A Templating System to Generate Provenance — Supplementary Material —

Luc Moreau, Belfrit Victor Batlajery, Trung Dong Huynh, Danius Michaelides, Heather Packer

1 Introduction

This document contains supplementary material for the PROV-TEMPLATE article.

- Section 2 presents a complete example of template, bindings, and expanded template.
- Section 3 focuses on techniques to generate bindings.
- Section 4 provides further insight on the quantitative evaluation.

Cross-references to the article are noted A-x to distinguish them from cross-references within this paper.

2 Example of Bindings and Expansion

Figure 1 lists a set of bindings designed for the template of Figure A-2 in the paper. Bindings are represented as a JSON dictionary under the `"var"` key, whereas the `"context"` key contains prefix declarations, in the form of a dictionary associating prefixes with namespace URIs. Variables are themselves appearing as keys in a dictionary, associating them with values.

The serialization of the bindings (Figure 1) was inspired from the JSON-LD serialization [1]:

- `context`: The context provided to interpret the prefixes used in values.
- `@id`: The identifier of the current value; it is represented as a compact IRI[1] whose prefix is defined in the `@context` object.
- `@type`: The data type of the current value, also represented as a compact IRI.
- `@value`: The string serialization of the value as specified by the XSD schema [2].

Figure 2 displays the expanded template for the set of bindings displayed in Figure 1; this is a detailed version of the provenance shown in Figure A-5 in the paper.

3 Bindings Generation

The purpose of this section is to provide illustrations of the bindings generation techniques outlined in the article. To do so, first, we set the context by introducing the Provenance Challenge workflow which is used as a running example. Then, we discuss the various techniques to create bindings.

3.1 Provenance Challenge Workflow

To focus the discussion, we consider the Provenance Challenge workflow [3], which is usually regarded as a benchmark for the provenance community. It is displayed in Figure 3. The workflow consists of five stages, representatives of the kind of transformations involved in creating a brain atlas.

In this section, we will focus on the first `align_warp` step, a process that takes four inputs, an image and its header, and a reference image and a header, and it results in a parameter file, to be used by the next stage.

- *The authors are with the Department of Electronics and Computer Science, University of Southampton, SO17 1BJ, Southampton, UK. E-mail:* `l.moreau@ecs.soton.ac.uk`

Manuscript revised January 23, 2017

1. Internationalized Resource Identifier, see https://www.w3.org/TR/json-ld/#compact-iris.

```
{
    "var": {
        "endtime": [ { "@type": "xsd:dateTime",
                       "@value": "2016-03-08T14:21:12.085844" } ],
        "literal_value": [ "True", "logic_negate/NOT", "False" ],
        "literal_type": [ { "@id": "xsd:boolean" },
                          { "@id": "xsd:string" },
                          { "@id": "xsd:boolean" } ],
        "produced_at": [ { "@type": "xsd:dateTime",
                           "@value": "2016-03-08T14:21:12.085819" } ],
        "consumed_at": [ { "@type": "xsd:dateTime",
                           "@value": "2016-03-08T14:21:12.085706" },
                         { "@type": "xsd:dateTime",
                           "@value": "2016-03-08T14:21:12.085706" } ],
        "produced": [ { "@id": "urn_uuid:0266d372-e539-11e5-b38b-54bef7084653" } ],
        "produced_name": [ "__return__" ],
        "block_instance": [ { "@id": "urn_uuid:0266c6c0-e539-11e5-b38b-54bef7084653" } ],
        "block_uri": [ "506" ],
        "starttime": [ { "@type": "xsd:dateTime",
                         "@value": "2016-03-08T14:21:12.085586" } ],
        "agent": [ { "@id": "estatwf:John" } ],
        "parent": [ { "@id": "urn_uuid:0266c558-e539-11e5-b38b-54bef7084653" } ],
        "consumed": [ { "@id": "urn_uuid:fe1bf93c-e538-11e5-b38b-54bef7084653" },
                      { "@id": "urn_uuid:0266d11a-e539-11e5-b38b-54bef7084653" } ],
        "literal": [ { "@id": "urn_uuid:fe1bf93c-e538-11e5-b38b-54bef7084653" },
                     { "@id": "urn_uuid:0266d11a-e539-11e5-b38b-54bef7084653" },
                     { "@id": "urn_uuid:0266d372-e539-11e5-b38b-54bef7084653" } ],
        "consumed_name": [ "arg0", "fn" ],
        "block_title": [ "BuiltinFunction" ],
        "block_type": [ { "@id": "estatwf:BuiltinFunction" } ]
    },
    "context": {
        "xsd": "http://www.w3.org/2001/XMLSchema#",
        "estatwf": "http://purl.org/net/statjr/wf#",
        "urn_uuid": "urn:uuid:"
    }
}
```


Fig. 1. Set of bindings (detailed version of Figure A-3 in article)

3.2 Baseline: Handcrafted Code For Provenance Generation

The provenance related to the first `align_warp` step is displayed in Figure 4. We see two activities, corresponding to the overall Provenance Challenge workflow and `align_warp1`. The generated entity (`warp1.warp`) is derived from all the inputs. Inputs and the output are all files.

Figure 5 displays the kind of Java code that a programmer would typically write to create the provenance for `align_warp1`. The code involves a method call for each node and for each edge of the graph. Auxiliary methods are used to facilitate the generation of attributes, e.g., for types and labels.

The resulting provenance, expressed using the PROV-N notation is displayed in Figure 6. It corresponds to the visual representation of Figure 4.

