Towards a Well-Formed Software Architecture Analysis

Najd Altoyan
Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas
naltoyan@utexas.edu

Dewayne E. Perry
Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas
perry@ece.utexas.edu

ABSTRACT

Over the past decade Software Architecture has proved to be a core component in software development processes. Therefore, the correctness of the software architecture is unsurprisingly crucial for the success of software products. Many methods for analyzing software architecture have been proposed in the literature in order to predict potential risks that impose far-reaching effects on the final product. However, almost none of these methods have been adopted in the industry as many of them require special knowledge or are simply not intuitive enough for the software developer. In this paper, we address this issue by proposing (the first step of) a framework for analyzing software architecture that is built using a semi-formal architecture description language (ADL), and a constraint based relational model. Architecture instances following our approach are then analyzed against a set of properties such as whether the architecture is *complete*, *self-sufficient* and *self-contained*. Satisfying these properties yields a well-formed architecture.

CCS CONCEPTS

• **Software and its engineering → Software architectures**; **Consistency**; **Completeness**; **Automated static analysis**; **Architecture description languages**; *Software system models*; Specification languages;

KEYWORDS

Software architecture, Modelling, Analysis, Well-formed architecture, Alloy

ACM Reference format:
Najd Altoyan and Dewayne E. Perry. 2017. Towards a Well-Formed Software Architecture Analysis. In *Proceedings of ECSA '17, Canterbury, United Kingdom, September 11–15, 2017,* 7 pages.
https://doi.org/10.1145/3129790.3129813

1 INTRODUCTION

Software architecture analysis is one of the main issues within the field of software architecture. The aim of analyzing system's software architecture is to discover important system properties [25] and predict its quality [18] using the system's architectural models. Early evaluation and verification of the system, before it is built, give way for detecting potential risks and therefore minimize the overhead cost of the system.

Many researchers are interested in formally analyzing software architectures in terms of structure, behavior, quality and other architectural aspects. Formal analysis offers a rigorous way to characterize the structure and properties of software architectures. It has the advantage of providing precise analysis and hence ensuring the correctness and consistency of software systems. Furthermore, as systems grow larger and more complex, manual techniques were no longer sufficient, and the need for automating formal methods became more evident. Unfortunately, despite the significant work done in this area, languages and methods found in the literature have failed, to a great degree, to be utilized in practice [8]. One of the reasons is that formal languages used as basis for analysis (e.g. CSP, FSP, etc.) require a considerable amount of effort to learn plus time to produce a complete architectural description [23]. Therefore, practitioners have settled for informal languages since they are more intuitive and less costly to adopt, and hence were forced to perform manual analysis to avoid using unsatisfactory ADLs [21]. Nonetheless, they still appreciate the need for architectural analysis offered by formal notation and call for a way to merge informal methods with formal and analyzable ones [21]. Of course, since informal communication usually entails ambiguity and therefor is error-prone [21], we believe that using semi-formal notation is the best, and perhaps the only, way to proceed.

In this paper, we address this issue by proposing a framework for software architecture analysis based on a semi-formal model for specifying software architecture, developed by Bhattacharya and Perry [5], and a first-order logic model for formal analysis developed by Jackson and his colleagues [1]. As we advocate the need for architecture analysis methods for software developers and practitioners, these models seemed a natural fit (see Section 3).

In section 2 we give an overview of related work. Section 3 provides a brief introduction to the models used in our approach, namely the abstract ADL and Alloy. Section 4 presents an overview of our approach together with a set of analysis rules used in the framework. Section 5 demonstrates our work through an example. Section 6 gives a brief discussion on the cost and benefits of our approach. We conclude in Section 7 and consider future work.

2 RELATED WORK

Architecture description languages were developed as a way for specifying software architecture with different degrees of formalization. However, not all of them provide a means for analysis. Those that support automatic analysis vary in their focus. For example, ADLs like Rapide [19] and Wright [3] focus on analyzing the behavior of the architecture, whereas AADL [7] provides a basis for

ECSA '17, September 11–15, 2017, Canterbury, United Kingdom
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5217-8/17/09…$15.00
https://doi.org/10.1145/3129790.3129813

system analysis in terms of quality attributes. Most of these ADLs were developed in the 1990s and have influenced the creation of other recent ADLs such as CONNECT [13]. However, none of these (old or recent) gained popularity in the industry as they did not improve properties of concerned by industry such as usability, but instead focused on other issues that are in less of need, such as code generation [23] [21].

