Modeling Constraints Improves Software Architecture Design Reasoning

Antony Tang
Swinburne University of Technology,
Melbourne, Australia
atang@swin.edu.au

Hans van Vliet
VU University,
Amsterdam, The Netherlands
hans@cs.vu.nl

Abstract

Requirements and project-related factors influence architectural design in intricate and multivariate ways. We are only beginning to understand some of the tacit but fundamental mechanisms involved in reasoning with design decisions, and one of them concerns the role of design constraints. To improve our understanding of this subject, this paper examines design constraints and how they shape design solutions. We introduce a design constraint model and an architectural design reasoning process for specifying design constraints and checking for design conflicts. In order to demonstrate that automated checking of constraint fulfillment can help verify the design, we apply first-order relational logic and the Alloy tool to an industrial example.

1. Introduction

Designing software architecture requires interpreting user requirements, non-functional requirements and considering technological and project factors. In order to create a workable solution all of these factors must be considered carefully and the design issues must be resolved satisfactorily. Architectural design is complex because of the diversity of these influencing factors and the number of design issues arising from their combinations. Typically, software architects rely on their knowledge, creative thinking and experience to deal with architectural design issues. The results and the quality of the architectural design vary depending on the architects. It is desirable to have a systematic design reasoning process to support software architecture design and to assure its quality.

Functional requirements define the scope of a system and as such narrows down the available design choices. Similarly, non-functional requirements can dictate the available design options. For example, a high performance requirement can eliminate most solutions that cannot meet the required performance level. Other factors, for instance the time and budget for an architect to develop a system, would also influences what architectural choices are available. These factors work in aggregation to constrain or eliminate the options in the solution space.

If requirements and different kinds of influencing factors dictate what design options are viable and what not through constraints, then one of the design reasoning tactics would be to find those design options that *do* fulfill the required constraints. Such is the focus of this study. We address three issues relating to design constraints. First, we illustrate the nature of design constraint and the purpose for representing them as a first-class entity. Using industry examples, we demonstrate in Section 3 that implicit design constraints can lead to incomplete or incorrect designs.

Second, we classify architectural design constraints into four generic categories and use industrial cases to demonstrate their explicit modeling.

Third, we propose a constraint-based design reasoning methodology (Section 4) for checking design constraints fulfillment and guiding architects to backtrack their design decisions when design conflicts are detected. By representing design constraints formally with first order relational logic we can uncover inconsistencies in design reasoning. Although design constraints have been recognized as important it has not received systematic study, this paper is an attempt to change that.

2. Defining Design Constraints

Design constraint is such a common term that its definition appears to be trivial. A survey of software architects [1] has indicated that design constraint is the second most important design rationale architects

use for design reasoning. However, the ways they affect the architectural design problem space and solution space are barely studied.

2.1. Why Study Design Constraint?

The current software architecture design practice relies very much on the experience and knowhow of the designers. Research on design has found that experienced designers just "know" what problems and constraints have to be framed in order to create a viable solution [2]. As such the quality of the architectural design relies on the craft of a designer.

In a psychological experiment using Coherence Model of Cognitive Consistency (Co3), it is demonstrated that the coherence of decisions can be modeled by a process of constraint satisfaction [3]. It shows that constraint satisfaction plays a key role in the decision making process to connect individual input factors to a common outcome. Similarly in software architecture, experienced architects may intuitively eliminate unviable solutions and steer their decisions towards those viable solutions that meet the implicitly specified design constraints.

Ongoing studies of design reasoning and architectural knowledge management such as CORE [4] and design rationale techniques [5, 6] have been proposed to make use of design constraints but these works lack an in-depth analysis of the nature of design constraints and their role in design reasoning.

2.2. A Design Constraint Definition

Besnard and Lawrie suggest that design is a problem solving activity and has a cognitive perspective where designers should identify constraints from requirements and use them for seeking design options in a solution space [7]. They assert that constraints come from requirements and therefore one of the design activities is to explicitly identify design constraints from requirements.

But requirements are not the only things that constrain design options in the solution space. There are many factors that can constrain what can viably be achieved in a design. Let us consider an example. McBride reports that business users require software deliveries once every three months in one organization [8]. This goal places an overriding constraint on what the development teams can produce within the time frame. It can affect development projects and make them unviable or it can force them to sacrifice product quality in favor of completing a product in time. Some of these constraints are tacit and they are never explicitly stated in either the requirement or technical design specifications.

Since design constraints can arise from different sources in addition to requirements, we need to have a general **definition** that extends beyond traditional requirement engineering: *A design constraint is a limiting factor which specifies the conditions that a viable design solution must fulfill.*

We also need a general architectural design constraint **principle**: *To have any viable design solutions, all design constraints pertaining to the related design decisions must be fulfilled.* As design constraints bound the solution space, they must be identified them and checked in order to create a viable software architecture design.