The method `align` defined in Figure 5 is invoked using the arguments displayed in Figure 7, to create the graph illustrated in Figure 6.

```
document
  bundle urn_uuid:f6d0b777-8028-4071-b76a-3addedbb4764
    prefix estat <http://purl.org/net/statjr/ns#>
    prefix estatwf <http://purl.org/net/statjr/wf#>
    prefix urn_uuid <urn:uuid:>
    activity(urn_uuid:0266c558-e539-11e5-b38b-54bef7084653,-,-)
    activity(urn_uuid:0266c6c0-e539-11e5-b38b-54bef7084653,
             2016-03-08T14:21:12.085Z,2016-03-08T14:21:12.085Z,
             [prov:label = "BuiltinFunction",estatwf:block = "506" %% xsd:string])
    entity(urn_uuid:0266d11a-e539-11e5-b38b-54bef7084653)
    wasDerivedFrom(urn_uuid:0266d372-e539-11e5-b38b-54bef7084653,
                   urn_uuid:0266d11a-e539-11e5-b38b-54bef7084653)
    entity(urn_uuid:0266d372-e539-11e5-b38b-54bef7084653)
    wasStartedBy(urn_uuid:0266c6c0-e539-11e5-b38b-54bef7084653,-,
                 urn_uuid:0266c558-e539-11e5-b38b-54bef7084653,
                 2016-03-08T14:21:12.085Z)
    entity(urn_uuid:fe1bf93c-e538-11e5-b38b-54bef7084653)
    wasGeneratedBy(urn_uuid:0266d372-e539-11e5-b38b-54bef7084653,
                   urn_uuid:0266c6c0-e539-11e5-b38b-54bef7084653,
                   2016-03-08T14:21:12.085Z,[estat:bindingname = "__return__" %% xsd:string])
    wasDerivedFrom(urn_uuid:0266d372-e539-11e5-b38b-54bef7084653,
                   urn_uuid:fe1bf93c-e538-11e5-b38b-54bef7084653)
    used(urn_uuid:0266c6c0-e539-11e5-b38b-54bef7084653,
         urn_uuid:fe1bf93c-e538-11e5-b38b-54bef7084653,
         2016-03-08T14:21:12.085Z,[estat:bindingname = "arg0" %% xsd:string])
    used(urn_uuid:0266c6c0-e539-11e5-b38b-54bef7084653,
         urn_uuid:0266d11a-e539-11e5-b38b-54bef7084653,
         2016-03-08T14:21:12.085Z,[estat:bindingname = "fn" %% xsd:string])
    wasAssociatedWith(urn_uuid:0266c558-e539-11e5-b38b-54bef7084653,estatwf:John,-)
    agent(estatwf:John)
  endBundle
endDocument
```

Fig. 2. Expanded provenance (detailed version of Figure A-5 in article)

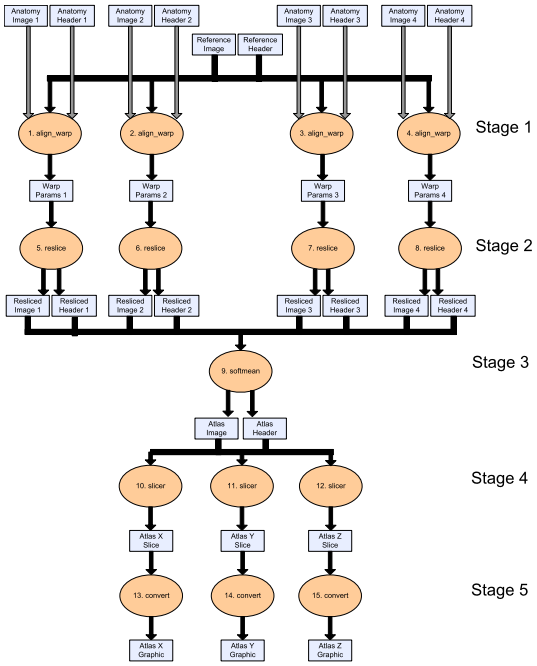

Fig. 3. The Provenance Challenge Workflow [3], illustrating steps involved in creating a brain atlas

3.3 Handcrafted Code for Bindings Construction

Instead of handcrafting code to generate provenance, the programmer can use PROV-TEMPLATE, and handcraft code that generates a set of bindings, suitable for expansion of the template provided in Figure A-1 of the paper. An example of such code is displayed in Figure 8.

The method `align` (in Figure 8) has the same signature as that of Figure 5 (up to the returned type), and therefore can be invoked as in Figure 7. The method creates bindings for the file identifiers, their labels, the activities and the agent.

While the number of lines in Figure 5 is marginally lower than in Figure 8, the logic is more complex, since only

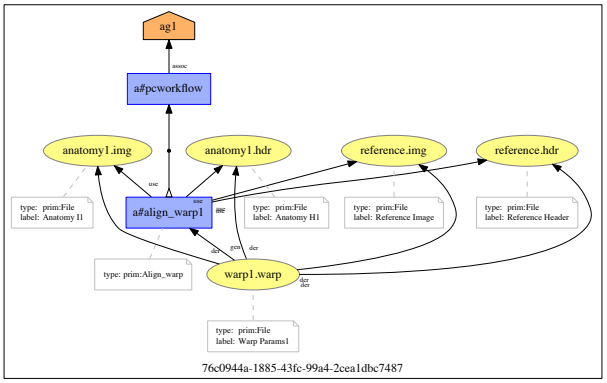

Fig. 4. Provenance pertaining to the first `align_warp` invocation

```
1     public Collection<StatementOrBundle> align(String imgfile,    String imglabel,
                                                  String hdrfile,    String hdrlabel,
                                                  String imgreffile, String imgreflabel,
                                                  String hdrreffile, String hdrreflabel,
                                                  String activity,
6                                                 String warpfile,   String warplabel,
                                                  String workflow,   String agent) {

        Collection<StatementOrBundle> ll=new LinkedList<StatementOrBundle>();
        Activity a1 = pFactory.newActivity(q(activity));                    // activity
11      pFactory.addType(a1, prim(ALIGN_WARP), name.PROV_QUALIFIED_NAME);
        Entity e1 = newFile(pFactory, imgreffile, imgreflabel);             // entities
        Entity e2 = newFile(pFactory, hdrreffile, hdrreflabel);
        Entity e3 = newFile(pFactory, imgfile,    imglabel);
        Entity e4 = newFile(pFactory, hdrfile,    hdrlabel);
16      Entity e5 = newFile(pFactory, warpfile,   warplabel);
        ll.addAll(Arrays.asList(a1,e1,e2,e3,e4,e5));
        ll.add(newUsed(a1, ROLE_IMG, e3));                                  // usages
        ll.add(newUsed(a1, ROLE_HDR, e4));
        ll.add(newUsed(a1, ROLE_IMG_REF, e1));
21      ll.add(newUsed(a1, ROLE_HDR_REF, e2));
        ll.add(newWasGeneratedBy(e5, ROLE_OUT, a1));                        // generation
        ll.add(newWasDerivedFrom(e5, e1));                                  // derivations
        ll.add(newWasDerivedFrom(e5, e2));
        ll.add(newWasDerivedFrom(e5, e3));
26      ll.add(newWasDerivedFrom(e5, e4));
        ll.add(newAgent(q(agent)));                                         // agent
        ll.add(newActivity(q(workflow)));
        ll.add(newWasAssociatedWith(q(workflow), q(agent)));
        ll.add(newWasStartedBy(q(activity), q(workflow)));
31      return ll;
    }
```

Fig. 5. Handcrafted Java code to generate provenance for the Provenance Challenge's `align` activity

bindings constructs are used in the latter, whereas a graph structure needs to be suitably constructed in the former, which requires dealing with nodes and edges, their direction, and their attributes. Thus, the maintenance effort and knowledge required is much higher for Figure 5. Furthermore, we note that lines 26–36 of code in Figure 8 bind variables to constants. We will see a further technique to avoid having to write such lines of code (see Section 3.4).