Other works on formal analysis, that are independent from ADLs, have been proposed in the literature in a variety of aspects. For instance, Magee et al. [20], Kramer et al. [17], and Inverardi and Tivoli [12] proposed different methods for verifying system behavior at an architectural level; Kazman et al. [15], He and Deng [11], and Bengtsson et al. [4] developed methods for analyzing system properties, SAAM, SAM, and ALMA respectively; Velasco-Elizondo et al. [26] focused on data analysis for detecting data-related mismatches (e.g. format mismatch); Ding and Zhang [6] showed how to analyze properties of a specific architectural style, namely the Publish-Subscribe style, using Petri Nets and Alloy; and many others. Closer to our approach are the works proposed by Kim and Garlan [16] and Mokni et al. [22]. Kim and Garlan proposed a general approach for analyzing architectural styles using Acme [9] as the description language and Alloy for formal analysis. However, they do not address the structure of architectures in general but only architectural styles and their related properties. Mokni et al. on the other hand focus on the analysis of architectural structure in terms of reuse rather than overall correctness. They use their own description language, Dedal, and employ B [2] as a means for formal analysis. Moreover, their work is inspired from type theory in object languages and is therefore type based.

Apart from the works listed above, we did not find in the recent literature any major work that is related to software architecture analysis. Most recent works are mainly focused on other aspects such as optimization, synthesis, etc. In our work, we propose a constraint based framework for analyzing software architecture. We initially focus on structure and ultimately plan to cover other aspects of analysis (such as non-functional properties). The novelty of our work is that we reconcile semi-formal techniques with analyzable first-order logic methods and accomplish this in a way that we believe to be practical and intuitive enough for industrial practice. Moreover, we view the architecture as a set of constraints rather than a set of types. For example, two architectural elements are said to be equivalent if we can imply one from the other through the set of constraints associated with them. Although this might not be clearly evident in this paper (since we are at the preliminary stage of our work), we paved the way for future work by carefully selecting the right basis; both models incorporated in our approach support constraint definition and analysis.

3 BACKGROUND

Our framework is based on two models: the Abstract Architectural Model (AAM) [5] and Alloy [1]. Although there are many ADLs that exist in the literature, the advantage of using AAM is twofold: 1) it uses a semi-formal notation, namely XML, as a means for writing architectural specification; and 2) it facilitates constraint-based evaluation by enabling the definition of architectural elements as a collection of constraints.

Alloy, on the other hand, is used for formal analysis. One of the reasons for choosing Alloy is that it defines a model as a set of constraints which fits with our approach. Moreover, the declarative nature of Alloy means that architectural instances can be incrementally analyzed; one can start by defining few constraints and later on add more as needed. Finally, unlike other formal specification languages (e.g. Z, B, VDM), Alloy allows fully automatic analysis [14].

3.1 Abstract ADL Model (AAM)

The abstract architectural model (based on [5]) is a semi-formal ADL that was designed to support various architectural needs but mainly focusing on component composition reasoning and constraint based evaluation. The model is consistent with the Perry and Wolf definition of software architecture [24] which, unlike most architecture models, is constraint based as opposed to type based. The model has three abstract architectural constructs: arch-element which represents both components and connectors (since they are structurally identical), arch-composition which represents the sub-architectural structure of an arch-element, and arch-region which represents an arbitrary collection of arch-elements and arch-compositions that adheres to, or are restricted to, a specific set of constraints.

An arch-element is defined by services, dependencies and constraints. Services are facilities provided by the element whereas dependencies represent supporting services needed from other architectural elements. Constraints define a set of functional and non-functional constraints that should be satisfied. These constraints are beyond the scope of this paper (refer to Section 7).

Arch-element = (name, {service specifications}, {dependency specifications}, {general constraints})

Both services and dependencies are eventually modelled in terms of constraints as well. Their specifications are further defined by a set of input and output constraints which captures constraints related to I/O data and I/O events, in addition to general specification constraints such as pre and post conditions.

Service specification = (name, Dependency specification)
Dependency specification = ({input constraints}, {output constraints}, {general specification constraints})

An arch-composition is mainly defined by a set of arch-elements and a set of mappings. The mapping maps a set of specifications (services or dependencies) to another set of specifications. This gives the flexibility of renaming (exported) services or combining multiple services in one. For example, if a given component has services M and N, then M may be exported as service P or the combination of services N and M may be exported as service Q. It also specifies the architectural dependencies within a composition.

