
- every *COMPONENT-CATEGORY-IMPLEMENTATION_name-of-component.impl* shall have exactly 1 *CONNECTIONS_connection-name CONNECTION_source*
- every *CONNECTION_source* shall have exactly 1 *CONNECTIONS_connection-name1 CONNECTION_source*

Notest that in the second relation we add the number 1 to the name of the relation, becase we can not have relations with same name.

7 Conclusion and Future Work

As future work we can improve our tool to support all the static parts of AADL and we also can implement a possible implementation of behaviour annex in PROVA.

In conclusion AADL is a very powerful language to describe architectures. Like Alloy, PROVA allow us to predict behaviour and early identification of errors, we had some difficulties in the analyse of the boilerplates syntax but thanks to our external supervisor, that help us when we needed, we could understood the boilerplates syntax and use them in our tool.

References

1. The SAE Architecture Analysis and Design Language Standard : A Language Summary, http://www.aadl.info/aadl/downloads/papers/AADLLanguageSummary.pdf