A source of errors in Figure 8 comes from the fact that template variable names need to be identified. It could be easy for a programmer to assign a value to the wrong variable. To avoid this problem, Section 3.5 discusses a technique in which bindings constructors are generated automatically.

```
document
   bundle uuid:3df1ef3b-aa2a-452d-99e4-4fb7fabb58e9
     prefix prim <http://openprovenance.org/primitives#>
     prefix pc1 <http://www.ipaw.info/challenge/>

     activity(pc1:a#align_warp1,-,-,[prov:type = 'prim:Align_warp'])
     activity(pc1:a#pcworkflow,-,-)
     agent(pc1:ag1)
     wasAssociatedWith(pc1:a#pcworkflow,pc1:ag1,-)
     wasStartedBy(pc1:a#align_warp1,-,pc1:a#pcworkflow,-)
     entity(pc1:anatomy1.img,[prov:type = 'prim:File', prov:label = "Anatomy I1"])
     entity(pc1:anatomy1.hdr,[prov:type = 'prim:File', prov:label = "Anatomy H1"])
     entity(pc1:reference.img,[prov:type = 'prim:File', prov:label = "Reference Image"])
     entity(pc1:reference.hdr,[prov:type = 'prim:File', prov:label = "Reference Header"])
     used(pc1:a#align_warp1,pc1:anatomy1.img,-,[prov:role = 'prim:Img'])
     used(pc1:a#align_warp1,pc1:anatomy1.hdr,-,[prov:role = 'prim:Hdr'])
     used(pc1:a#align_warp1,pc1:reference.img,-,[prov:role = 'prim:ImgRef'])
     used(pc1:a#align_warp1,pc1:reference.hdr,-,[prov:role = 'prim:HdrRef'])
     entity(pc1:warp1.warp,[prov:type = 'prim:File', prov:label = "Warp Params1"])
     wasGeneratedBy(pc1:warp1.warp,pc1:a#align_warp1,-,[prov:role = 'prim:Out'])
     wasDerivedFrom(pc1:warp1.warp, pc1:anatomy1.img)
     wasDerivedFrom(pc1:warp1.warp, pc1:anatomy1.hdr)
     wasDerivedFrom(pc1:warp1.warp, pc1:reference.img)
     wasDerivedFrom(pc1:warp1.warp, pc1:reference.hdr)
   endBundle
endDocument
```

Fig. 6. Generated Provenance for `align_warp1`

```
align("anatomy1.img",  "Anatomy I1",
      "anatomy1.hdr",  "Anatomy H1",
      "reference.img", "Reference Image",
      "reference.hdr", "Reference Header",
      "a#align_warp1",
      "warp1.warp",    "Warp Params1",
      "a#pcworkflow",  "ag1"));
```

Fig. 7. Invocation of the Method `align` with Concrete Values

3.4 Customization of Templates

A developer developing templates for an application will be faced with a tension between the genericity of templates and the effort of creating bindings to instantiate such templates. The more general a template, the more frequent the situation where it can be instantiated, but also the more variables the template contains, and the more complex bindings potentially are going to be. The less generic the template, the easiest the bindings become, but the more templates are required for a given system.

For instance, the template of Figure A-1 is generic, allows for any activity type, any number of consumed entities, and any number of produced entities, each with their respective roles, labels, and types. It is the only template used by the ebook application, but it comes with a large number of variables, some of which allowed to have multiple values: for instance, the `align_warp` uses 4 entities, and generates a single one.

PROV-TEMPLATE allows for templates to be customized, in effect by partially instantiating them. Such a customization allows for arity to be specified (number of used or generated artifacts) and some constant values to be provided (e.g., type of an entity or activity). Potentially, some variables may be dropped. To do this, one simply applies the template expansion algorithm with the permissive mode in which unbound variables are kept in the expanded template; alternatively one applies the expansion algorithm with bindings that associate variables with values (when a constant is to be specified) or with a variable (when a value is still expected): this latter approach is adopted here, since it allows us to fix the arity of variables.

For instance, by expanding the generic template of Figure A-2 in the paper with the bindings of Figure 9, we obtain the template of Figure 10: there we see that four entities are used by an activity of type `Align_warp`, and their respective types are instantiated.

```
    public Collection<Bindings> align(String imgfile1,    String imglabel,
                                      String hdrfile1,    String hdrlabel,
                                      String imgreffile1, String imgreflabel,
                                      String hdrreffile1, String hdrreflabel,
                                      String activity,
                                      String warpfile,    String warplabel,
                                      String workflow,    String agent)  {

        Bindings bindings1=new Bindings(pFactory);

        bindings1.addVariable(VAR_CONSUMED,         pc(imgfile1));
        bindings1.addVariable(VAR_CONSUMED,         pc(hdrfile1));
        bindings1.addVariable(VAR_CONSUMED,         pc(imgreffile1));
        bindings1.addVariable(VAR_CONSUMED,         pc(hdrreffile1));
        bindings1.addAttribute(VAR_CONSUMED_LABEL, imglabel);
        bindings1.addAttribute(VAR_CONSUMED_LABEL, hdrlabel);
        bindings1.addAttribute(VAR_CONSUMED_LABEL, imgreflabel);
        bindings1.addAttribute(VAR_CONSUMED_LABEL, hdrreflabel);
        bindings1.addVariable(VAR_BLOCK_INSTANCE,  pc(activity));
        bindings1.addVariable(VAR_PRODUCED,         pc(warpfile));
        bindings1.addAttribute(VAR_PRODUCED_LABEL, warplabel);
        bindings1.addVariable(VAR_PARENT,           pc(workflow));
        bindings1.addVariable(VAR_AGENT,            pc(agent));

        // and some bindings to constants
        bindings1.addAttribute(VAR_CONSUMED_NAME,  prim(ROLE_IMG));
        bindings1.addAttribute(VAR_CONSUMED_NAME,  prim(ROLE_HDR));
        bindings1.addAttribute(VAR_CONSUMED_NAME,  prim(ROLE_IMG_REF));
        bindings1.addAttribute(VAR_CONSUMED_NAME,  prim(ROLE_HDR_REF));
        bindings1.addAttribute(VAR_CONSUMED_TYPE,  prim(FILE));
        bindings1.addAttribute(VAR_CONSUMED_TYPE,  prim(FILE));
        bindings1.addAttribute(VAR_CONSUMED_TYPE,  prim(FILE));
        bindings1.addAttribute(VAR_CONSUMED_TYPE,  prim(FILE));
        bindings1.addAttribute(VAR_BLOCK_TYPE,     prim(ALIGN_WARP));
        bindings1.addAttribute(VAR_PRODUCED_TYPE,  prim(FILE));
        bindings1.addAttribute(VAR_PRODUCED_NAME,  prim(ROLE_OUT));

        return Collections.singleton(bindings1);
    }
```

