Research Reports in Software Engineering and Management 2012:02 ISSN 1654-4870

Towards Formalizing Assumptions on Architectural Level: A Proof-of-Concept

Md Abdullah Al Mamun, Matthias Tichy, Jörgen Hansson

Department of Computer Science and Engineering

Towards Formalizing Assumptions on Architectural Level: A Proof-of-Concept

Md Abdullah Al Mamun, Matthias Tichy, Jörgen Hansson

© Md Abdullah Al Mamun, Matthias Tichy, Jörgen Hansson, 2012

Report no 2012:02

ISSN: 1654-4870

Department of Computer Science and Engineering

Chalmers University of Technology and University of Gothenburg

Chalmers University of Technology and University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden

Telephone + 46 (0)31-772 1000

Göteborg, Sweden 2012

Research Reports in Software Engineering and Management No. 2012:02

Towards Formalizing Assumptions on Architectural Level: A Proof-of-Concept

Md Abdullah Al Mamun
Matthias Tichy
Jörgen Hansson

Department of Computer Science and Engineering
CHALMERS | University of Gothenburg

Gothenburg, Sweden 2012

Towards Formalizing Assumptions on Architectural Level: A Proof-of-Concept

Md Abdullah Al Mamun, Matthias Tichy, and Jörgen Hansson

Software Engineering Division
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg, Sweden
[abdullah.mamun|tichy|jorgen.hansson]@chalmers.se

Abstract. While designing an architecture, architects often make assumptions about different factors like execution environment, structural properties of the artifacts, properties of input/output data etc. Implicit and invalid assumptions have been identified as a primary reason for architectural mismatches. Such mismatches cause nightmares to the people working at the system integration phase. Today's complex systems operate in dynamic and rapidly changing environments. Implicit assumptions in the reusable components often make it challenging to adopt the components in a changed operational domain. If not documented, assumptions are often forgotten and it is both difficult and expensive to find them out from previously built software. This paper aims to formally capture assumptions at the architecture level so that they can be checked automatically throughout the system development. Automated checking of assumptions would facilitate a practicable assumption management system for complex and large-scale software system development with smooth integration and evolution.

Keywords: assumptions, formal specification, software architecture, integration, evolution

1 Introduction

Software development for big software systems usually follows the divide-and-conquer principle by developing an architectural design in the early phases and subsequently developing the different parts of the architecture independent from each other. The parts are later taken and integrated with each other. This also enables a reuse of proven parts of older systems.

This approach is facilitated by a thorough definition of the interfaces of the different parts. However, still many problems appear in the integration phase. Those issues are often the result of implicit and/or invalid assumptions by the developers of the different parts. Invalid assumptions have been reported as the root cause of system and project failure [1–3]. Our previous work [4] explores existing challenges of assumptions and their possible areas of use.

Examples of assumptions as given by Garlan et al. [1] have been classified with respect to the categories *Nature of components*, *Nature of the connectors*, *Global architectural structure*, and *Construction process*. Examples of the first kind of assumptions deal with which components control the control flow or how the environment will manipulate data managed by a component.

In order to overcome those problems, these implicit assumptions, first, have to be made explicit and, second, must be formalized in such a way that they are automatically checkable throughout the software development process. Moreover, assumptions should be already addressed at the architectural level as the architecture is used for decomposing and later integration as mentioned before.

Several approaches [5–8] have been proposed which either use formal or semi-formal approaches to manage assumptions. While the semi-formal approaches, are not usable for automatic checking, the formal approach [5] does not allow to formalize cross-cutting assumptions which are related to several components but however focuses on connections between two components.

Our vision for a complete formal assumption management approach is (1) to develop a meta-model to capture assumptions at different system levels, e.g., component, system, system of systems, (2) enrich the components as basic software building blocks with assumptions, and (3) use these models to identify problems early in the software development process and throughout the whole system development process. We specifically focus not only on pairwise assumptions between components but also on cross-cutting assumptions.

The contribution of this paper is to show that existing real-life assumptions, the ones described by Garlan et al. [1] which are related to the software architecture, can be formalized and checked using the Alloy language [9] and the Alloy tool as the first step towards realizing our vision.

In the next section, we discuss related approaches. Section 3 contains a presentation of a taxonomy of assumptions including the one that our approach targets. In Section 4, we present how we formalize assumptions on the architectural level and check it using the Alloy tool. We conclude in Section 5 and give an outlook on future work.

2 Related Work

The attempts on modeling assumptions can be divided into two classes that are formal and semi-formal. By formal, we focus on those modeling approaches that formalize the statement or fact of an assumption along with its different attributes e.g., category description, impact, criticality, tractability, states etc. On the other hand, by semi-formal, we focus on those approaches that describe the statement of the assumption in natural language but attributes are captured structurally. Assumptions are informal when they are documented without proper structure and are completely defined using free text.

The advantage of formally modeled assumptions is that they are potentially machine-checkable. Even though, semi-formal approaches are not machine-checkable as the fact/statement of an assumption is in free text, assumptions can be retrieved/searched by their attributes. For example, in semi-formal approaches, it would be possible to find, which assumptions might impact a critical component in the system.

2.1 Formal Approaches

An assumption management framework has been developed by Tirumala [5] who has developed a language for documenting assumptions formally. The assumptions and the guarantees for the assumptions are encoded as part of the architectural components in a way that it is possible to check automatically whether there is any mismatch between an assumption and its associated guarantee. They have implemented the framework for the architectural assumptions in the Architecture Analysis and Design Language (AADL) [10], which is an Architecture Description Language (ADL).

There are two major differences between our approach and that of Tirumala [5]. Firstly, we focus on assumptions that target existing architectural artifacts. Thus, in our approach, an assumption is constructed based on what we already have in the architecture. If not, we have to add or modify the architecture and then capture the assumptions. So, in our case, assumptions are tightly coupled with the architecture. On the other hand, assumptions are loosely coupled with the architecture in the approach of Tirumala [5]. In that case, an assumption is primarily based on parameters, which are assumed properties that a component makes about another component. Examples of such parameters are error rate, sensing delay, sensing jitter etc., which are not necessarily part of the architecture.