2.3. Design Constraint Identification

Design constraints can come in different forms and can influence design decisions in different ways. In the previous example [8], the *time constraint* comes from a business requirement. There exist other types of constraint such as development resources, technical feasibility, usability and performance. We group these constraints into four categories:

- **Requirement Related Constraints** - limiting factors from functional requirements that define the scope of a solution design.
- **Quality Requirement Related Constraints** - limiting factors from quality requirements that define the quality of a solution, e.g. performance and availability.
- **Contextual Constraints** – limiting factors that have bearings on the environment for constructing a solution, e.g. project context such as costs and schedule, technology context such as what technology platforms are mandated.
- **Solution-related Constraints** - limiting factors that arise during the design process. They come from technical limitations imposed by the chosen design components, e.g. the interface format of a software component.

The first three categories of constraints are mostly dictated by the environment. Constraints from functional requirements are decided and given by the users; non-functional requirements constraints come from interpreting the quality attributes; contextual constraints arise from the project and business environments.

Solution-related constraints are those constraints that are identified as the solution is being developed. When a design decision is made to employ a certain design component, the design component implicitly constrains how the rest of the system can be

designed. For instance, if a distributed architectural model is chosen for an application, then it constrains how transaction processing may be carried out in order to maintain transaction integrity. Some of these design constraints are obscure and their impacts and restrictions on the design cannot be easily recognized until much later in the development cycle or after the system is deployed.

2.4. Design Constraint and Design Process

Design constraints can play a guiding role during the design process. At any point in the design process, the set of design constraints determine what design actions are viable, in the following ways:

Eliminating unviable design options. One practical use of identifying design constraints is to use them to eliminate unviable design options from the solution space .

Determining design conflict. If there is no design solution that can satisfy all of the relevant design constraints, then a design conflict has been identified. In order to resolve a design conflict, some of the constraints must be relaxed or compromised, for instance by changing a *requirement* or relaxing a *contextual constraint*. This is when design trade-offs and stakeholders negotiation are applicable.

A general principle for evaluating design options. Using design constraints provides a general way to allow different types of limitations on the solution space to be evaluated in combination. For instance, if project cost and requirements are represented as constraints, then the possible design options in the solution space can be evaluated against the combined constraints.

Assessing evolving constraints. Design constraints evolve as the architectural design progresses from initial development to maintenance phases. There are two common scenarios: first, new and evolving requirements impose additional constraints on the system; second, as new design components are added or integrated to a system, their constraints may raise new design conflicts. As these design constraints continue to develop and change, they may impact on areas of design that are thought to be complete and stable. Therefore, examining new constraints against previous designs for potential design conflicts become necessary.

3. Implicit Constraints Obstruct Design Reasoning

Design constraints originate from different sources and sometimes are not stated explicitly. Experienced designers seem to be able to recognize them and cater for them intuitively [2]. However, when design constraints are not stated explicitly, their impact on a design may remain unnoticed for too long. This holds especially for complex designs, or designs that involve multiple architects.

We use real-life cases to illustrate some of the negative impacts of implicit design constraints. This industrial case is from a software system that is built for a car manufacturer to test vehicles. Development of the system took 2 years to complete and the system is now used by engineers in Australia and the U.S.A. The system is a web-based system that allows users to specify their test specifications, and XML documents are generated as an interchange format to instruct test vehicles. From this industrial case, we have identified several constraint related issues:

(1) Implicit Constraints. Designers had to choose a method to store XML data in an Oracle database. There were two storage options: (a) using a Character Large Object (CLOB) or (b) using a XML-Schema-based XML Type. Apart from the requirement of representing the schema in XML, there were no explicit specifications of any other constraint on the selection of the design. For instance, modifiability of the XML schema is implicit and not articulated.

When an update is made to the XML document using the CLOB method, one cannot make a change to an XML element without updating the entire XML document. On the other hand, the XML-schema-based method has a high maintenance overhead because of data synchronization issues.

Not recognizing that modifiability is an implicit constraint, the designer chose option b. When they had to repeatedly update the schema definition and found themselves having to spend a lot of time on maintenance, they recognized that the modifiability constraint should be one of the driving forces of this design decision. With this constraint explicitly specified, the designers reverted to option a, because it satisfies both the *functional requirement constraint* and the *quality requirement constraint*.

(2) Evolving Constraints. A design was created to store server-side web session information in memory. The physical memory usage from the initial design was 1KB to store the user-privilege information, and the memory usage is an insignificant constraint that was not documented explicitly. A design change was later made to keep transactional information in-memory, resulting in a significant increase for physical memory (10MB) for each user session. The additional demand for memory further constrains the architectural design especially the performance and

scalability of a system. The designer who changed the design was not the person who initially designed the system and was not aware of the potential implications. The new constraint has severely limited the number of users it could support unless more physical memory is added to relieve such a constraint.