Fig. 8. Handcrafted Code To Create Bindings for Provenance Challenge `align` Activity

3.5 Bindings Bean Generation

From a software engineering viewpoint, it is challenging to keep the handcrafted bindings of Section 3.3 synchronized with the templates. Indeed, as template variables appear in the program as constant strings, if the programmer edits the template, for instance, by renaming a variable, the handcrafted bindings will not automatically be changed, and template expansion will result in incomplete provenance (or will fail), because no binding is found for the renamed variable.

To address this problem, a solution is to use a template in order to generate automatically the code to create sets of bindings for that template. Figure 11 illustrates a so-called *bindings bean* automatically generated for the template `align_warp` of Figure 10. For every variable in the template, there is a method in the bindings bean. For instance, for `var:agent` in the template, we find `addAgent` in the bindings bean class `AlignBindingsBean`.

Equipped with such a bindings bean, the programmer calls the bean constructor to construct a set of bindings: Figure 12 illustrates how a set of bindings can be constructed for the `align_warp` template. Using the method `align` (with the same signature as the method in Figure 8, up to the returned result), one can create an instance of the `AlignBindingsBean`, and populate it with values for the relevant variables.

Figure 12 shows that the methods generated automatically from the templates are being called to construct the sets of bindings. For instance, we see a call to `addAgent` passing the agent's qualified name. If a template is changed by renaming a variable or dropping a variable, the bindings bean class can be recompiled; at compile-time, the Java compiler will detect the invocation of a method that no longer exists.

Further optimisations are possible. For instance, the bindings bean could detect if some variable is unbound before the set of bindings is passed to the template expansion algorithm. Such a technique of generation of bindings beans from templates helps with the maintenance of applications when templates change, as discussed in the paper in Section A-7.3,

Figure 13 displays the JSON serialization of the set of bindings constructed by the code of Figure 12 after invocation displayed in Figure 7.

```
{ "var" : {
    "b" :              [ { "@id" : "vargen:b" } ],
    "agent" :          [ { "@id" : "var:agent" } ],
    "block_instance" : [ { "@id" : "var:block_instance" } ],
    "block_type" :     [ { "@id" : "prim:Align_warp"  } ] ,
    "parent" :         [ { "@id" : "var:parent" } ],
    "produced_label" : [ { "@id" : "var:produced_label"  } ],
    "produced_type" :  [ { "@id" : "prim:File" } ],
    "produced" :       [ { "@id" : "var:produced" } ],
    "produced_at" :    [ { "@id" : "var:produced_at" } ],
    "produced_name" :  [ { "@id" : "prim:Out" } ],
    "consumed" :       [ { "@id" : "var:consumed1" },
                         { "@id" : "var:consumed2" },
                         { "@id" : "var:consumed3" },
                         { "@id" : "var:consumed4" } ],
    "consumed_type" :  [ { "@id" : "prim:File" },
                         { "@id" : "prim:File" },
                         { "@id" : "prim:File" },
                         { "@id" : "prim:File" } ],
    "consumed_label" : [ { "@id" : "var:consumed_label1" },
                         { "@id" : "var:consumed_label2" },
                         { "@id" : "var:consumed_label3" },
                         { "@id" : "var:consumed_label4" } ],
    "consumed_name" :  [ { "@id" : "prim:Img" },
                         { "@id" : "prim:Hdr" },
                         { "@id" : "prim:ImgRef" },
                         { "@id" : "prim:HdrRef" } ],
    "consumed_at" :    [ { "@id" : "var:consumed_at1" },
                         { "@id" : "var:consumed_at2" },
                         { "@id" : "var:consumed_at3" },
                         { "@id" : "var:consumed_at4" } ]  },
    "context" : {
        "prim"  : "http://openprovenance.org/primitives#",
        "pc1"   : "http://www.ipaw.info/challenge/",
        "var"   :  "http://openprovenance.org/var#",
        "vargen :  "http://openprovenance.org/vargen#" } }
```

Fig. 9. Set of bindings for customization of template for align_warp

```
document
  bundle vargen:b
    prefix prim <http://openprovenance.org/primitives#>
    prefix tmpl <http://openprovenance.org/tmpl#>
    prefix var <http://openprovenance.org/var#>
    prefix vargen <http://openprovenance.org/vargen#>

    activity(var:block_instance,-,-,[prov:type = 'prim:Align_warp'])
    activity(var:parent,-,-)
    agent(var:agent)
    wasAssociatedWith(var:parent,var:agent,-)
    wasStartedBy(var:block_instance,-,var:parent,-)
    entity(var:consumed1,[prov:type = 'prim:File', tmpl:label = 'var:consumed_label1'])
    entity(var:consumed2,[prov:type = 'prim:File', tmpl:label = 'var:consumed_label2'])
    entity(var:consumed3,[prov:type = 'prim:File', tmpl:label = 'var:consumed_label3'])
    entity(var:consumed4,[prov:type = 'prim:File', tmpl:label = 'var:consumed_label4'])
    used(var:block_instance,var:consumed1,-,[prov:role = 'prim:Img', tmpl:time = 'var:consumed_at1'])
    used(var:block_instance,var:consumed2,-,[prov:role = 'prim:Hdr', tmpl:time = 'var:consumed_at2'])
    used(var:block_instance,var:consumed3,-,[prov:role = 'prim:ImgRef', tmpl:time = 'var:consumed_at3'])
    used(var:block_instance,var:consumed4,-,[prov:role = 'prim:HdrRef', tmpl:time = 'var:consumed_at4'])
    entity(var:produced,[prov:type = 'prim:File', tmpl:label = 'var:produced_label'])
    wasGeneratedBy(var:produced,var:block_instance,-,[prov:role = 'prim:Out', tmpl:time = 'var:produced_at'])
    wasDerivedFrom(var:produced, var:consumed1)
    wasDerivedFrom(var:produced, var:consumed2)
    wasDerivedFrom(var:produced, var:consumed3)
    wasDerivedFrom(var:produced, var:consumed4)
  endBundle
endDocument
```