Secondly, we have focus on the cross-cutting nature of assumptions where an assumption can relate to more than two components and different issues among them. Tirumala [5] has primarily modeled assumptions pair-wise between two components. Thus, the composition of an assumption with a guarantee is also restricted between two components. Let's take an example assumption "Component X expects that all other components have component Y as a library". In our approach, we can model this assumption as a single assumption. However, the approach of Tirumala would require writing one assumptions for each of the other components. Hence, removing or modifying such assumptions might cause inconsistencies in the assumption model. Moreover, certain crosscutting assumptions are not possible to model in their approach. For example, the assumption "Component X assumes that all the components having a property P are completely independent of each other i.e., there is no connection between them" is not possible to model.

2.2 Semi-Formal Approaches

A simple assumption management system prototype has been developed by Lewis et al. [6]. The prototype records and extracts assumptions from code written in Java into a repository. The assumptions are written in the code using XML and saved into the repository using an assumption extractor. It is possible to browse and search assumptions with given criteria by using this web-based assumption management system. A person who acts as a validator, reviews the stored assumptions. The management system also maintains system and project related information like users, roles, projects and types of assumptions. This prototype focuses on the assumptions at the implementation level.

Lago and Vliet [11] developed a meta-model for explicating assumptions in the software architecture. The model is able to capture assumption dependencies between the product feature model and the architectural model. They have discussed the essence of cross-cutting assumptions and worked with a software product family architecture implementing variability to achieve flexibility. Their focus was on non-technical (e.g., managerial, organizational), and cross-cutting assumptions. They introduce the term invariability and argue that invariability should also be modeled along with variability to let the model express what cannot be changed. They argued that explicit assumptions modeling would capture invariability because when assumptions are taken as granted, they act like constraints that impose limitations on the system organization and behavior.

An assumptions modeling method for explicating assumptions in the AADL has been developed by Ordibehesht [8]. This modeling method consists of an assumption specification meta-model for structuring assumptions information and an assumptions specification approach to specify the meta-model together with the architecture descriptions. The meta-model contains dependency information between the assumptions and the architectural components in order to facilitate traceability.

All the approaches in this subsection capture the statement/fact of an assumption as free text and, thus, the assumptions are not machine-checkable.

3 Taxonomy of Assumptions

Garlan et al. [1] focused on architectural assumptions. They discuss four main assumptions categories and some subcategories related to components and connectors that can result in architectural mismatch. The categories are:

- Nature of component: This category describes three subcategories *infrastructure, control model* and *data model*. *Infrastructure* assumptions are about the substrate on which the component is built. *Control model* assumptions are about which components will control the sequencing of computation and *data model* assumptions are about the way data, managed by a component, would be manipulated by the environment.

– Nature of connectors: Two subcategories of this category are - *protocols* that captures assumptions about the characteristics of a connector's patterns and interactions and *data model* that is concerned with the kind of data being communicated.
– Global architecture structure: This category captures assumption about the topology of the system communication. It also covers existence, i.e., presence or absence related assumptions of particular components and connectors.
– Construction process: This category includes assumptions related to the order in which building blocks are instantiated.

Dewar et al. [12] define assumptions in the context of assumption-based planning. In a later work, Dewar [13] has classified assumptions into categories - *about problems* vs. *about solutions, implicit* vs. *explicit, unaddressed* vs. *addressed, non-load-bearing* vs. *load-bearing, invulnerable* vs. *vulnerable* and *one-sided* vs. *two-sided*. The classification of Dewar [13] is broad in the sense that it covers a wide variety of assumptions including facts, constraints, design decisions and design rationales. It should be mentioned that assumptions often overlap requirements, constraints, and design rationales. A brief description on how assumptions are related to these concepts, can be found in [4].

Lewis et al. [6] have presented a classification of assumptions from the viewpoint of the software developers while they are coding. The assumptions types are *control, environment, data, usage,* and *convention*. Steingruebl and Peterson [14] support the classification of Lewis et al. [6] and suggest adding detailed level, e.g. checklist under the major assumption types. They also mention *security* as an assumption type. A classification of three assumptions classes are presented by Lago and Vliet [11], motivated by the general information system literature. Their study focuses on the architectural assumptions which are *technical, organizational,* and *managerial*. Spiegel et al. [15] identify three major classes of constraints that are based on the types of object attributes in the constraints (*invariant* vs. *dynamic*) and on the object scope of the constraints (one vs. many) that are *invariant, dynamic,* and *inter-object*.

Tirumala [5] classifies assumptions based on three dimensions namely *time-frame*, *criticality* and *abstraction*. Three assumption types *static*, *system configurations* and *dynamic* are described under *time-frame*. The validity of *static* assumptions remains the same during the lifetime of the software. However, validity can be changed as the system evolves. For the *system configuration* assumptions, validity does not change during a single execution of the system; though, validity can be changed between different executions. *Dynamic* assumptions are those whose validity might be changed during the system's execution.

King and Turnitsa [16] mentioned some possible assumptions classes, which are *intension* vs. *Extension, primary* vs. *Derivative, joint* vs. *Disjoint with others, exogenous* vs. *Endogenous, deterministic* vs. *Probabilistic*, and *controllable* vs. *Non-controllable*.

Assumptions can also be classified according to their state e.g., *unchecked* vs. *checked, invalid* vs. *valid, conflicting,* and *mismatched*.

This study focuses on the classification proposed by Garlan et al. [1] because the classification is based on the structural relations between architectural artifacts with a focus on component based software systems which fits well to our vision of assumption management on an architectural level.