Arch-composition = (name, {arch-elements}, {mappings})

For the purpose of this paper, we focus our attention on arch-elements and arch-compositions only. More details about the arch-region construct or the abstract model in general can be found at [5].

3.2 Alloy

Alloy [14], [1] is a declarative specification language used for describing a set of structures and constraints over these structures. It is based on first order relational logic and is automatically analyzable. Models written in Alloy can be checked for correctness using the *Alloy Analyzer*. However, since the analyzer is a bounded checker, it can only perform finite checks. As a result, Alloy cannot guarantee the correctness of infinite models but instead tries to find counter examples that contradict a set of constraints within restricted scope. The downside is that there may still be unfound counter examples that might be found if Alloy is given a larger scope.

All structures in Alloy are represented as relations. *Signatures* define sets of atoms and the relation between these sets is defined through *fields*. For example, consider the following model:

```
module Registration
    sig Student{
        courses: Course,
        gpa: Grade,
        dep: Department }

    sig Course{offeredby: Department}
    sig Department{}
    abstract sig Grade{}
    one sig A, B, C, D, F extends Grade{}

    //Courses taken by a student should be offered
    //by the student's department only
    fact{all s:Student | all c:s.courses |
        s.dep = c.offeredby}
    }
```

Here we have four signatures: Student, Course, Department and Grade. Grade is extended by five *singular* sets. Each student in the `Student` set has three relations: `courses` with the set `Course`, `gpa` with `Grade` and `dep` with `Department`. The last line is a fact, which should hold at all times, ensuring that courses taken by every student instance from the model are offered by the same department the student belongs to. Using facts is one way to define constraints. Now consider the following predicate which returns true when all students in the model have a GPA not equal to F. When Alloy is executed using the run statement, it searches for an instance that satisfies (the previous fact and) the predicate within a restricted scope - in our case the scope is set to be three.

```
  pred all_passed[]{all s:Student | s.gpa != F}
  run all_passed for 3
```

4 THE MODEL

As mentioned earlier, we use the abstract ADL in our work for the specification of software architecture. However, we slightly modify the terminology to allow multi-level/nested compositions and to distinguish between components and connectors for the purpose of logical analysis. So now, an arch-element can either be a basic arch-element or an arch-composition, whereas a basic arch-element can either be a component or a connector. An arch-composition is a collection of arch-elements.
Arch-element = basic arch-element | arch-composition
Basic arch-element = arch-component | arch-connector
Arch-component = arch-element interface
Arch-connector = arch-element interface
Arch-composition = (arch-element interface, {arch-elements}, {mappings})
Arch-element interface = (name, {service specifications}, {dependency specifications}, {general constraints})

Note that components and connectors are still defined in the same way. However, we decided to distinguish between them since they have distinct roles within an architecture and this distinction is needed in our analysis. To this end, our model is composed of the 1) *basic model* which is the Alloy representation of the abstract ADL, 2) a set of *analysis rules* needed for well-formed architecture analysis, and 3) architectural instances that are written using the abstract ADL notation and then translated to Alloy (by extending the basic model). Our approach is illustrated in Figure 1.

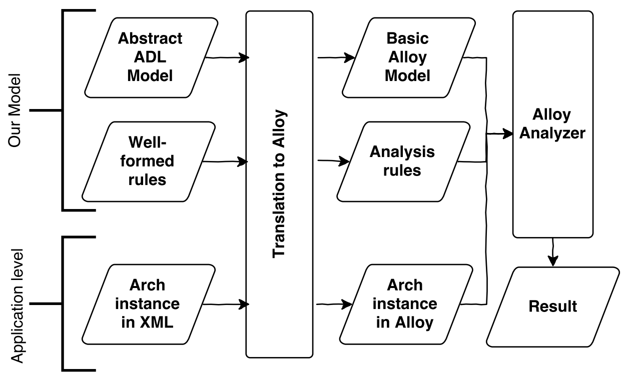


Figure 1: Overall analysis flowchart.

4.1 Basic Model

The basic model is an Alloy representation of the abstract ADL. It captures the notation of the ADL in an Alloy format. This process is very important in facilitating the creation of architectural instances later on. It is worth nothing that this is done only once. However, the basic model should be updated to reflect any changes made to the abstract ADL model.