(3) Conflicting Constraints. Design constraints imposed by different requirements may conflict with each other such that no solution is possible. There was a requirement where the users had to retrieve data through a user interface (i.e. a *requirement constraint*). Initially the amount of data to display online to the users was about 150KB, which was relatively small. The design decision was to choose a database solution to store the data and display them on request; this solution would satisfy both constraints. Later the amount of data increased to 1500MB due to a changing requirement. The solution could no longer display the data online to the users without an adverse effect on the user interface response time and usability. A design conflict has therefore arisen as there would be no solution that could fulfill the two constraints simultaneously, in this case processing volume versus response time and usability. Such conflicting constraints would create a situation in which one or more of the constraints would need to be relaxed in order to create a viable design solution.

These real-life examples demonstrate that unless constraints are explicitly specified and checked against a design, design reasoning may falter and some of the following situations may occur: (a) Imprecise or inadequate specification of architectural constraints may lead to a less optimal design solution; (b) Implicit design constraints may become apparent only when a design option is considered in detail. The discovery of new constraints can feed back to the design reasoning process; (c) Premature commitment to a design solution without considering all relevant constraints may remove viable architectural design options; (d) Conflicting constraints can make a design unviable but architects do not realize these conflicts until much later in the development life-cycle.

4. Design Constraint Modeling and Design Reasoning

From the above examples, it could be argued that design constraints are simply requirements. We suggest that constraint represent a different perspective, it highlights *how* the requirements bound the solution space. Additionally, it can represent the combined bounding effects of requirements and other influences on finding a solution.

Modeling of design decisions and design rationale have been proposed by various researchers [6, 9] and also recommended by the IEEE standard [10]. Constraint modeling can be built upon such prior work. Figure 1 depicts a design decision model in UML based on the causal relationships between design concerns and design outcomes [6]. This model captures *design concerns*, *design decisions* and *design options* and the causal relationships between them. *Design concerns* are the inputs of a design decision, they represent requirements and other factors that influence a design decision. One aspect of design concern's influences on a decision is how it constrains a design decision. In order to explicitly represent design constraints, we capture them as an attribute within a design concern.

A *decision* node is a point where an architect deals with a design concern and it contains design rationale as justification. The results of a design decision, i.e. *design options*, describe the chosen design and the rejected designs. A chosen design can in turn be a design concern if it acts as a design concern, i.e. an input, to another design decision.

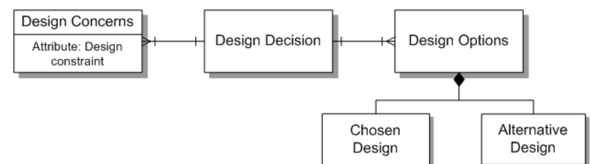

Figure 1 – A General Design Decision Model

Previous studies have modeled design constraint as a rationale without exploring how design constraints affect the design process [5, 11]. In the following sections, we show that explicitly representing design constraints and constraint fulfillment can enhance general design decision reasoning process.

4.1. Design Constraint Fulfillment

The explicit representation of design constraints can support design reasoning. It allows checking if a design option satisfies the constraints for different types of design concern. If the constraint(s) exerted by a design concern cannot be fulfilled by any design option, then some compromises on the design constraints become necessary. To allow such checking, design constraints need to be represented either qualitatively or quantitatively:

a. **Functional requirements (FR)** – qualitative design constraints.

b. **Non-functional** or **quality requirements (NFR)** – qualitative NFR, e.g. usability, modifiability; quantifiable NFR, e.g. performance, reliability.
c. **Project context** – qualitative context, e.g. software development standards; quantifiable context, e.g. budgeted cost and schedule for constructing a sub-system.
d. **Design components** – qualitative characteristics, e.g. behavior and limitations of a component; quantifiable characteristics, e.g. performance of a component.

For all design constraints, the following information can be modeled:

- **Description** – specify the constraint, e.g. *x >= 10000 tps*, meaning design option *x* must be able to process at least 10,000 transactions per second; or an example of a functional requirement: a file must be contained within a single directory in a file system.
- **Priority** – specify the importance of the constraint. The higher the scale, the less it can be compromised. 1 is least important and can be compromised, 5 is most important and non-negotiable. This information is used in tradeoff analysis and not in automated design reasoning.

If design constraints are systematically and explicitly stated, then design reasoning can be a process of finding those design options that satisfy the stated constraints. In other words, when a design option is chosen all the design constraints that influence the decision must be fulfilled. The *constraint fulfillment* of each design option is characterized by the stated design constraints of a decision. For instance, to satisfy a minimum tps design constraint, a viable design option must have design fulfillment of *x >10000tps*.

The characterization of the behavior of design options using *design constraint fulfillment* allows us to check if the *design constraints* are fulfilled. Design constraint fulfillment is specified as follows:

- **Design constraint fulfillment** – specify the behaviors of a design, e.g. (*x >= 11000tps and x <= 12000tps*).
- **Certainty scale** – note the risk level of a design option. Scale is 1 to 5 where 1 is totally uncertain that the design would work and 5 is totally certain that the design would work. This is an indication if further decisions need to be explored to refine the solution [12].