4 Formalizing Assumptions

We have taken the assumptions stated by Garlan et al. [1]. The assumptions were identified in an attempt to develop a system Aesop with extensive use of existing piece of software. Aesop is an environment-generating system that was constructed to produce a custom design environment from a set of architectural style descriptions given as input. The basic idea of Aesop is to support creating new architectural styles and then use them to create architectural design. Styles are configured with the generated environment in such a way that it guides the designer in making decision about the architectural design. Aesop offers shared infrastructure which consists of different tools and packages providing basic support service for architectural design. The shared infrastructure includes *Interviews* - a graphical user interface for the modification and creation of new design; *OBST* - an object-oriented database for storing the designs; *SoftBench* - an event-based tool-integration framework to enable integrating new tools (e.g., compilers, analysis tools etc.) easily; *MIG* (Mach RPC Interface Generator) - an RPC stub generator. The estimated development time of the system was six months and one person-year. But in reality, after two years and about five person-years, only a prototype of the system was build, which was sluggish and difficult to maintain. Mismatched architectural assumptions were identified as the primary cause of the problems.

The discussed assumptions are primarily concerned with the reuse of the four standard pieces of software i.e., *Interviews*, *OBST*, *SoftBench*, and *MIG*. Since *MIG* and *SoftBench* are related to communication, they are described as connectors by Garlan et al. [1]. However, we have modeled them as components since they are standard software packages.

A brief description of the assumption categories are given in Section 3 and detailed description of the assumptions and their categories can be found in [1]. The selected assumptions and the problems occurred due to them are briefly described below.

- **Category: Nature of component**
  - **Subcategory: Infrastructure**
    −− **Assm_NoCom_Infra**: SoftBench broadcast message Server assumed that all other components would have a graphical user interface (GUI). Thus, SoftBench uses the X library to provide communication primitives. In reality, tools like compilers, design-manipulation utilities etc. did not have their own GUI and they had to include the X library as subcomponent in order to avoid communication problems.

– **Subcategory: Control model**

–– **Assm_NoCom_CM**: Communication events are dealt by three packages named SoftBench, Interviews, and MIG. These packages use an event loop for this purpose. The details of the communication substrate are encapsulated by the event loops. These details help a developer to structure a component's interactions with its environment around a set of callback modules. However, event loops used by the different packages are not identical and none of the control loops is compatible with the others. SoftBench realizes its main thread of control on the X Intrinsics package. On the other hand, the event loop in Interviews is implemented directly in terms of Xlib routines where the event loop is an object-based abstraction. A handcrafted loop for the server to wait for Mach messages is used by MIG.

This assumption was mainly about which part of the software held the main thread of control. But as the components used different types of event loops, it was not possible to bridge different event-control regimes by using an event gateway. The primary reason of this problem arises due to the incompatibility of the different event loops.

– **Category: Nature of connectors**

– **Subcategory: Protocols**

–– **Assm_NoConn_Protocol**: Two types of interactions namely notify status as broadcast, and request/reply pair are provided by the SoftBench Broadcast Message Server. Even though these two interaction types are not compatible, SoftBench assumes that they are compatible and attempts to handle both of them uniformly. However, since these two interaction types were not compatible, other tools communicating through SoftBench had to introduce concurrency even though sequential programming is enough and easy to implement such a case. As a solution to this problem, MIG (Mach RPC) replaced SoftBench for the database interaction because it was the most critical and heavily used communication link in the Aesop system.

– **Subcategory: Data models**

– **Assm_NoConn_DM**: MIG (Mach RPC) and SoftBench are two communication mechanisms. A C-based Model is provided by the MIG where in a C-based model, data is exchanged through procedure calls and data is realized based on C constructs and Arrays. On the other hand, SoftBench communicates data in the form of ASCII strings.

Both MIG and SoftBench had different assumptions about the data models. MIG assumes that other tools would have C-based model and SoftBench assumes that most communication will be in the form of ASCII strings (i.e., about files and the data contained in them). To solve this problem, translation routines and intermediate interfaces were introduced between different models. However, translating every database call resulted in a huge overhead, which was the most critical performance bottleneck in the system.

– **Category: Global architecture structure**

– **Assm_GAS**: OBST makes an assumption that the other tools are only connected with OBST and they are not directly connected to each other. In other words, all the tools are independent of each other and it views

any concurrency among components as conflict. In order to enforce this assumption, OBST handled incoming requests one at a time. However, in reality, this assumption was invalid because other components had connections between them. In such a scheme, problem occurs when a component tries to release a resource to another component while the first component has not completed its task. In the Aesop system, the mechanism of OBST did not allow when a component tried to release the database to a corresponding component.

An assumption of the category nature of component with subcategory infrastructure has not been modeled because it deals with performance issues. We have not considered the data model assumption, which is the subcategory of the nature of component category due to the reason that this assumption is more implementation related rather than architecture. The other assumption that we have not modeled is of category construction process, which is a process related assumption.

The development of Aesop system by Garlan et al. [1] encountered six main problems while integrating the four subsystems, which we have mentioned as components and described earlier in this section. It is very interesting that virtually all of these problems occurred because of architectural mismatch due to invalid assumptions. If the assumptions were captured along with the components by the component developers, an early analysis of the system, in terms of assumptions, could have been performed to check the applicability of the components. In this way, modeling assumptions can help performing early analysis revealing architectural defects that would obviously help measuring the complexity of the project, checking the appropriateness of certain reusable components, and making better estimation of project time and cost.