Generally speaking, the abstract ADL is translated to an Alloy model by mapping entities from the ADL to Alloy signatures. For example, the Arch-element (from the ADL) is represented by an Element signature. The definitions given in the previous section (Section 3.1) are translated as follows:

```
abstract sig System{
    components: some Component,
    connectors: set Connector,
    versions: some Version}
//fact
{one i:Interface | versions.interface = i}

abstract sig Element {interface: one Interface}

abstract sig Composition extends Element{
    elements : some Element,
    mappingS: Service set -> set Service}

abstract sig BasicElement extends Element
```

```
{_system: System}

    abstract sig Component, Connector extends
BasicElement{}

    sig Version in Composition{ _system: System}

    sig Interface{
        services : some Service,
        dependencies: set Dependency,
        constraints: set GeneralConstraint,
        _element: some Element}
```

An abstract architecture has a set of components, connectors and versions. A version is a high-level composition that exports the interface of the system. All versions should share the same interface since the system should have only one interface, but the way an interface is fulfilled differs between versions. This allows providing the same set of services promised by a system with the flexibility of using different design approaches (compositions or implementations).

An architectural element can be either a composition or a basic element. This is represented by extending the Element signature. A composition has a set of arch-elements and a set of mappings. Recall that a version is a special case of a composition. Each architectural element has its own interface which defines a set of services provided by the element, dependencies required by the element and constraints that should hold for the element.

This is a partial translation of the ADL, and the rest should be carried out in a similar fashion. So far, this is just a model, and no concrete instances of architecture are created yet. This is discussed in a different section below (Section 4.3)

The basic model is accompanied with a set of general *basic rules* (or *facts* as per Alloy's terminology) that captures the common guidelines in building an architecture (e.g. a composition should not include itself).

4.1.1 Basic Rules. The set of rules governing the general structure of any given architecture is defined by means of Alloy facts. These include the following facts:

- Any architecture should have at least one component, at least one version and zero or more connectors. This means that the smallest architecture allowed in our model is a composition of one component that its interface is exported to the system.
- Connectors logically describe the needs of the elements in their interactions with other elements. The number of possibly distinct connectors depends on the how many components interact with each other and how that interaction is to be structured. However, there has to be at least one connector in any composition of two or more elements. Usually, a connector connects two components, but in some cases, one can have a highly intelligent connector that can coordinate the communication between all participating components.
- An interface belongs to one element but a specification (i.e. a service or a dependency) may belong to more than one element. This is especially essential to enforce multiple versions having the same interface definition. Other facts are added to ensure a proper relationship between signatures (implied by the nature of Alloy modelling). For example, saying that each element has one interface and each interface belongs to one element is not enough to ensure a one-to-one relationship between an element and its interface. Instead, we need to also include a fact stating that the relationship mapping elements to interfaces is equal to the inverse relationship (denoted by ~) mapping interfaces to elements:

```
interface = ~_element
```

4.2 Analysis Rules

The analysis rules are adopted, modified, and extended from the Habermann/Perry well-formed system composition paper [10]. The rules we included in our initial analysis are as follows.

4.2.1 Components Rules.

- Completeness: Each service provided by the system is provided by at least one of its components. In other words, the provide list of the system is a proper subset of the union of all services provided by individual components within that system. This ensures that the system will not promise to provide services that are beyond the capacity of its components. However, since our ADL allows mapping (refer to Section 3.1), checking this rule is less straightforward; it is not enough to check the interface of the system and match it against all components, but instead we need to check the chain of service mappings to ensure that the leaf of the mappings chain represents a service that is indeed provided by an existing component.

```
assert completeness{
  no c:sys.versions |
  some s:(c.interface.services -(c.mappingS).Service)
    + Service.(c.mappingS) |
    s not in BasicElement.interface.services and no
      (s.^((c.^elements.mappingS)+c.mappingS)
      & BasicElement.interface.services)
}
```

This rule represents the completeness rule for the system architecture. A looser form for component compositions is defined below, namely the *self-contained* rule.