4.2. Modeling Design Constraints

In this section, we use the same case study to illustrate how design constraints are modeled and how such modeling supports explicit design reasoning. We have selected a real-life example of an evolving requirement to illustrate how explicit modeling of design constraints can work in a design reasoning process.

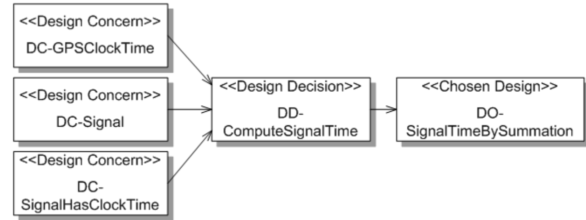

Figure 2 – A Constraint Modeling Example

In this system, monitoring data is collected from a Vehicle Data Logger (VDL) installed in a vehicle. The VDL would stamp each data reading, also called *signals*, with a clock-tick which is relative to a regularly transmitted Coordinated Universal Time (UTC) based on the GPS system. The clock time of the signals are computed by adding a clock-tick to the last GPS clock time (see Fig. 2).

The example in Fig. 2 and Table 1 illustrates the design concerns used in the decision process. Using the three design concerns, a decision was made to create a design object which calculates the UTC time for each signal. During the design process, the design concerns were explicitly represented as requirements and design context, however the design constraints remained implicit. When we explicitly model these constraints, we notice that specific conditions that are required to guide the design are missing from the design concerns. Consider *Design Concern 1* for instance, which states that the GPS clock-time must be available periodically. This outlines what is needed, but it does not specify how it might affect the design. On the other hand, if we consider how it constrains the design, we note that the clock time must be in a specific format, in this case UTC format, and this reference time must be present before computing the time of any data readings or signals.

In terms of stating a design concern, a description of the requirement or context may not be enough for design reasoning. How a requirement or a context influences a design by way of *constraining* a prospective design is also important. Design constraints thus provide a design reasoning perspective to elicit additional design information.

After designing and developing the software, the firmware designer noticed that the GPS clock time is not available immediately when the VDL unit is

switched on and there is a delay before the GPS component can obtain the UTC clock time. During that period, some signals would arrive and they would be without the reference GPS time. Additionally, this was discovered only days before the software was due for user acceptance testing, so there was only a limited time to fix the problem, thus *Design Concern 4* dictates what can be done within the time frame. Table 2 illustrates the additional constraints to the decision making process.

Design Concern 1 - DC-GPSClockTime	
Description:	A GPS clock time is available periodically.
Constraint:	(A) GPS clock time in UTC format should be available prior to any data readings or signals.
Design Concern 2 - DC-Signal	
Description:	A signal contains the clock tick relative to the previous GPS clock time.
Constraints:	(A) Clock tick is available in each signal.
Design Concern 3 - DC-SignalHasClockTime	
Description:	A signal should be reported with UTC clock time.
Constraint:	(A) Computation of a signal clock time requires the presence of signal clocktick relative to the GPS clocktime.
Design Decision - DD-ComputeSignalTime	
Description:	How to compute the clock time of a signal?
Chosen Design Component 1 - CD-SignalTimeBySummation	
Description:	Compute the clock time of a signal based on the summation of the GPS clock time and the clock ticks in a signal.
Fulfillment:	(A) Compute *SignalTime = GPS clock time + clock ticks in signal*

Table 1 – Design Constraint Example

Design Concern 1 - DC-GPSClockTime	
Description:	A GPS clock time is available periodically.
Constraints:	(A) A GPS clock time in UTC format should be available prior to any data readings or signals. (B) Due to hardware constraints, the first GPS clock time will arrive about 3 seconds after the unit is switched on, and arrive later than some signals.
Design Concern 4 - DC-MeetProjectSchedule	
Description:	Software must be delivered according to the planned project schedule.
Constraint:	(A) Acceptance testing must be completed on time according to project schedule.

Table 2 – Additional Design Constraints

With the additional constraints, the existing design outcome *CD-SignalTimeBySummation* is no longer workable because it cannot fulfill constraint *B* of *DC-GPSClockTime* and constraint *A* of *DC-SignalHasClockTime* simultaneously. So new design options from the solution space must be considered. The first design option is to create a special signal in the firmware that allows the software to calculate when the first GPS clock time should have been based on a clocktick that counts from power-on to the first GPS timestamp. Such implementation would take time to investigate and implement. This design option would fail the delivery constraint *A* of *Design Concern 4*.