4.1 Capturing the Architecture

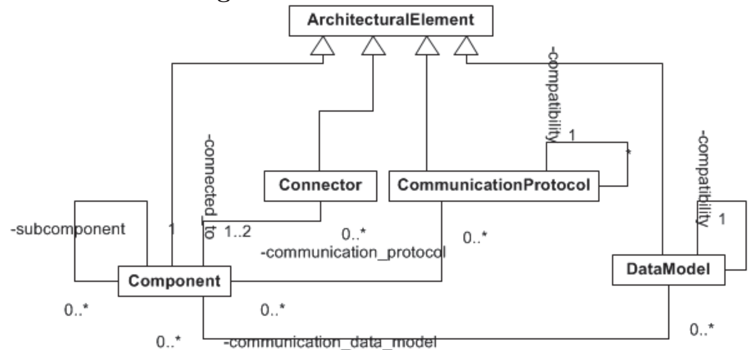
We have used the Alloy [9] language to model the architecture and the assumptions. It is a declarative language influenced by Z specification language. Models in Alloy are amenable to fully automatic semantic analysis. Alloy expressions are based on first order logic and analysis of the expressions is performed by SAT solvers. In order to be fully automated, Alloy sacrifices the complete proof of a system's correctness and instead, tries to find counterexamples of the system showing constraints violation within a limited scope [9].

The meta-model that we have used to build the architecture, is very simple and it fits well within the scope of the selected assumptions. We could select a rich meta-model specific to an architecture modeling language but for simplicity, we have kept it simple. The meta-model of the architecture is depicted in Fig. 1 in UML notation.

As Fig. 1 shows, *ArchitecturalElement* is the top most class that can refer to any other classes in the meta-model. *CommunicationProtocol* and *DataModel*

classes, extended from *ArchitecturalElement*, captures communication protocol and data model related information. The *compatibility* association maps both of the classes to themselves with multiplicity one to many. The *Connector* class captures the connections between components. The Component class may represent software components/tools. A component can be connected to zero to more other components as depicted by *connected_to* association. However, a connector must be connected to at least one component and can be connected to at most two components. The associations *subcomponent*, *communication_protocol* and *communication_data_model* respectively show that a component may have zero to many Components as subcomponents, *CommunicationProtocols* and *DataModels*.

Fig. 1. The architecture meta-model



Listing 1.1 shows the meta-model in Alloy syntax. In Alloy, all data types, even sets, scalars, tuples, everything is a relation. However, Alloy can be interpreted as object-oriented (OO) languages like C++/Java, which is helpful to understand an Alloy model at a glance. However, OO often leads to wrong understanding and implementation of Alloy thus, it is important to understand Alloy in terms of sets, elements, and relations among sets and elements. For a high level understanding, here, we briefly state how to read the second *sig* and *fact* statement of Listing 1.1 in OO terminologies.

Sig (signature) statement is used to define set (i.e., class in OO). So, Component is a class extended from superclass *ArchitecturalElement*. *subcomponent* is a field of *Component* pointing to zero to many *Components*. The *fact* statement is used to capture constraints. In *fact about_component_connector*, *cm* and *cn* are instances of class *Component* and *Connector*. Dot (.) is used to access a field. So, *cm.connected_to* would return something of type *Connector*.

Two fact statements in Listing 1.1 model constraints about *Component* and *Connector*. Of them, *about_connector* says that a connector can not exist without being connected to a *Component* and it can be connected to maximum two *Components*. Fact *about_component_connector* says that if there is a connection

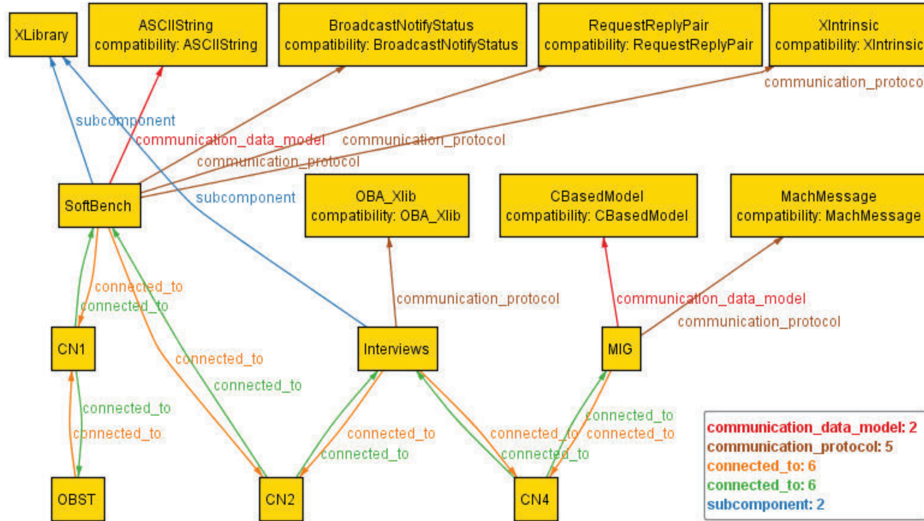
from a *Component* to a *Connector* then there must be a connection from that *Connector* to the *Component* and vice versa.

```
abstract sig ArchitecturalElement {}
abstract sig Component extends ArchitecturalElement {
   subcomponent: set Component,
   connected_to: set Connector,
   communication_protocol: set CommunicationProtocol,
   communication_data_model: set DataModel
}
abstract sig Connector extends ArchitecturalElement {
   connected_to: set Component
}
fact about_connector {
   all cn:Connector | cn.connected_to != none
   all cn:Connector | #cn.connected_to =< 2
}
fact about_component_connector {
  all cm: Component | all cn: Connector | (cn in cm.
     connected_to implies cm in cn.connected_to) &&
  (cm in cn.connected_to implies cn in cm.connected_to )
}
abstract sig CommunicationProtocol extends
    ArchitecturalElement {
    compatibility: set CommunicationProtocol
}
abstract sig DataModel extends ArchitecturalElement {
   compatibility: set DataModel
}
```