Fig. 10. Customized template for align_warp

```
// Generated Automatically by ProvToolbox for template "Align"
package org.example;

import java.lang.String;
import org.openprovenance.prov.model.ProvFactory;
import org.openprovenance.prov.model.QualifiedName;
import org.openprovenance.prov.template.Bindings;
import org.openprovenance.prov.template.BindingsBean;

public class AlignBindingsBean implements BindingsBean {
  private final Bindings bindings;

  private final ProvFactory pf;

  public AlignBindingsBean(ProvFactory pf) {
    this.pf = pf;
    this.bindings = new Bindings(pf);
  }

  public Bindings getBindings() {
    return bindings;
  }

  public void addBlockInstance(QualifiedName arg) {
    bindings.addVariable("block_instance",arg);
  }

  public void addAgent(QualifiedName arg) {
    bindings.addVariable("agent",arg);
  }

  public void addParent(QualifiedName arg) {
    bindings.addVariable("parent",arg);
  }

  public void addProduced(QualifiedName arg) {
    bindings.addVariable("produced",arg);
  }

  public void addConsumed1(QualifiedName arg) {
    bindings.addVariable("consumed1",arg);
  }

  public void addConsumed2(QualifiedName arg) {
    bindings.addVariable("consumed2",arg);
  }

  // code removed for clarity of presentation
  // ...
}
```

Fig. 11. Automatically generated bindings bean for `align_warp`

3.6 Architectural Overview

Figure 14 provides an overview of how we can generate provenance for the Provenance Challenge workflow. We started from the block template of Figures A-1 and A-2 in the paper. We then generated a new template for each workflow step by customizing the block template, with sets of bindings similar to that of Figure 9, making explicit the number of entities used and generated by activities, the types of activities and entities, and other attributes. Each workflow step template was then used to generate bindings beans automatically. Such bindings beans were constructed by the overall workflow, and sets of bindings serialized, and used by the expansion algorithm to expand the corresponding templates, resulting in the overall provenance.

```
    public Collection<BindingsBean> align(String imgfile1,    String imglabel,
                                          String hdrfile1,    String hdrlabel,
                                          String imgreffile1, String imgreflabel,
                                          String hdrreffile1, String hdrreflabel,
                                          String activity,
                                          String warpfile,    String warplabel,
                                          String workflow,    String agent)  {

        AlignBindingsBean bean=new AlignBindingsBean(pFactory);

        bean.addConsumed1(pc(imgfile1));
        bean.addConsumed2(pc(hdrfile1));
        bean.addConsumed3(pc(imgreffile1));
        bean.addConsumed4(pc(hdrreffile1));

        bean.addConsumedLabel1(imglabel);
        bean.addConsumedLabel2(hdrlabel);
        bean.addConsumedLabel3(imgreflabel);
        bean.addConsumedLabel4(hdrreflabel);

        bean.addBlockInstance(pc(activity));

        bean.addProduced(pc(warpfile));
        bean.addProducedLabel(warplabel);

        bean.addParent(pc(workflow));
        bean.addAgent(pc(agent));

        return Collections.singleton((BindingsBean)bean);
    }
```

Fig. 12. Handcrafted code exploiting automatically-generated bindings bean

```
{ "var" : { "consumed1" :       [ { "@id" : "pc1:anatomy1.img" } ],
            "consumed2" :       [ { "@id" : "pc1:anatomy1.hdr"  } ],
            "consumed3" :       [ { "@id" : "pc1:reference.img" } ],
            "consumed4" :       [ { "@id" : "pc1:reference.hdr" } ],
            "consumed_label1" : [ "Anatomy I1" ],
            "consumed_label2" : [ "Anatomy H1" ],
            "consumed_label3" : [ "Reference Image" ],
            "consumed_label4" : [ "Reference Header" ],
            "block_instance" :  [ { "@id" : "pc1:a#align_warp1" } ],
            "produced" :        [ { "@id" : "pc1:warp1.warp" } ],
            "produced_label" :  [ "Warp Params1" ],
            "agent" :           [ { "@id" : "pc1:ag1" } ],
            "parent" :          [ { "@id" : "pc1:a#pcworkflow"  } ]  },
  "context" : { "pc1" : "http://www.ipaw.info/challenge/" } }
```

Fig. 13. Resulting set of bindings for `align1` activity

3.7 CSV to Sets of Bindings

It is not always possible to modify a legacy application to insert code that generates provenance. Sometimes, the only way to generate provenance is by accessing application data, and reconstructing provenance from it. To help provenance-enabling legacy applications, it is, therefore, useful to have some support for common data formats.

Application data is often available in, or exportable to, tabular format such as, for instance, the standardized "comma-separated value" (CSV) format [4]. It is rather straightforward to convert CSV to the representation of sets of bindings. Figure 15 shows a tabular representation of all values logged for `align_warp` activities, where the column headings are used to indicate the variables that the are being defined, and each line represents a possible set of bindings for such variables. By combining the column heading and the first line of the table, one can easily construct the set of bindings displayed in Figure 13.

3.8 Asynchronous Bindings Generation With Bindings Fragment

So far, the discussion has been mostly on what goes into sets of bindings and how this can be programmed, with minimal effort. An important question is also *when* such sets of bindings should be created. The ebook application contains a workflow interpreter that was augmented to log execution information in the bindings format [5]. The key moments for capturing information during the interpretation of a workflow are: at the beginning of a block evaluation, at the point its input arguments have been evaluated, at the point when outputs are generated, and finally at the end of block evaluation.

