Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 28 (2014) 555 – 564

Procedia Computer Science

Conference on Systems Engineering Research (CSER 2014)

Eds.: Azad M. Madni, University of Southern California; Barry Boehm, University of Southern California; Michael Sievers, Jet Propulsion Laboratory; Marilee Wheaton, The Aerospace Corporation
Redondo Beach, CA, March 21-22, 2014

A formal method for assessing architecture model and design maturity using domain-independent patterns

Kristin Giammarco[a]*

[a]*Department of Systems Engineering, Naval Postgraduate School, Monterey, CA 93943 USA*

Abstract

Design patterns have been used as a formal or systematic means for extracting and patterning knowledge about good design choices, as well as capturing lessons learned associated with poor design choices (or so-called anti-patterns). Yet little attention is devoted specifically to pattern languages that are based on the fabric of architecture models – the conceptual data model – to capture reusable design knowledge and architecting best practices that can be applied in more than one domain at a high level of abstraction. This paper demonstrates a simple model-based method for identifying and patterning architecture design aspects that are domain-independent, and thus transferable and reusable in any system design with a comparable data model. The use of this method in formally documenting good and poor patterns in an abstract way is demonstrated by example. Discovered patterns such as those presented herein can be distributed, codified in a tool of choice, and sought out in actual architecture models of systems using automation. Since there may not be universal agreement on a common set of "good" and "poor" patterns, individual architects or organizations can use this method to state their particular practices as formal axioms, and structure them to assist in the assessment of model and design maturity against their own specific standards.

Keywords: architecture; model; design; maturity; patterns; formal methods; Alloy, Lifecycle Modeling Language

---

* Corresponding author.
E-mail address: kmgiamma@nps.edu

1. Introduction

Researchers and practitioners continue to face a longstanding challenge throughout government and industry: to understand and predict architecture-level design flaws earlier, and thus relatively less expensively, compared with fixing them later in the lifecycle [1]. Methods for documenting architectural patterns that are desired or constructive ("good"), as well as architectural patterns that are undesired, disruptive or destructive ("poor"), assist the systems engineering and sciences communities with the exposure and transfer of reusable information. A simple model-based method for identifying and patterning architecture design aspects that are domain-independent, and thus transferable and reusable, is found at the intersection of the software and systems engineering fields, where formal methods are used to codify rules that apply in any system architecture model with a comparable data model. The nomenclature for the architecture data model used in this paper is provided in the box below. This nomenclature, which matches that found within the Lifecycle Modeling Language (LML) specification [2], provides a level of entity abstraction and decomposition that substantially simplifies the identification of architecture-level patterns through entities and their relationships.

Nomenclature

LML Class	*Description [2]*	*DoDAF [3]/ UPDM [4] Equivalent Class*
Asset	An object, person, or organization that performs Actions, such as a system, subsystem, component, or element.	Performer
Action	The mechanism by which inputs are transformed into outputs.	Activity
Input/Output	The information, data, or object input to, trigger, or output from an Action.	Resource
Conduit	The means for physically transporting Input/Output entities between Asset entities. It has limitations (attributes) of capability and latency.	Connector

2. Related work

The use of design patterns as a formal or systematic means to extract and manage knowledge about good and poor designs can be found throughout the literature. The languages of these patterns run the gamut from formal to informal/natural. Gamma et. al [5] introduce design patterns, or "reusable micro-architectures that contribute to an overall system architecture," as a means for abstracting and describing recurring themes that appear in many object oriented designs. Bosch [6] describes various kinds of structural and behavioral design patterns (e.g., Adapter, Bridge, Observer, Mediator) as object-oriented compositions that are free of implementation details and thus reusable in many designs calling for the functions or behaviors described in those patterns. The concept of design patterns has been applied in the systems engineering domain, for example Haskins' [7] pattern for "Multidisciplinary Teams," and Cloutier's [8] pattern for "Command and Control." Rebovich and DeRosa [9] capture two success patterns in information technology (IT)-intensive systems in a government acquisition environment, namely "Balancing the Supply Web" and "Harnessing Technical Complexity." There has also been work in capturing lessons learned associated with poor design choices, or anti-patterns [10], and patterns for system misuse [11]. Weilkiens [12] points out that graphical layouts for MBSE diagrams can be patterned, such as a tree diagram being presented in a top-down, bottom-up, or non-tree layout. In the system sciences community, 55 candidate systems processes have been identified for patterning at multiple levels of abstraction [13]. Giammarco [14,15,16] and Rodano and Giammarco [17] utilize an architecture data model similar to that presented in the nomenclature as a pattern language, and develop domain-independent patterns from that language. This work in pattern finding pertains to formally specifying and transferring knowledge about design practices and configurations that permeate architecture models of any system described using the LML or a compatible data model such as the DoDAF Meta Model (DM2) or the Unified Profile for DoDAF and MoDAF (UPDM).