4.2.2 Composition Rules.

- Self-contained: dependencies required by a composition are either satisfied by the composition itself (i.e. via components within the composition) or are exported. Exporting a dependency makes it clear that the composition cannot be used if the dependency is not eventually satisfied. Therefore, it does not make sense to export a dependency that is already satisfied by the composition and this should be avoided. A dependency can either be satisfied or exported but not both.

```
assert selfContained{
  no c:Composition |
    some d:c.elements.interface.dependencies |
    exported[d, c] <=> satisfied[d, c]}
```

- Self-sufficient: A composition is said to be self-sufficient if the services provided by the system are also provided by the composition (i.e. the provide list of the system is a proper set of services provided by the composition). This rule is needed since compositions usually include a subset of the components available within a system and we want to make sure that the composition realizing the system is able to provide all services promised by the system. A case where a composition is not self-sufficient is when the system provides a service that is not provided by the composition but instead provided by an individual component within the system. In that case, relevant components should be included accordingly in the composition until it matches the provide list of the system[1].

```
assert selfSufficient{
  no c:Composition| getServices[c] not in
    (c.elements.interface.services +
      (c.mappingS).Service)}
```

- The set of dependencies required by a composition can be *derived* from the (unsatisfied) dependencies of the components within the composition.
- The set of services provided by a composition is a proper subset of the union of the services provided by its components.

This is our initial set of rules. Additional rules, especially those related to constraints, are to be added in future work (refer to Section 7).

4.3 Translated Instance

Now that we have our abstract model ready, how do we represent a specific architectural instance? This is a straightforward process and can be achieved by simply extending signatures from the basic Alloy model appropriately. Most signatures in the basic model are declared abstract to enforce the inclusion of a specific set rather than an arbitrary set when creating an architectural instance. For example, if a given architecture has, say, two components A and B, then the abstract Component signature is extended to include A and B by adding the following line: `one signature A, B extends Component{}` If the `Component` signature was not declared abstract, then there is no guarantee that A and B are the only architectural components present in the architectural instance.

Recall that architectural elements are ultimately defined as a set of constraints. For example, a service specification (of a given component) is modelled as a set of I/O data constraints, I/O event constraints and Pre/Post constraints. These constraints are to be defined (by the developer) using Alloy notation and therefore no translation is necessary for this part of the instance. Additional facts that are specific to the architecture under analysis are added to the basic model. Facts may represent various constraints on individual architectural elements or on the architecture as a whole (e.g. the form of the architecture).

5 EXAMPLE

In this section we show how to model and analyze a given architecture. For simplicity, let us say the goal is to build a system that allows end users to open, edit, save, and print files. Moreover, let us assume that we have three main architectural components: 1) a File Manager (FM) responsible for services related to file management such as opening and closing files, and that relies on services provided by the Operating System (OS); 2) a Text Processor (TP) which provides services for manipulating text such as changing lower case letters to upper case; and 3) a Multifunctional Printer (MFP) that offers printing and scanning capabilities and depends on the facilities provided by a Printer Driver. In addition, we define a Generic Connector (GC) that coordinates the interaction between these components. Figure 2 shows how the components are architecturally composed to deliver the system in mind, where architectural elements (basic elements or compositions) are represented in boxes, services and dependencies appear in the upper and lower rows respectively, and arrows represent decomposition.

The top-level element represents (a version of) the system where the shown services and dependencies represent the system's interface. The mid-level elements in our example show intermediate compositions that use our three components and selectively export certain services. Ideally, dependencies are also exported if they fail to be satisfied (refer to Section 4.2). Some services are renamed or combined together, for example, *openFile* from the Editor element combines both *open* and *read* services from FM element whereas *save* (in Editor) is a new name for the *write* service (in FM).

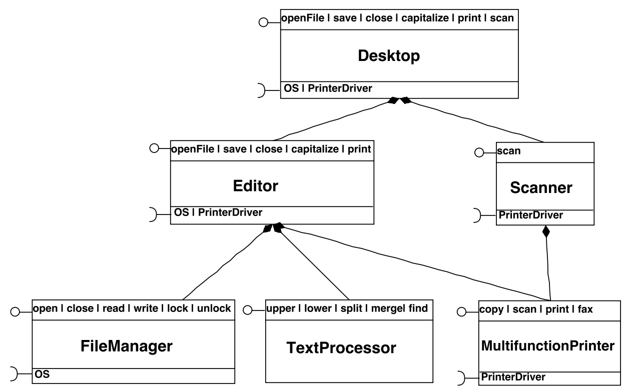

openFile | save | close | capitalize | print | scan
Desktop
OS | PrinterDriver
openFile | save | close | capitalize | print
Editor
OS | PrinterDriver
scan
Scanner
PrinterDriver
open | close | read | write | lock | unlock
FileManager
OS
upper | lower | split | merge| find
TextProcessor
copy | scan | print | fax
MultifunctionPrinter
PrinterDriver

Figure 2: Example architecture.