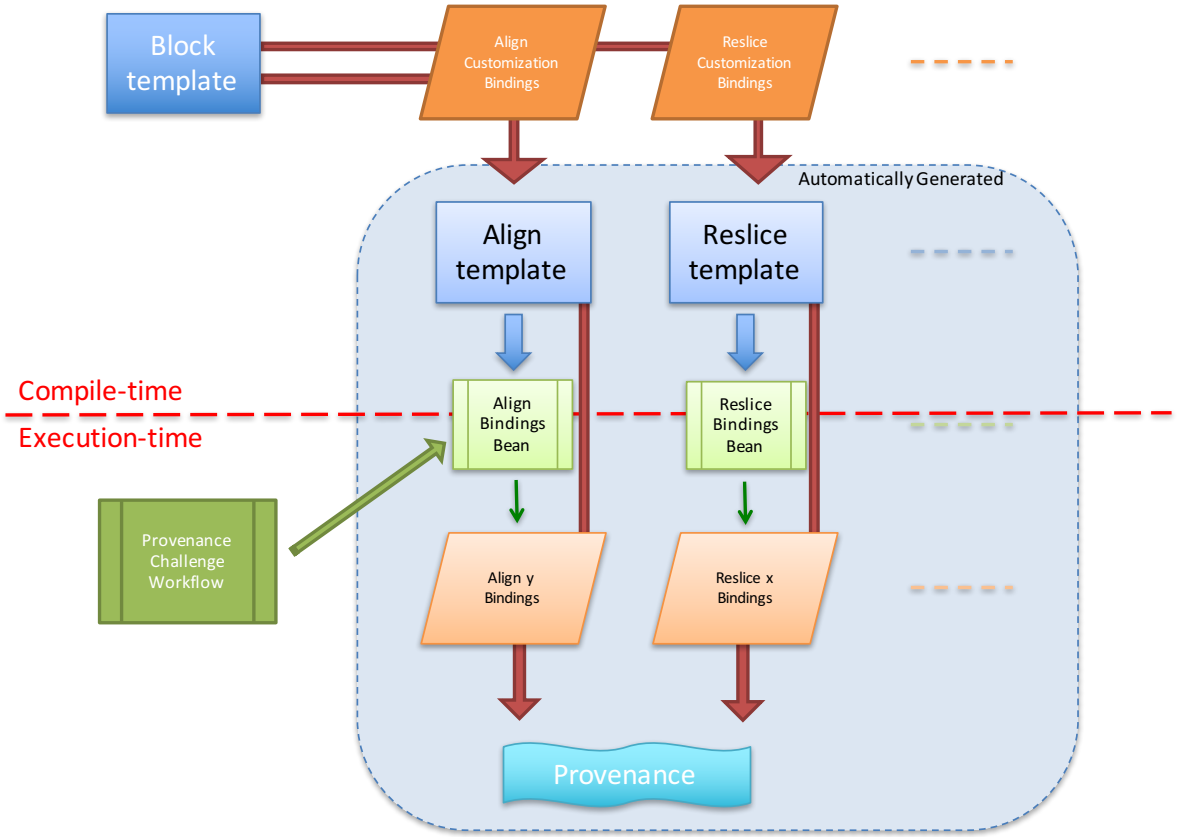

Fig. 14. Final architecture of the Provenance Challenge workflow. Darker boxes have to be generated by hand (whether templates, bindings or code), whereas lighter boxes are generated automatically (whether templates, bindings or code). Blue rectangles denote templates, green boxes denote code, and orange parallelograms represent sets of bindings.

block_instance	parent	consumed1	consumed_label1	consumed2	consumed_label2	...
pc1:a#align_warp1	pc1:a#pcworkflow	pc1:anatomy1.img	Anatomy I1	pc1:anatomy1.hdr	Anatomy H1	...
pc1:a#align_warp2	pc1:a#pcworkflow	pc1:anatomy2.img	Anatomy I2	pc1:anatomy2.hdr	Anatomy H2	...
pc1:a#align_warp3	pc1:a#pcworkflow	pc1:anatomy3.img	Anatomy I3	pc1:anatomy3.hdr	Anatomy H3	...
pc1:a#align_warp4	pc1:a#pcworkflow	pc1:anatomy4.img	Anatomy I4	pc1:anatomy4.hdr	Anatomy H4	...

Fig. 15. Tabular representation of sets of bindings

At each capture point, values for a subset of variables of the block template are captured; we call this a *bindings fragment*. The bindings fragment is appended to a log along with the type of recording: *begin*, *input*, *output* and *end*.

To generate the complete log after execution, we iterate over the bindings fragment log and use a stack to aid in combining the fragments. A *begin* fragment pushes a new bindings record onto the stack, filling in values of variables known at block start (i.e., the top 6 variables in Figure 16). This includes the parent variable which is the value of the `block_instance` variable in the next record on the stack. *Record input/output* appends values to the `consumed`, `produced` and `literal` variables as appropriate, including the `_at`,`_name`, `_value` and `_type` variables. *End* finalises a binding setting the `endtime` variable, pops the sets of bindings from the stack, and commits it to the log.

For performance reasons, we wanted to log bindings as quickly as possible and with as little impact on the runtime system as possible. When augmenting the application, code changes were kept to a minimum by using Python decorators. Logging with fragments limits the information that needs to maintained during runtime execution. It was important to the application that the evaluator could be easily paused and resumed and for the bindings to still be available.

```
('06265dea-96da-11e6-8d8c-54bef7084653',
   {'var:literal_type': [],
    'var:parent': UUID('06265c96-96da-11e6-8d8c-54bef7084653'),
    'var:literal': [],
    'var:produced_at': [],
    'var:produced_name': [],
    'var:consumed_name': [],
    'var:block_instance': UUID('06265dea-96da-11e6-8d8c-54bef7084653'),
    'var:consumed_at': [],
    'var:literal_value': [],
    'var:starttime': datetime.datetime(2016, 10, 20, 16, 29, 43, 228982),
    'var:consumed': [],
    'var:block_title': 'BinaryOperator',
    'var:produced': [],
    'var:block_uri': '9q5pww2cpx7v7lupzu9c',
    'var:endtime': None,
    'var:block_type': 'estatwf:BinaryOperator'}),

('06265dea-96da-11e6-8d8c-54bef7084653',
   {'var:literal_type': 'xsd:string',
    'var:literal': UUID('063102cc-96da-11e6-8d8c-54bef7084653'),
    'var:literal_value': 'ADD'}),

('06265dea-96da-11e6-8d8c-54bef7084653',
   {'var:consumed_name': 'operator',
    'var:consumed_at': datetime.datetime(2016, 10, 20, 16, 29, 43, 229018),
    'var:consumed': UUID('063102cc-96da-11e6-8d8c-54bef7084653')}),

('06265dea-96da-11e6-8d8c-54bef7084653',
   {'var:consumed_name': 'operandl',
    'var:consumed_at': datetime.datetime(2016, 10, 20, 16, 29, 43, 229018),
    'var:consumed': UUID('0625fa4e-96da-11e6-8d8c-54bef7084653')}),

('06265dea-96da-11e6-8d8c-54bef7084653',
   {'var:literal_type': 'xsd:integer',
    'var:literal': UUID('0625fa4e-96da-11e6-8d8c-54bef7084653'),
    'var:literal_value': '1'}),

('06265dea-96da-11e6-8d8c-54bef7084653',
   {'var:consumed_name': 'operandr',
    'var:consumed_at': datetime.datetime(2016, 10, 20, 16, 29, 43, 229018),
    'var:consumed': UUID('0625fcba-96da-11e6-8d8c-54bef7084653')}),

('06265dea-96da-11e6-8d8c-54bef7084653',
   {'var:literal_type': 'xsd:integer',
    'var:literal': UUID('0625fcba-96da-11e6-8d8c-54bef7084653'),
    'var:literal_value': '2'}),

('06265dea-96da-11e6-8d8c-54bef7084653',
   {'var:produced': UUID('06310a1a-96da-11e6-8d8c-54bef7084653'),
    'var:produced_name': '__return__',
    'var:produced_at': datetime.datetime(2016, 10, 20, 16, 29, 43, 229323)}),

('06265dea-96da-11e6-8d8c-54bef7084653',
   {'var:literal_type': 'xsd:integer',
    'var:literal': UUID('06310a1a-96da-11e6-8d8c-54bef7084653'),
    'var:literal_value': '3'}),

('06265dea-96da-11e6-8d8c-54bef7084653',
   {'var:endtime': datetime.datetime(2016, 10, 20, 16, 29, 43, 229366)}),
```