3. Method

The method for formally documenting good and poor architecture-level design patterns is described and demonstrated, along with some example patterns that were identified while using the method. The research methodology that ultimately resulted in this formal method is documented in more detail in the author's doctoral research [14]. This method is a generalization and abbreviation of a formal method tailored specifically for interoperability assessment of a modeled architecture, also created as part of the author's doctoral research.

3.1. Identify the problem to be addressed

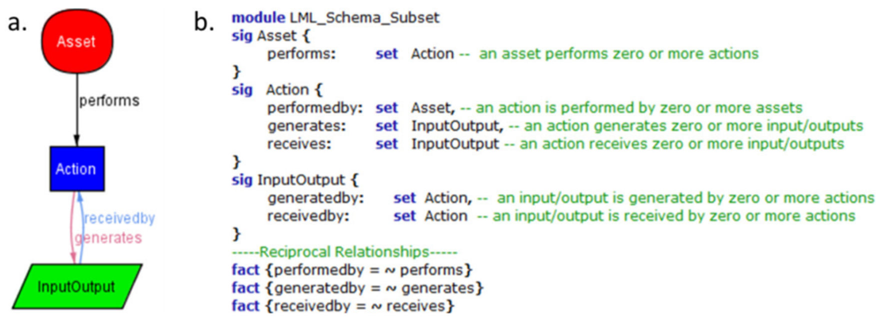
The analysis question of interest must first be identified to bound the scope of the effort. That is, what is it that the architect wants to know about the model? For the purposes of this demonstration, assume that the architect asks the following questions:

1. Are there any patterns present in the model that may indicate failure to implement tried and true best practices used by experienced architects?
2. Are there any patterns present that may create issues for the interoperability of the design described by the model?

Note the generality of both questions in the sense that these questions can be asked of any system under design, i.e., they are domain-independent questions. Also note that the questions are framed to expose problems or deficiencies in the model for correction. This is consistent with the architect's intent to correct known deficiencies early in design. The first question focuses on determining how well-formed [17,14] the model is, while the latter identifies a specific area of concern for the design described by the model (i.e., interoperability [14]). To answer these questions, the architect must define specifically what conditions would cause a deficiency in the well-formedness of the model, and what conditions would cause a deficiency in the interoperability of the design, respectively.

3.2. Model the data model subset pertinent to the problem

To keep the patterns domain-independent, the conditions identified by the architect are expressed in terms of architecture model entities, such as those in the LML nomenclature presented in the introduction, and relationships among those entities (i.e., a conceptual data model: the metamodel capturing general constraints on how elements of the design are related). Fig. 1(a) depicts a directed graph of allowable relationships from the LML specification, and Fig. 1(b) shows corresponding Alloy [18] code used to generate this metamodel. Only the first three classes: Asset, Action and Input/Output, are included in the initial scope for this demonstration. These classes were chosen due to their high relevance to the problem, as relationships among these entities have a high bearing on model form and interoperability of the design. The analysis is later extended with addition of the conduit class and its relationships.



```
module LML_Schema_Subset
sig Asset {
      performs:      set  Action -- an asset performs zero or more actions
}
sig  Action {
      performedby:  set  Asset, -- an action is performed by zero or more assets
      generates:    set  InputOutput, -- an action generates zero or more input/outputs
      receives:     set  InputOutput -- an action receives zero or more input/outputs
}
sig InputOutput {
      generatedby:     set  Action, --  an input/output is generated by zero or more actions
      receivedby:      set  Action  -- an input/output is received by zero or more actions
}
-----Reciprocal Relationships-----
fact {performedby = ~ performs}
fact {generatedby = ~ generates}
fact {receivedby = ~ receives}
```

Fig. 1. (a) Architecture model schema subset metamodel; (b) Architecture model schema subset Alloy code.

The Alloy language is a high level language that enables abstract modeling of software and system architecture structures [18]. As can be seen from the annotations, the code straightforwardly formalizes the data model subset to be examined for patterns. In this initial model, no constraints are present to restrict multiplicity (how many relationships are allowed between a given pair of entities).