Chosen Design Component 1 - CD-SignalTimeBySummation	
Description:	Compute the clock time of a signal based on the summation of the GPS clock time and the clock ticks in a signal.
Fulfillment:	(A) If GPS clock time is available, compute *SignalTime = GPS clock time + clock ticks in signal*
Chosen Design Component 2 - CD-ComputeSignalTimeByTimeMarker	
Description:	When GPS clock time is not yet available, buffer all signals until a Time Stamp Marker and a *GPS clock time* become available.
Fulfillment:	(A) For signals where a *GPS clock time* is not available, buffer all interim signals (B) When *Time Stamp Marker* and *GPS clock time* become available:- compute *FirstGPSClockTime* = GPS clock time - *Time Stamp Marker;* using the *FirstGPSClockTime*, compute time of the buffered interim signals by :- *SignalTime = FirstGPSClockTime + clock ticks in signal*

Table 3 – Design Outcomes and Design Constraint Fulfillment

The second design option is a temporary measure to rectify the current problem. It assumes that the time between unit power-up and the first GPS signal arrival is a 3 second period, and use that to recalculate the clock time for each signal before the first GPS timestamp. This design option does not fulfill the constraint of *Design Concern 3* because of its inaccuracy.

Neither of the two design options can fully fulfill all the design constraints, yet there is no other design option available. We call this situation *conflicting constraints*. In order to resolve the conflicting constraints, one or more of the design concerns and their design constraints have to be compromised or relaxed. The only compromisable constraint is in *Design Concern 3*.

If the constraint in this design concern is relaxed to allow "*Computation of a signal clock time is based on its clocktick relative to an estimated GPS clocktime*", then we have a viable design option. In this design, the initial signals are approximated based on the 3 seconds estimate until the accurate GPS clocktime arrives.

Subsequent to this interim solution, the constraint in *Design Concern 4* was no longer applicable because the project deadline had passed. With the removal of this constraint, designers can revisit the design decision to find a more suitable solution. First, the constraint in *Design Concern 3* is reverted back to its original form since the relaxed requirements are no longer applicable. Second, additional design components are created to satisfy the constraints. As

illustrated in Table 3, the design component *CD-ComputeSignalTimeByTimeMarker* is added to the solution. This solution provides a time marker which marks the number of clock-ticks from when the VDL is switched on until the first GPS clock time arrives. As such, the missing GPS clock time can be retrospectively computed. Together with the modified design component *CD-SignalTimeBySummation*, all the constraints within the design concerns can be fulfilled.

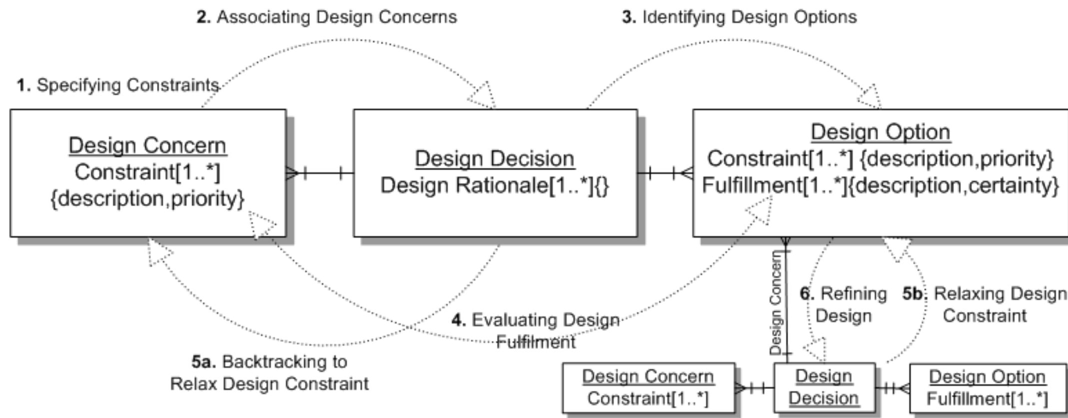

Figure 3 – Constraint-based Design Reasoning Process

4.3. Constraints-driven Design Reasoning Process

Using the previous industrial case as an illustration, we have observed that (a) design constraints provide specific information for design reasoning; (b) there are different kinds of design constraint deriving from requirements and project context; (c) design constraint representation can highlight constraint conflicts when no solution fulfills all related constraints. Through checking constraint fulfillment, designers can identify the design concerns that are conflicting and pinpoint areas where the constraints may be relaxed or compromised. In this section, we propose a constraint-driven design reasoning process to achieve this.