Fig. 16. Bindings Fragments

4 Quantitative Evaluation (Continued)

In this section, we provide further insight on the quantitative evaluation appearing in Section A-5 of the paper. Table A-4 in the paper provides a summary of expansion times — mean, standard deviation, median — and average bindings set size. The box plot of Figure 17 shows expansion times for each template, after normalizing the time with respect to the expansion time. Figure 17 also shows the median for each application; their value appears in Table 1.

TABLE 1
Normalized median (also displayed as horizontal lines in Figure 17)

application	normalized median (in ms/Kb)
smart	0.155
food	0.193
ebook	0.153
picaso	0.210
total	0.183

To get further insight into the expansion process, we can compare the size of a template and the size of its expansion. Figure 18 displays such a ratio for all the templates. Table 2 summarizes the ratios for the various applications.

TABLE 2
Summary of ratios (templates/expanded provenance)

application	mean	std. dev.	median
smart	0.751	0.227	0.817
food	0.797	0.419	0.857
ebook	0.706	0.150	0.714
picaso	0.580	0.218	0.547
total	0.657	0.274	0.680

Figure 18 displays some variability in the range of ratios for some templates. In ebook's `block_run`, the bindings with the lowest ratio (0.07) is for the same block as discussed in the paper, with the large number of outputs (31). The bindings with ratio (1.23) are for blocks which do not consume or produce any entity; in that case, the template has a large number of unused variables, making it larger than its expansion.

Likewise, in the food application, the templates `foodspec` for food specification and `analysis` for sampling analyis are general and contain a very large number of variables so that they can cater for all types of food and analyses, respectively. However, some bindings provide values for only a few of these variables, resulting in the expanded provenance being smaller than the templates.