Listing 1.1. Meta model of the architecture in Alloy

The complete model along with the instance-model of the architecture is available online at `http://www.cse.chalmers.se/~abmd/onlinematerials/Garlan_Assumptions_Model.als`. Fig. 2 shows an instance of the architecture modeled in Alloy. The instance model in Fig. 2 shows that there are total five components (*SoftBench, OBST, Interviews, MIG* and *XLibrary*), three connectors (*CN1, CN2* and *CN4*), seven communication protocols of them three of type *EventLoop* (*OBA_Xlib, MachMessage* and *XIntrinsic*), two of type *DataModel* (*CBasedModel* and *ASCIIString*) and two *CommunicationProtocol* (*BroadcastNotifyStatus* and *RequestReplyPair*). The *compatibility* relations in the communication protocols are represented as text instead of arrows. *XLibrary* is a component that is possessed by *SoftBench* and *Interviews* as subcomponent. *SoftBench* is the most highly connected component that has two *CommunicationProtocol*, a communication *DataModel* and an *EventLoop*. It is also connected to *OBST* and *Interviews* components respectively via *CN1* and *CN2* connectors. *MIG* defines a communication *DataModel* and a *CommunicationProtocol*, and is also connected to *Interviews*. *OBST* is the least connected component that does not define any *communication protocol* or *DataModel*.

Fig. 2. An instance of the architecture modeled in Alloy



4.2 Capturing and Checking Assumptions

Alloy assertions are used to capture the assumptions. Then these assumptions are checked using *check* statements. Alloy *check* yields possible counterexamples revealing mismatches of an assumption with the modeled architecture. If an assumption is valid, a counterexample associated with that assumption indicates problem(s) in the architecture. On the contrary, if we find that the case identified by the counterexample is valid then the assumption itself is invalid. If there is no counterexample generated from an assumption then both the architecture and the assumption is valid.

Among the assumptions, some are *general* that can be used without necessarily modifying the assumption in a changed composition of the architecture. Thus they are easy to maintain and reusable. Other assumptions are *application specific* that might require modification of the assumption in a new composition of the architecture. For example, assumptions *Assm_NoCom_CM* in Listing 1.3 and *Assm_NoConn_DM* in Listing 1.5 are general as they can be reused in any composition of the architecture constructed following the same meta-model. On the other hand, *Assm_NoCom_Infra* in Listing 1.2, *Assm_NoConn_Protocol* in Listing 1.4 and *Assm_GAS* in Listing 1.6 are application specific that are only applicable for the specific cases.

The rest of this subsection describes the captured assumptions and the counterexamples resulting from them. The counterexamples are checked with respect to the instance model of the architecture as shown in Fig. 2.

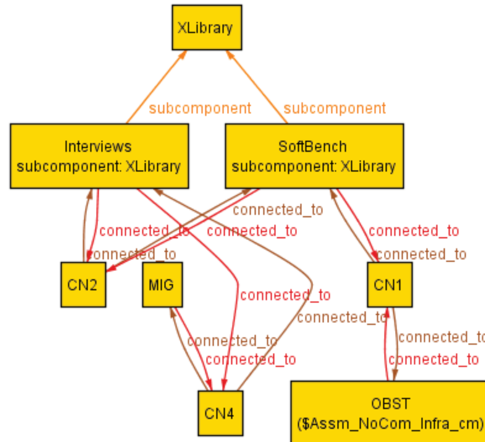
Assm_NoCom_Infra In this assumption, the *SoftBench* component assumes that other components connected to *SoftBench* would use *XLibrary* as subcomponent. The assumption modeled in Alloy is shown in Listing 1.2, which can be equivalently read in natural language as - *If a Component is connected to SoftBench via a Connector then the Component must have XLibrary as subcomponent.*

Among the components, *Interview* and *OBST* are connected to *SoftBench* where *Interviews* possesses *XLibrary* as subcomponent but *OBST* does not, which is a violation of this assumption. When this assumption is checked, a counterexample depicting the violation (e.g., mismatch) of this assumption is reported by Alloy as shown in Fig. 3. The *Assm_NoCom_Infra_cm* tag in *OBST* in Fig. 3 says that being an instance of *cm* of type *Component* in Listing 1.2, *OBST* violates the *assert* statement *Assm_NoCom_Infra* i.e., the assumption.

```
assert Assm_NoCom_Infra {
    all cm: Component | one sb: SoftBench | one xl: XLibrary |
        sb in cm.connected_to.connected_to implies cm.
        subcomponent = xl
}
```

Listing 1.2. Assumption Assm_NoCom_Infra

Fig. 3. Counterexample of Assm_NoCom_Infra assumption



Assm_NoCom_CM This assumption relates three components *SoftBench*, *Interviews* and *MIG* which use different versions of *EventLoop CommunicationProtocol*. We have modeled this assumption in terms of the compatibility of different

EventLoop used by the components, which is shown in Listing 1.3. In natural language, the modeled assumption can be read as - *if two Components are connected via a Connector and one of them has EventLoop CommunicationProtocol then both of the Components must have EventLoop protocols that are compatible to each other.*

The checking of assumption Assm_NoCom_CM reports three unique counterexamples between the components *SoftBench* - *OBST*, *SoftBench* - *Interviews*, and *Interviews* - *MIG*. However, the total number of reported counterexample is ten because the Alloy checker finds similar situations that violate the assumption with different instances of *cm1, cm2* and *el*. It is noticeable that even though *OBST* does not define any *EventLoop*, the communication between *OBST* and *SoftBench* has been identified as a violation of this assumption since *SoftBench* defines *EventLoop*. Fig. 4 shows a counterexample of Assm_NoCom_CM this assumption. The counterexample expresses that *SoftBench*, and *Interviews*, being instances of respectively *cm1*, and *cm2*, have violate the assumption because there does not exist any *EventLoop* between the components that is compatible to both of the components.

```
assert Assm_NoCom_CM {
   all cm1, cm2: Component | all el: EventLoop |
   ((cm1 in cm2.connected_to.connected_to-cm2 ||
   cm2 in cm1.connected_to.connected_to-cm1) &&
   (el in cm1.communication_protocol ||
   el in cm2.communication_protocol)) implies
   (el in cm1.communication_protocol.compatibility &&
   el in cm2.communication_protocol.compatibility)
}
```