1. **Specifying design constraints** - specify *design constraints* for each design concern. Each design concern may have 1 or more *design constraints*.
2. **Associating design concerns to a decision** – to structure a design decision [13], related design concerns are identified and associated to a decision. For instance, if the design is about the application achieving certain database performance, the relevant design concerns would be: (a) data volume of an application; (b) the performance requirements; (c) designated hardware platform. Design constraints of these design concerns are evaluated at the decision point.
3. **Identifying design options** - at each decision point, identify possible design options that satisfy the design concerns and their constraints. During this step, architects specify the *design fulfillment* for each design option. A design option contains a set of ***constraint fulfillments*** that represents how a design option satisfies the design constraints exerted upon it. A design option can also influence another decision, as such it puts ***constraints*** on those design that it influences (see *Design Option* box in Fig. 3).
4. **Evaluating design constraint fulfillment** - for each design option, evaluate if a design option can fulfill all the *design constraints* by comparing *Design Option Fulfillment* with *Design Constraint.* During the iterative evaluation process, new design constraints may be uncovered. The possible scenarios of constraint fulfillment are:
   - If there is only <u>one</u> design option that satisfies all specified *design constraints*, then this design option can be chosen in the decision.
   - If there are <u>multiple</u> design options that satisfy all the constraints, select a design option by assessing their relative pros and cons or using some tradeoff analysis method such as [14, 15]. The *certainty scale* of a potential solution can help designers assess the risk of implementation.
   - If <u>no</u> design option can satisfy all the design constraints, then a constraint conflict has arisen since there are no viable solutions to

solve the design problem. The only way is to relax one or more design constraints and seek alternative design options (see step 5). Tradeoff analysis methods can be used at this point. The *priority* of a design constraint can help designers judge which constraint to relax.

5. **Relaxing design constraints** – when no design solutions can be found to satisfy a set of constraints at a decision point, one or more design constraints must be relaxed. To do so, architects should traverse backwards to the design concerns to relax their constraints (step 5a in Fig. 3). When a design constraint is associated with a requirement or a project context that has no parent nodes, the design constraint relaxation cannot proceed any further, architects would have to negotiate these externally imposed constraints with the relevant stakeholders in order to have them relaxed. If the design constraint to be relaxed is itself a design outcome, then changing it implies reconsidering all previous decisions that have led to this design outcome (see step 5b in Fig. 3). In this case, backtracking of design reasoning involves going through the series of decisions and design constraints to ensure that the design constraints of all related design concerns can be fulfilled.
6. **Refining design** - when the *certainty scale* of a design option is low, i.e. a designer is not sure if the chosen design option would work, then further exploration is needed. A designer can repeat steps 1 to 4 to decompose the design to provide more design details until the behavior of the design options are well understood and the certainty of a design is acceptable. In this case, the *chosen design option* would influence lower level design decisions through newly formed constraints.

4.4. Automated Checking of Design Constraint Fulfillment

Until this point, we have theorized that design constraint fulfillment can help design reasoning. We have provided an explicit yet informal model to represent constraint-based design reasoning and a process to support design reasoning. However, the checking of constraint fulfillment based on human interpretation may be error prone. The design solution may not logically fulfill all the constraints. Therefore, it would be desirable to verify the fulfillment of design constraints with a stringent and automated method. We propose to use first-order relational logic to this end. The following example shows the feasibility of applying such a method.

Firstly, we represent design constraints as *sets* of *elements* and we define the relationships between them, e.g., a *Signal* is a data record containing *clockticks* and *relativeTime*. Secondly, we define the predicates to show *how* the constraints are implemented. Finally, we check if the constraints are violated by the implemented solution.

We make use of the Alloy tool to automatically check constraint fulfillment [16]. In Alloy, the signature (*sig*) section defines the design constraints and the predicate (*pred*) section defines how constraints are used by a design object. The following example represents the constraints and the design outcome described in Table 1:

```
/* Design Concern 1 Constraint A - GPS clocks will
be available and in ascending clock time order */
fact CD_GPSClockTimeInAscendingOrder {
        all g: GPSClock - max[GPSClock] | let g' =
        nextGPSClock[g] | g'.clocktime >=
        g.clocktime }

/*  Design concern 2 Constraint A – clocktick is
available in each signal */
sig Signal extends DataRecord {
        clockticks : one Int,
        gpsClock : GPSClock lone -> File,
         relativeTime : Int lone -> Int -> File }

/* Design Component 1 fulfillment –
     DO-SignalTimeBySummation */
pred DO_SignalTimeBySummation[f, f': File, s: Signal] {
        let prevClocks = prevs[s] & GPSClock |
                s.gpsClock.f' = max[prevClocks]
        s.relativeTime.f' = (s.gpsClock.f'.clocktime)->
                s.clockticks }
```

The design constraints in Alloy specify that the *clocktick* is an integer and is relative to a *gpsClock*. In the design solution, the *SignalTime* is calculated by summing the *clockticks* of a *Signal* to the last *GPSClock*. In order to analyze if the constraints can be fulfilled logically, we check the following assertion to see if all signal's *relativeTime* can be calculated by summing their *clockticks* with the preceding *GPSClock.clocktime*.

```
/* Check all signals have a Clock Time */
assert DC_SignalHasClockTime {
        all s: Signal, f: File | let f' = f.next |
          DO_SignalTimeBySummation[f, f', s] =>
                one s.relativeTime.f' }
check DC_SignalHasClockTime for 2 GPSClock,  3
Signal, 2 File
```

An *assertion* acts like a regression test, it tries to discover if there are any scenarios, called *counterexamples*, where the the constraints cannot be fulfilled by the implemented solution.