Now that we have a general idea of how our architecture would look like, we want to go ahead and analyze the architecture using our framework. Note that the specification of the architecture is not complete (e.g. architectural constraints have not been defined yet). Nevertheless, we still can proceed with analysis and at a later stage add the missing information (i.e. performing incremental analysis).

The first step is to write the architectural description of our example using the abstract ADL notation. The resulting description is communicated through an XML file (a sample is given in Appendix A). More information regarding this step can be found at [5].

Next, the architectural description is translated to an Alloy model. As mentioned in the previous section, an architectural instance is created by extending corresponding signatures from the basic model. For example, the MFP component is defined as follows:

```
open archModel as m
/**Arch-elem = Multifunction Printer (MFP)*/
one sig MFP extends Component{}{interface=iMFP}
one sig iMFP extends Interface{}{
  services= Copy+Scan+Print+Fax
  dependencies= PrinterDriver
  _element= MFP}
one sig Copy, Scan, Print, Fax extends Service{}
{_interface=iMFP}
```

[1]Note that, in some cases, compositions may provide services not required by the system, and hence, only a subset is exported as needed.

```
one sig PrinterDriver extends Dependency{}
```

Assuming the remaining components (and connectors) are defined in a similar way, the Editor composition can be defined as follows:

```
/**Composition = Editor*/
one sig Editor extends Composition{}{
  interface=iEditor
  elements = FM+TP+MFP+GC
  mappingS = {OpenFile->Open+OpenFile->Read}+
      {Save->Write}+{Capitalize->Upper}}
one sig iEditor extends Interface{}{
services = Close+Save+OpenFile+ Capitalize+Print
  dependencies = OS+PrinterDriver
  _element = Editor}
one sig OpenFile,Save,Capitalize extends Service{}
{_interface = iEditor}
```

Other compositions are defined in a similar way except that they do not have any mappings. Also, Desktop is marked as a version by adding the fact {Desktop in Version}.

Although this step is currently done manually, we plan to automate it in the future.

Now that we have an architectural instance using Alloy notation, the final step involves performing automated analysis against our analysis rules by executing the lines below.

```
check m/completeness for 15
check m/selfContained for 15
check m/selfSufficient for 15
```

No counterexamples are found, which means our example is likely to be correct and increasing the scope still does not change the result. However, if, say, a non-existing service is introduced part of Desktop element, as follows:

```
/**Composition = Desktop*/
one sig Desktop extends Composition{}{
  interface = iDesktop
  elements = Editor+Scanner+GC
  no mappingS
}
one sig iDesktop extends Interface{}{
  services=OpenFile+Save+Close+
         Capitalize+Print+Scan+Dummy
  dependencies = OS + PrinterDriver
  _element = Desktop
}
one sig Dummy extends Service{}
```

Then running the altered model yields a counterexample indicating incorrectness of the model as per our rules.

Another example is not to export a (unsatisfied) dependency. Again, we modified the Desktop element (interface part only) such that it does not export the PinterDriver dependency anymore.

```
one sig iDesktop extends Interface{}{
  services = OpenFile+Save+Close+
    Capitalize+Print+Scan
  dependencies = OS //+ PrinterDriver
  _element = Desktop
}
```

A counterexample was found as expected violating the `selfContained` assertion.

6 COST/BENEFIT OF APPROACH

As discussed previously, the need for more intuitive notation for formal analysis is emphasized by practitioners. The main two languages that are used in our approach are XML and Alloy. The benefit of using XML is that it is a semi-formal language, commonly used in the industry, and requires minimum time to learn. Alloy on the other hand is based on formal notation but at the same time uses more conventional syntax and simpler semantics compared to other specification languages currently available (such as OCL) [14]. This makes Alloy easier to learn and use. Moreover, it is worth mentioning that once our approach is fully automated, software architects need only to learn how to write architectural constraints using Alloy instead of mastering the whole language. In other words, having both background knowledge in first-order logic and basic notation of Alloy (for constraints writing purpose) should suffice in adopting our approach.

Another feature of our framework it that it provides fully automated analysis. Once the architectural model is translated to Alloy, it can be automatically executed for analysis. Furthermore, our approach supports incremental analysis. This means that a given architecture does not have to be completely written before starting any analysis activities. For example, we could still analyze our example from Section 5 even if some elements were not part of the architecture yet (e.g. Scanner and MFP elements were not defined). Incremental modelling with analysis allows for early detection of flaws, is less costly to make changes, and helps in brining insight regarding properties that were overlooked.