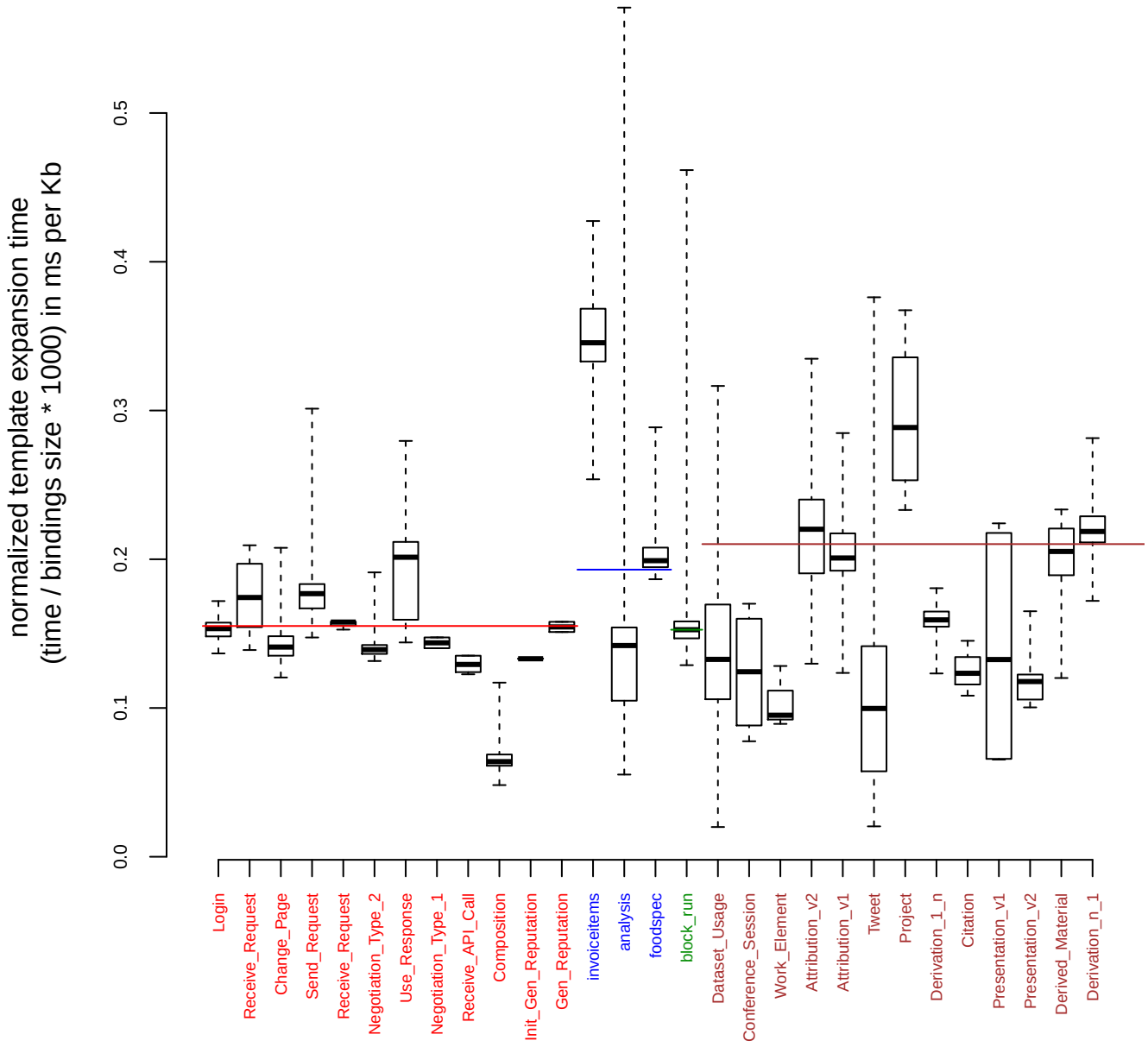

Fig. 17. Expansion time normalized by the length of sets of bindings

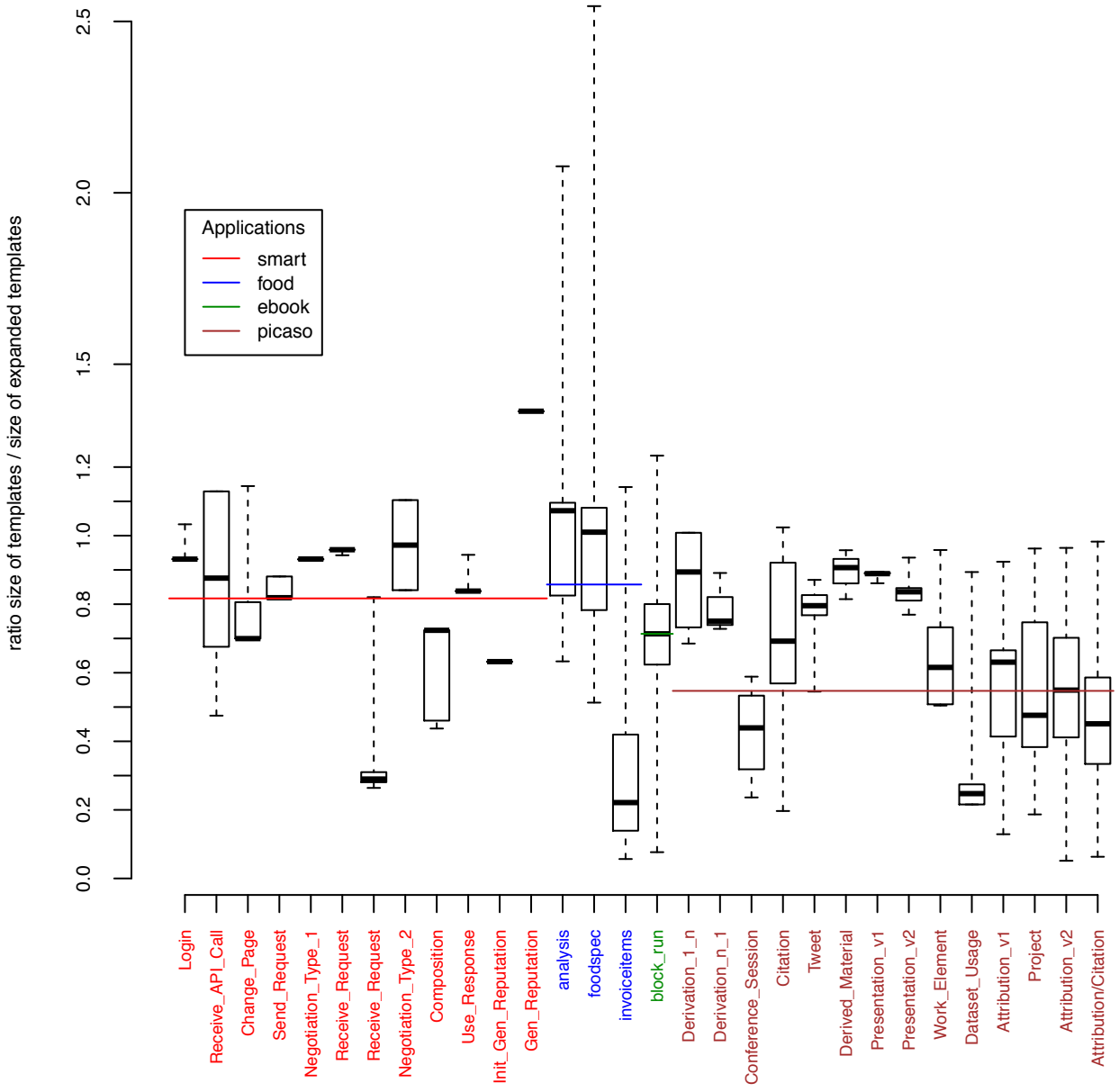

Fig. 18. Ratio template vs expanded provenance, per application, per template. The legend shows the color adopted for each application. Colored horizontal lines are the median compression ratio for each application.

REFERENCES

[1] M. Sporny, D. Longley, G. Kellogg, M. Lanthaler, and N. Lindström, "A JSON-based serialization for linked data," World Wide Web Consortium, W3C Recommendation, 2014. [Online]. Available: https://www.w3.org/TR/2014/REC-json-ld-20140116/

[2] D. Peterson, S. S. Gao, A. Malhotra, C. M. Sperberg-McQueen, and H. S. Thompson, "W3C XML schema definition language (XSD) 1.1 part 2: Datatypes," World Wide Web Consortium, W3C Recommendation, 2012. [Online]. Available: https://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/

[3] L. Moreau, B. Ludaescher, I. Altintas, R. S. Barga, S. Bowers, S. Callahan, G. Chin Jr., B. Clifford, S. Cohen, S. Cohen-Boulakia, S. Davidson, E. Deelman, L. Digiampietri, I. Foster, J. Freire, J. Frew, J. Futrelle, T. Gibson, Y. Gil, C. Goble, J. Golbeck, P. Groth, D. A. Holland, S. Jiang, J. Kim, D. Koop, A. Krenek, T. McPhillips, G. Mehta, S. Miles, D. Metzger, S. Munroe, J. Myers, B. Plale, N. Podhorszki, V. Ratnakar, E. Santos, C. Scheidegger, K. Schuchardt, M. Seltzer, Y. L. Simmhan, C. Silva, P. Slaughter, E. Stephan, R. Stevens, D. Turi, H. Vo, M. Wilde, J. Zhao, and Y. Zhao, "The First Provenance Challenge," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 409–418, Apr. 2008. [Online]. Available: http://www.ecs.soton.ac.uk/~lavm/papers/challenge-editorial.pdf

[4] J. Tennison and Gregg Kellogg (eds.), "Model for tabular data and metadata on the web," World Wide Web Consortium, Recommendation, 2015. [Online]. Available: https://www.w3.org/TR/2015/REC-tabular-data-model-20151217/

[5] D. Michaelides, R. Parker, C. Charlton, W. Browne, and L. Moreau, "Intermediate notation for provenance and workflow reproducibility," in *6th International Provenance and Annotation Workshop (IPAW'16)*, McLean, VA, US, Jun. 2016, pp. 1–12. [Online]. Available: http://eprints.soton.ac.uk/393117/