In this case, Alloy has discovered a counter example. This counter example describes that when the GPS clock time is not available before the arrival of signals, computation cannot proceed. It illustrates that one of the constraints is missing. That is the system does not have a behavior which guarantees that a GPS clock time would arrive before any signals. The Alloy checker has found a logical inconsistency in this design.

Separately, the designer also discovered the same situation through system testing (see *Design Concern 1 - DC-GPSClockTime* constraint *B* in Table 2). This issue was rectified eventually by adding *Component 2 to satisfy the constraint* (see Table 3). When we run Alloy again after adding the constraint and the component, all design constraints are satisfied and no design inconsistency or counter examples are found. This example has demonstrated that formal constraint specification and fulfillment checking can provide a way to validate constraint fulfillment of an architectural design. It can complement human design reasoning process to check for mistakes.

Limitations of Automated Checking. Although we have shown that design constraints and their fulfillment can be analyzed automatically, its implementation is localized to each decision point. Constraint model checking over a chain of related decisions will be a topic of continued research. An automated design decision checking is possible but design conflict resolution relies on human reasoning. Additionally, the specification and application of first-order relational logic is not user-friendly and designers may not find the benefits worthwhile for the time it takes them. These two areas need to be further investigated if an automated design constraint fulfillment method is to be applied effectively.

5. Related Work

The concept of design constraint is currently not well-defined for architecture design reasoning. Roeller et al. suggest that it is tricky to draw the line between assumptions, requirements and constraints [17]. Design rationale systems such as DRL [18] have implicitly provided some support for constraint modeling but they have not explicitly modeled the relationship between constraints and the solution space. Herold et al. suggest that architectural constraints prohibit some requirements to be realized and therefore requirements need to be renegotiated [19]. Kruchten describes constraints as a relationship constraining the properties or the existence between decisions [20].

Dardenne et al. define constraints as an operational objective to be achieved by the composite system [21]. They argue that a constraint definition can lead to the identification of new objects and actions involved in a constraint. KAOS aims at decomposing *goals* into *actionable objects*. They focus on functional requirements decomposition. Similarly, Feather et al. consider constraint as an implementable goal [22]. These approaches differ from our approach where we focus on using constraints as a key criterion in design decision making to check for viable design solutions. On the other hand, Chung et al. [23] describe NFR goal decomposition in their work using constraint to describe the generation of subclasses.

In [24], Crnkovic et al. have suggested that properties of goals can be decomposed to end up with a set of quantifiable quality attribute (QA) properties which then can be satisfied by design components. Such QA properties can characterize the constraints QA put on an architecture design. The ATAM method considers sensitivity points where tradeoffs are made [15]. This has similarities to detecting conflicting design illustrated in this paper except that ATAM does not model design reasoning.

Constraints in software architecture design can be represented by rules that govern the design at the component and connector level [25, 26]. These rules provide a list of `shalls' to tell programmers what they must do in order to comply with the architecture. Specialized constraint languages such as Core Constraint Language (CCL) and Design Constraint Language (CDL) can specify design constraints at the programming level [27, 28]. Object Constraint Language (OMG-OCL) can specify invariants for classes and types, and to specify constraints on operations [29]. Our definition of constraint has a broader context than what these languages represent and therefore we have chosen to use first-order relational logic instead.

6. Conclusion

In this paper, we discuss design constraint modeling and using it for checking design solution fulfillment. When design constraints are not explicitly and systematically documented, it obstructs design reasoning. We have used an industrial case study to illustrate that such implicit design constraints can have negative consequences to an architecture design.

To alleviate this problem, we propose to model design constraint as a first-class entity.

Design constraint characterizes how requirements and design concerns influence design selection from the solution space. This characteristic can be used to verify solution fulfillment by considering design solutions that satisfy all the constraints. On this basis, we propose a constraint-based design reasoning process that would help architectural design reasoning and choosing viable design solutions.

Designers' reasoning about design constraint can sometimes be absent or erroneous. We have shown that it is possible to automatically check architectural design constraint fulfillment using first-order relational logic and the Alloy tool. This method complements the design reasoning process and helps detect logical design errors when constraints are unfulfilled or absent.

7. References

[1] A. Tang, M.A. Barbar, I. Gorton, and J. Han, "A survey of architecture design rationale," *Journal of Systems and Software,* vol. 79 (12), pp. 1792-1804, 2006.

[2] N. Cross, "Creative Thinking by Expert Designers," *The Journal of Design Research,* vol. 4 (3), 2004.

[3] K. J. Holyoak and D. Simon, "Bidirectional Reasoning in Decision Making by Constraint Satisfaction," *Journal of Experimental Psychology: General,* vol. 128 (1), pp. 3-31, 1999.