However, one of the immediate disadvantages of using our framework, in its current state, is that it is not fully automated. For relatively large systems, creating architectural descriptions using XML is a tedious task and the translation of architectural instances to Alloy is time consuming. However, we plan to overcome these issues by having a customized GUI that automatically generates the required XML file and to automate the translation to Alloy structures.

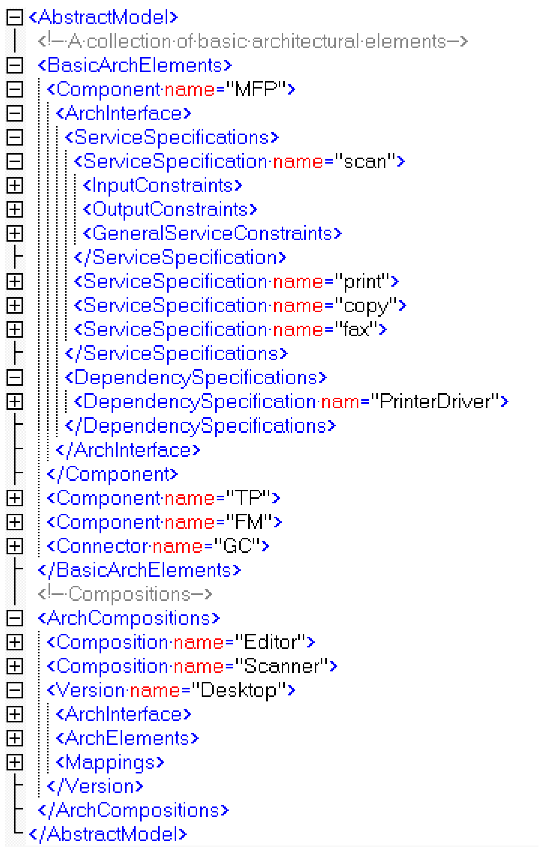
Another down side of our approach is that, by using Alloy, the scope of analysis is limited and we cannot fully guarantee the correctness of our architectural model under test. Increasing the scope might result in finding a counter example but would also reduce performance. However, since incremental analysis is supported in our framework, performance can be enhanced by using the concept of *divide and conquer*; subparts of the architecture can be analyzed independently and then grouped together for final analysis. Finally, our framework does not support features like multiple views management, cross-view checking and dynamic analysis. At this stage it is too early to include such features but they can be introduced later as extensions to our work.

7 CONCLUSION AND FUTURE WORK

In this paper, we laid the building blocks of e presented a basis for a framework for analyzing software architecture. In our proposed model, architectural instances are described in a semi-formal language and are then translated into an Alloy model. The analysis is based on a given set of rules related to architectural structure such as completeness. An example of our approach was given showing the result of automatically checking the architecture against these rules.

Our future research on formal analysis will cover remaining architectural aspects including system behavior and quality attributes. It will also cover rules for evaluating constraints within basic architectural elements (intra-relationship), within an architectural composition (inter-relationship), or within an architectural region to detect possible consistency violations. Moreover, since many architecture systems are relatively large, we intend to facilitate the process of writing architectural specification through a user interface and to automate translating architectural instances into their corresponding Alloy models.

A EXAMPLE ARCHITECTURAL SPECIFICATION



```
<AbstractModel>
 <!--A collection of basic architectural elements-->
 <BasicArchElements>
  <Component name="MFP">
   <ArchInterface>
    <ServiceSpecifications>
     <ServiceSpecification name="scan">
      <InputConstraints>
      <OutputConstraints>
      <GeneralServiceConstraints>
     </ServiceSpecification>
     <ServiceSpecification name="print">
     <ServiceSpecification name="copy">
     <ServiceSpecification name="fax">
    </ServiceSpecifications>
    <DependencySpecifications>
     <DependencySpecification nam="PrinterDriver">
    </DependencySpecifications>
   </ArchInterface>
  </Component>
  <Component name="TP">
  <Component name="FM">
  <Connector name="GC">
 </BasicArchElements>
 <!--Compositions-->
 <ArchCompositions>
  <Composition name="Editor">
  <Composition name="Scanner">
  <Version name="Desktop">
   <ArchInterface>
   <ArchElements>
   <Mappings>
  </Version>
 </ArchCompositions>
</AbstractModel>
```

REFERENCES

[1] 2012. Alloy. (2012). http://alloy.mit.edu/alloy/

[2] Jean-Raymond Abrial and Jean-Raymond Abrial. 2005. *The B-book: assigning programs to meanings.* Cambridge University Press.