Listing 1.3. Assumption Assm_NoCom_CM

Assm_NoConn_Protocol *SoftBench* provides two interaction types *BroadcastNotifyStatus* and *RequestReplyPair*. However, *SoftBench* assumes that these two interactions are compatible where in reality they are not. Listing 1.4 shows the assumption in Alloy, which can be read in natural language as - *BroadcastNotifyStatus CommunicationProtocol is compatible to RequestReplyPair CommunicationProtocol*

Fig. 5 shows the counter example of this assumption that explains that there does not exist any *CommunicationProtocol cp* that is compatible to both *BroadcastNotifyStatus* and *RequestReplyPair*.

```
assert Assm_NoConn_Protocol {
   some cp: CommunicationProtocol | cp in
       BroadcastNotifyStatus.compatibility && cp in
       RequestReplyPair.compatibility
}
```

Listing 1.4. Assumption Assm_NoConn_Protocol

Fig. 4. Counterexample of Assm_NoCom_CM assumption

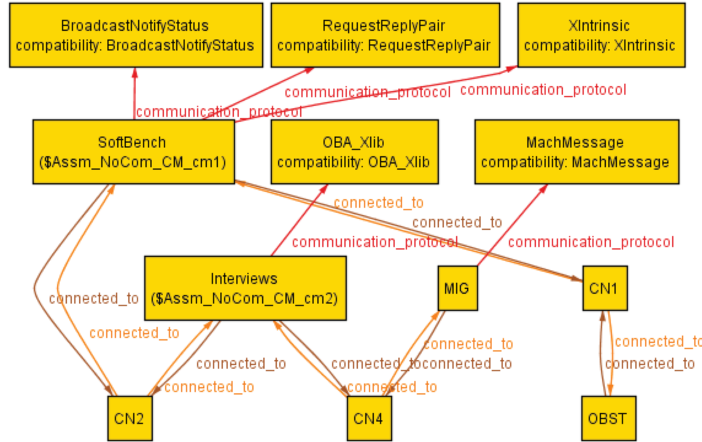
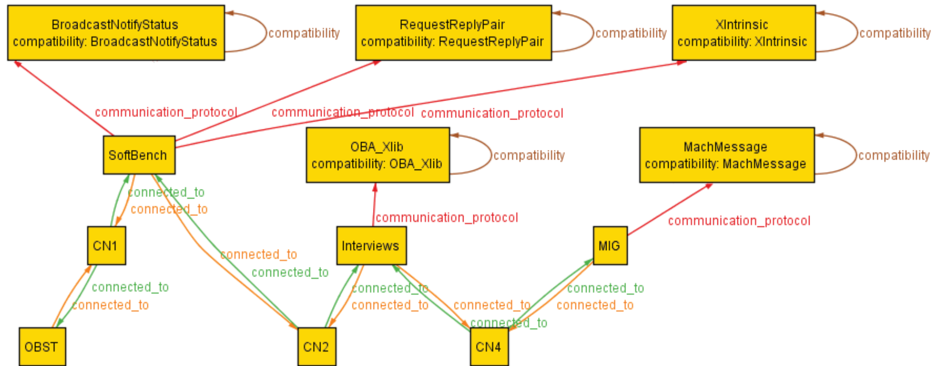


Fig. 5. Counterexample of Assm_NoConn_Protocol assumption



Assm_NoConn_DM *SoftBench* and *MIG* implements two types of data models for communication with other components because they assume that other components would communicate with the same data models that they have implemented. The assumption modeled in Alloy is shown in Listing 1.5 that can be read in natural language as - *if two Components are connected via a Connector and one of them specifies a communication DataModel then both of the Components must have DataModel that are compatible to each other.*

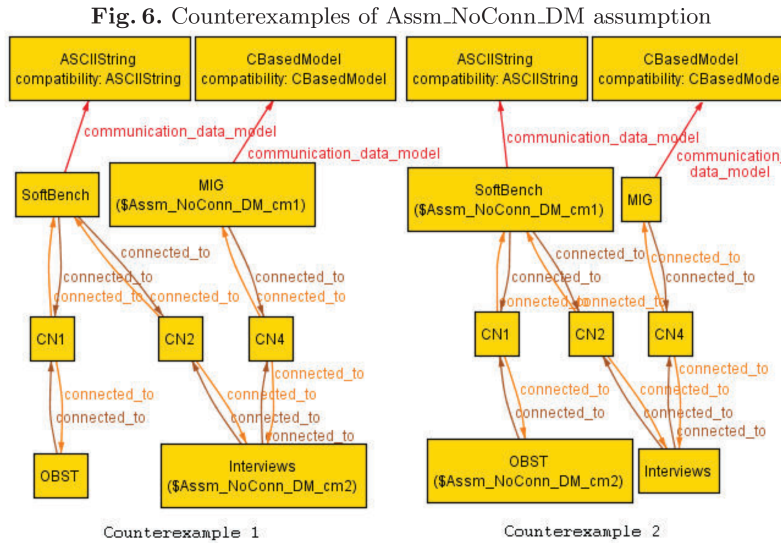
The check of this assumption results in three unique counterexamples out of total six where two of them are shown Fig. 6. Counterexample 1 in Fig. 6 shows that *MIG* and *Interviews*, being instances of *cm1* and *cm2* respectively, violate the assumption *Assm_NoConn_DM*, because, there does not exist any *DataModel* between *MIG* and *Interviews* that is compatible to both of them. Counterexample

2 shows the violation of the assumption between the components *SoftBench* - *OBST*.

```
assert Assm_NoConn_DM {
    all cm1, cm2: Component | all dm: DataModel |
    ((cm1 in cm2.connected_to.connected_to-cm2 ||
    cm2 in cm1.connected_to.connected_to-cm1) &&
    (dm in cm1.communication_data_model ||
    dm in cm2.communication_data_model)) implies
    (dm in cm1.communication_data_model.compatibility &&
    dm in cm2.communication_data_model.compatibility)
}
```

Listing 1.5. Assumption Assm_NoConn_DM

Fig. 6. Counterexamples of Assm_NoConn_DM assumption



Assm_GAS *OBST* assumes that the composition of the components forms a star-topology where *OBST* remains in the center keeping connections with other components. Listing 1.6 shows the assumption that can be read in natural language as - *components are only connected to OBST via connectors*.