[4] R. C. de Boer, R. Farenhorst, P. Lago, H. van Vliet, V. Clerc, and A. Jansen, "Architectural Knowledge: Getting to the Core," in *3rd International Conference on the Quality of Software Architectures (QoSA)*, 2007.

[5] J. Tyree and A. Akerman, "Architecture Decisions: Demystifying Architecture," *IEEE SOFTWARE,* vol. 22 (2), pp. 19-27, 2005.

[6] A. Tang, Y. Jin, and J. Han, "A rationale-based architecture model for design traceability and reasoning," *Journal of Systems and Software,* vol. 80 (6), pp. 918-934, 2007.

[7] D. Besnard and A. T. Lawrie, "Lessons from industrial design for software engineering through constraints identification, solution space optimisation and reuse," in *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC)*, 2002, pp. 732-738.

[8] M. R. McBride, "The software architect.," *Communications of the ACM,* vol. 50 (5), pp. 75–81, 2007.

[9] A. Jansen and J. Bosch, "Software Architecture as a Set of Architectural Design Decisions," in *Proceedings 5th IEEE/IFIP Working Conference on Software Architecture*, 2005, pp. 109-120.

[10] IEEE, "IEEE Recommended Practice for Architecture Description of Software-Intensive System (IEEE Std 1471-2000)," IEEE Computer Society 2000.

[11] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford, *Documenting Software Architectures : Views and Beyond*, Boston ed. Addison Wesley, 2002.

[12] A. Tang and J. Han, "Architecture Rationalization: a Methodology for Architecture Verifiability, Traceability and Completeness," in *12th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems ECBS 2005*, U.S.A., 2005, pp. 135-144.

[13] C. Zannier, M. Chiasson, and F. Maurer, "A model of design decision making based on empirical results of interviews with software designers," *Information and Software Technology,* vol. 49 (6), pp. 637-653, 2007.

[14] T. Al-Naeem, I. Gorton, M. A. Babar, F. Rabhi, and B. Benatallah, "A quality-driven systematic approach for architecting distributed software applications," in *Proceedings. 27th International Conference on Software Engineering (ICSE 2005)*, 2005, pp. 244-253.

[15] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The architecture tradeoff analysis method," in *Proceedings of the Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '98)*, 1998, pp. 68-78.

[16] D. Jackson, "Automating First-Order Relational Logic," in *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering*, 2000, pp. 130-139.

[17] R. Roeller, P. Lago, and H. van Vliet, "Recovering architectural assumptions," *Journal of Systems and Software,* vol. 79 (4), pp. 552-573, 2006/4 2006.

[18] J. Lee and K. Lai, "What is Design Rationale?," in *Design Rationale - Concepts, Techniques, and Use*, T. Moran and J. Carroll, Eds. New Jersey: Lawrence Erlbaum, 1996, pp. 21-51.

[19] S. Herold, A. Metzger, A. Rausch, and H. Stallbaum, "Towards Bridging the Gap between Goal-Oriented Requirements Engineering and Compositional Architecture Development," in *Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent (SHARK-ADI '07)*, 2007.

[20] P. Kruchten, P. Lago, H. v. Vliet, and T. Wolf, "Building up and Exploiting Architectural Knowledge," in *5th Working IEEE/IFIP Conference on Software Architecture*, 2005.

[21] A. Dardenne, A. Van-Lamsweerde, and S. Fickas, "Goal-directed Requirements Acquisition," *Science of Computer Programming,* vol. 20 pp. 1-36, 1993.

[22] M. S. Feather, S. Fickas, A. v. Lamsweerde, and C. Ponsard, "Reconciling System Requirements and Runtime Behavior," in *Proceedings of the 9th international workshop on Software specification and design*, 1998, p. 50.

[23] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos, *Non-functional requirements in software engineering*. Boston. Kluwer Academic, 2000.

[24] I. Crnkovic, M. Larsson, and O. Preiss, "Concerning Predictability in Dependable Component-Based Systems: Classification of Quality Attributes," *Book on Architecting Dependable Systems III,* pp. 257-278, 2005.

[25] D. Garlan and M. Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering,* vol. 2 pp. 1- 39, 1993.

[26] J. S. Poulin, N. Kemerer, M. Freeman, T. Becker, K. Begbie, C. D'Allesandro, and C. Makarsky, "A reuse-based software architecture for management information systems," 1996, pp. 94-103.

[27] C. Tibermacine, R. Fleurquin, and S. Sadou, "Simplifying transformation of software architecture constraints," in *Proceedings of the 2006 ACM symposium on Applied computing (SAC'06)*, 2006, pp. 1240-1244.

[28] N. Klarlund, J. Koistinen, and M. I. Schwartzbach, "Formal design constraints," in *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '96)*, 1996, pp. 370-383.

[29] Object Management Group, "UML 2 Superstructure Final Adopted Specification," 2003.