[3] Robert Allen and David Garlan. 1997. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 6, 3 (1997), 213–249.

[4] PerOlof Bengtsson, Nico Lassing, Jan Bosch, and Hans van Vliet. 2004. Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software* 69, 1 (2004), 129–147.

[5] Sutirtha Bhattacharya and Dewayne E Perry. 2007. An Abstract Architectural Model for Composition, Analysis and Evaluation. *Working International Conference on Software Architecture* (January 2007).

[6] Junhua Ding and Dongmei Zhang. 2015. Modeling and Analyzing Publish Subscribe Architcture using Petri Nets.. In *SEKE.* 589–594.

[7] Peter H Feiler, David P Gluch, and John J Hudak. 2006. *The architecture analysis & design language (AADL): An introduction.* Technical Report. DTIC Document.

[8] David Garlan. 2014. Software architecture: a travelogue. In *Proceedings of the on Future of Software Engineering.* ACM, 29–39.

[9] David Garlan, Robert T Monroe, and David Wile. 2000. Acme: Architectural description of component-based systems. *Foundations of component-based systems* 68 (2000), 47–68.

[10] A. Nico Habermann and Dewayne E. Perry. 1980. *Well Formed System Composition.* techreport CMU-CS-80-117. Carnegie-Mellon University.

[11] Xudong He and Yi Deng. 2002. A framework for developing and analyzing software architecture specifications in SAM. *Comput. J.* 45, 1 (2002), 111–128.

[12] Paola Inverardi and Massimo Tivoli. 2003. Software architecture for correct components assembly. *Formal Methods for Software Architectures* (2003), 92–121.

[13] Valérie Issarny, Amel Bennaceur, and Yérom-David Bromberg. 2011. Middleware-Layer Connector Synthesis: Beyond State of the Art in Middleware Interoperability.. In *SFM,* Vol. 6659. Springer, 217–255.

[14] Daniel Jackson. 2012. *Software Abstractions: logic, language, and analysis.* MIT press.

[15] Rick Kazman, Len Bass, Gregory Abowd, and Mike Webb. 1994. SAAM: A method for analyzing the properties of software architectures. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on.* IEEE, 81–90.

[16] Jung Soo Kim and David Garlan. 2006. Analyzing architectural styles with alloy. In *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis.* ACM, 70–80.

[17] Jeff Kramer, Jeff Magee, and Sebastian Uchitel. 2003. Software architecture modeling & analysis: A rigorous approach. In *Formal Methods for Software Architectures.* Springer, 44–51.

[18] Nico Lassing, Daan Rijsenbrij, and Hv Vliet. 1999. The goal of software architecture analysis: Confidence building or risk assessment. In *Proceedings of First BeNeLux conference on software architecture.* 47–57.

[19] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. 1995. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering* 21, 4 (1995), 336–354.

[20] Jeff Magee, Jeff Kramer, and Dimitra Giannakopoulou. 1999. Behaviour analysis of software architectures. In *Software Architecture.* Springer, 35–49.

[21] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. 2013. What industry needs from architectural languages: A survey. *IEEE Transactions on Software Engineering* 39, 6 (2013), 869–891.

[22] Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Huaxi Yulin Zhang. 2014. Towards automating the coherence verification of multi-level architecture descriptions. *Proc. of the 9th ICSEA, Nice, France* (2014), 416–421.

[23] Mert Ozkaya and Christos Kloukinas. 2013. Are we there yet? Analyzing architecture description languages for formal analysis, usability, and realizability. In *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on.* IEEE, 177–184.

[24] Dewayne E Perry and Alexander L Wolf. 1992. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes* 17, 4 (1992), 40–52.

[25] Richard N Taylor, Nenad Medvidovic, and Eric M Dashofy. 2009. *Software architecture: foundations, theory, and practice.* Wiley Publishing.

[26] Perla Velasco-Elizondo, Vishal Dwivedi, David Garlan, Bradley Schmerl, and Jose Maria Fernandes. 2013. Resolving data mismatches in end-user compositions. In *International Symposium on End User Development.* Springer, 120–136.