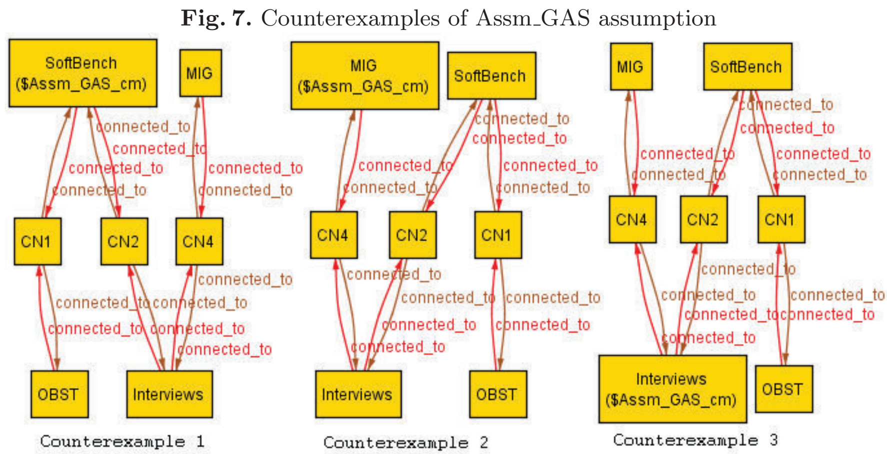
Within the selected scope of the architecture, *OBST* is only connected with *SoftBench*. Thus, all other connections among the other components are counterexamples of this assumption as shown in Fig 7. Counterexample 1 in Fig 7 shows that *SoftBench*, being an instance of *cm*, violates the *Assm_GAS* assumption because it is connected to a component (i.e., *Interviews*) in addition to

OBST. Counterexample 2 and 3 shows that *MIG* and *Interviews* are connected to components other than *OBST*.

```
assert Assm_GAS {
    all cm: (Component - OBST) | ((cm.connected_to).
        connected_to - cm - OBST) = none
}
```

Listing 1.6. Assumption Assm_GAS

Fig. 7. Counterexamples of Assm_GAS assumption



5 Conclusions and Future Work

In this paper, we have presented an approach to formally capture architectural assumptions that are amenable to fully automated check. We have focused on the cross-cutting nature of assumptions and modeled them using the Alloy language. The assumptions are selected from a study by Garlan et al. [1] reporting assumptions identified from a real project and categorizing them according to factors related to components and connectors.

The modeled assumptions are checked with Alloy tool, which is a checker based on SAT solvers. When checked, the assumptions show counterexamples depicting violation of the assumptions with the architecture i.e., mismatch in the architecture. The counterexamples have correctly explored every single mismatch in the architecture related to associated assumptions. Thus, reuse of components can highly benefit from capturing assumptions formally and it can provide early feedback on the architecture.

The proposed approach supports capturing general/high-level assumptions, which are less dependant on specific instances of architectural artifacts. Thus, they are both reusable and easy to maintain. It also supports capturing application-specific assumptions, those are more dependant on specific instances. In general, our approach supports capturing assumptions about the architectural structure including relations/connections among different artifacts.

This study is the first step of our vision to develop a holistic assumption management framework that would support identifying assumptions, capturing and managing assumptions and automated checking of them. In this study, we have worked with a very simple architecture meta-model that fits within the scope of the selected assumptions. Our future work includes, developing an enriched meta-model to capture assumptions at different system levels (e.g., component, system, etc.), enrich the components as basic software building blocks with assumptions, and develop tool supporting automated extraction of assumptions from existing system or ongoing system development, automated analysis of the assumptions revealing early defects in the architecture and providing early feedback about the design.

References

1. Garlan, D., Allen, R., Ockerbloom, J.: Architectural mismatch: Why reuse is so hard. IEEE Software **12**(6) (1995) 17–26
2. Rogers, W.P.: Report of the presidential commission on the space shuttle challenger accident. Technical report, U.S. Government Accounting Office, Washington, D.C. (1986)
3. Board, I.: ARIANE 5 – flight 501 failure. Technical report, European Space Agency, Paris, France (July 1996)
4. Al-Mamum, M.A., Hansson, J.: Review and challenges of assumptions in software development. In: Proc. of the Second Analytic Virtual Integration of Cyber-Physical Systems Workshop (AVICPS). (2011)
5. Tirumala, A.S.: An assumptions management framework for systems software. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA (2006)
6. Lewis, G.A., Mahatham, T., Wrage, L.: Assumptions management in software development. Technical Report CMU/SEI-2004-TN-021, DTIC Document (2004)
7. Lago, P., van Vliet, H.: Explicit assumptions enrich architectural models. In Roman, G.C., Griswold, W.G., Nuseibeh, B., eds.: 27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA, ACM (2005) 206–214
8. Ordibehesht, H.: Explicating critical assumptions in software architectures using AADL. Master's thesis, University of Gothenburg, Gothenburg, Sweden (2010)
9. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2) (April 2002) 256–290
10. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The architecture analysis & design language (AADL): an introduction. Technical Report CMU/SEI-2006-TN-011, DTIC Document (2006)
11. Lago, P., van Vliet, H.: Explicit assumptions enrich architectural models. In: Proceedings of the 27th international conference on Software engineering. ICSE '05, New York, NY, USA, ACM (2005) 206–214

12. Dewar, J.A., Builder, C.H., Hix, W.M., Levin, M.H.: Assumption-Based planning; a planning tool for very uncertain times. Technical report (1993)
13. Dewar, J.A.: Assumption-Based Planning: A Tool for Reducing Avoidable Surprises. Cambridge University Press (October 2002)
14. Steingruebl, A., Peterson, G.: Software assumptions lead to preventable errors. Security Privacy, IEEE **7**(4) (August 2009) 84 –87
15. Spiegel, M., Reynolds, P.F., J., Brogan, D.: A case study of model context for simulation composability and reusability. In: Simulation Conference, 2005 Proceedings of the Winter. (December 2005) 8 pp.
16. King, R.D., Turnitsa, C.D.: The landscape of assumptions. In: Proceedings of the 2008 Spring simulation multiconference. SpringSim '08, San Diego, CA, USA, Society for Computer Simulation International (2008) 81–88

A Appendix: Complete Model in Alloy

```
abstract sig ArchitecturalElement
{}

abstract sig Component extends ArchitecturalElement
{
   subcomponent: set Component,
   connected_to: set Connector,
   communication_protocol: set CommunicationProtocol,
   communication_data_model: set DataModel
}

abstract sig Connector extends ArchitecturalElement
{
   connected_to: set Component
}

fact about_connector
{
   all cn:Connector | cn.connected_to != none
   all cn:Connector | #cn.connected_to =< 2
}

fact about_component_connector
{
  all cm: Component | all cn: Connector | (cn in cm.
      connected_to implies cm in cn.connected_to) &&
  (cm in cn.connected_to implies cn in cm.connected_to )
}

//----------- Communication Protocols ---------------
abstract sig CommunicationProtocol extends
    ArchitecturalElement
{
```

```
    compatibility: set CommunicationProtocol
}
abstract sig EventLoop extends CommunicationProtocol
{}

one sig XIntrinsic extends EventLoop
{}
{
   compatibility = XIntrinsic
}

one sig OBA_Xlib extends EventLoop
{}
{
   compatibility = OBA_Xlib
}

one sig MachMessage extends EventLoop
{}
{
   compatibility = MachMessage
}

one sig BroadcastNotifyStatus extends CommunicationProtocol
{}
{
   compatibility = BroadcastNotifyStatus
}

one sig RequestReplyPair extends CommunicationProtocol
{}
{
   compatibility = RequestReplyPair
}

//---------------- DataModel -----------------------
abstract sig DataModel extends ArchitecturalElement
{
   compatibility: set DataModel
}

one sig CBasedModel extends DataModel
{}
{
   compatibility = CBasedModel
}

one sig ASCIIString extends DataModel
{}
{
```

```
   compatibility = ASCIIString
}

//———————————————————— Connectors ———————————————————
one sig CN1 extends Connector
{}
{
  connected_to =OBST + SoftBench
}

one sig CN2 extends Connector
{}
{
  connected_to =  Interviews + SoftBench
}

one sig CN4 extends Connector
{}
{
   connected_to = Interviews + MIG
}

//—————————————————— Components ———————————————————
one sig XLibrary extends Component
{}
{
   connected_to = none
   subcomponent = none
   communication_protocol = none
   communication_data_model = none
}

one sig OBST extends Component
{}
{
    connected_to = CN1
    subcomponent = none
    communication_protocol = none
    communication_data_model = none
}

one sig Interviews extends Component
{}
{
   connected_to = CN2 + CN4
   subcomponent = XLibrary
   communication_protocol=  OBA_Xlib
   communication_data_model = none
}
```

```
one sig MIG extends Component
{}
{
   connected_to = CN4
   subcomponent = none
   communication_protocol = MachMessage
   communication_data_model = CBasedModel
}

one sig SoftBench extends Component
{ }
{
   connected_to = CN1 + CN2
   subcomponent = XLibrary
   communication_protocol = XIntrinsic + BroadcastNotifyStatus
       + RequestReplyPair
   communication_data_model = ASCIIString
}

//--------- Assumptions as Assertions ---------------
assert Assm_NoCom_Infra
{
   all cm: Component | one sb: SoftBench | one xl: XLibrary |
   sb in cm.connected_to.connected_to  implies  cm.
       subcomponent = xl
}

assert Assm_NoCom_CM
{
   all cm1, cm2: Component | all el: EventLoop |
   ((cm1 in cm2.connected_to.connected_to-cm2 || cm2 in cm1.
       connected_to.connected_to-cm1) &&
   (el in cm1.communication_protocol || el in cm2.
       communication_protocol))  implies
   (el in cm1.communication_protocol.compatibility  && el in
       cm2.communication_protocol.compatibility)
}

assert Assm_NoConn_Protocol
{
   some cp: CommunicationProtocol | cp in
       BroadcastNotifyStatus.compatibility && cp in
       RequestReplyPair.compatibility
}

assert Assm_NoConn_DM
{
   all cm1, cm2: Component | all dm: DataModel |
   ((cm1 in cm2.connected_to.connected_to-cm2 || cm2 in cm1.
       connected_to.connected_to-cm1) &&
```

```
    (dm in cm1.communication_data_model || dm in cm2.
        communication_data_model)) implies
    (dm in cm1.communication_data_model.compatibility && dm in
         cm2.communication_data_model.compatibility)
}

assert Assm_GAS
{
    all cm: (Component - OBST) | ((cm.connected_to).
        connected_to - cm - OBST) = none
}

//----------------- Checks ------------------------
check Assm_NoCom_Infra
check Assm_NoCom_CM
check Assm_NoConn_Protocol
check Assm_NoConn_DM
check Assm_GAS

//------------------------------------------------
pred show () {}

run show
```

Listing 1.7. Complete model in Alloy