3.3. Document constraints on the model that capture domain-independent patterns

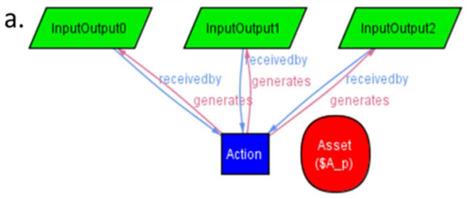
The Alloy code in Fig. 1(b) is executable in the Alloy Analyzer tool [19]. For example, the metamodel in Fig. 1(a) was generated automatically from the code in Fig. 1(b). The ability to execute this code is the basis of this formal method. Upon execution, the Alloy code generates many possible instances, showing various implementations of the entities and relationships as they might appear in architecture models using this schema. Thus, this approach provides the architect with an abstract way to reason about the schema of any architecture model without referring to specific architecture models.

The code in Fig. 1(b) permits the instance of data model relationships shown in Fig. 2(a). There are a number of model form issues with this example instance: the action in this scenario is not performed by any asset, the asset in this scenario is not performing any action, and the action appears to be looping the same three input/outputs. While it is conceivable that an action may generate and/or receive three input/outputs, the architect responsible for this design wishes to prevent any instances of "looping" input/outputs because (s)he interprets the definition of Action to transform inputs into *different* outputs. The rationale: if an input/output crosses a functional boundary, as is the case when an input/output leaves the action, it does not make sense to wrap it back around into precisely the same action that produced it. It may be preferable to show such loops at a lower level of decomposition between child actions for a more precise and stepwise description of the feedback transformation and use. To remove the example undesired instances discussed above, the architect inserts the constraints shown in Fig. 2(b) into the Alloy model (verifying them one at a time), which prevent admission of the model form deficiencies illustrated in Fig. 2(a).

When the model is executed again with the added constraints, the instance in Fig. 3(a) presents. Several model form issues are spotted in this scenario: None of the input/outputs received by the actions were generated by any action, which could mean that specification of the source for this input/output was overlooked; and one of the actions does not generate or receive any input/outputs: an idle action is a possible oversight in the design. (The former is also evident in Fig. 2(a), but this method provides for the removal of one concern at a time so that the architect may validate that each new independent constraint has the intended effect [14].) Furthermore, the example highlights a concern regarding interoperability in designs for which the two assets shown are expected to interoperate with each other, or some other asset not shown.

Interoperability may be formally defined as follows: "the ability of a performer to exchange resources with one or more other performers and to use those resources to accomplish its performed activities according to expected criteria" [14]. This definition of interoperability synthesizes key concepts embodied by most existing definitions [20-26], with one major difference: it shifts the scope from describing the ability of performers in general to specifying the ability of a singular performer in the context of other performers with which it may be expected to interoperate. This critical difference treats each performer in the architecture model as a discrete entity, so that interoperability of each performer can be separately and deterministically evaluated based on entities, relationships, and even attributes (for timing and performance factors) in the model.

This definition of interoperability can be used to evaluate the design in Fig. 3(a), which shows two performers (called assets in LML) that do not exchange resources (called input/outputs in LML) with any other performer in the scope of the diagram. To address this concern, a constraint is needed to exclude assets that do not exchange input/outputs with any other asset from the scope of what is considered interoperable according to the definition above. Note that certain assets may be deliberately designed to be standalone performers that do not interact with other assets, and these assets may be systems composed of constituent assets that are interoperable with each other. The constituent assets that meet this definition of interoperability may be considered interoperable, even if the top level asset does not meet the above definition, which is focused on exchanges with *other* assets. Fig. 3(b) highlights the modifications to the Alloy code that adds this interoperability constraint and also excludes scenarios where input/outputs are not generated by any action, and where actions do not generate or receive any input/outputs.



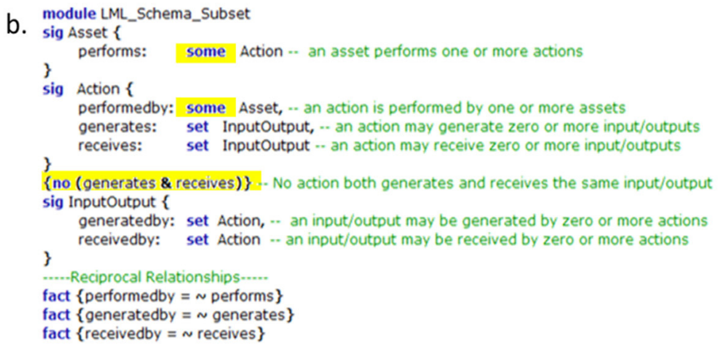
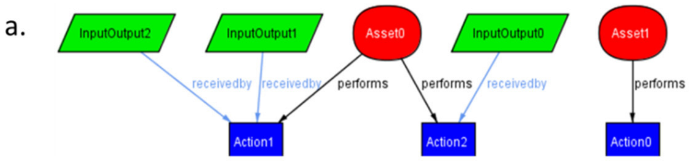

```
b. module LML_Schema_Subset
   sig Asset {
         performs:      some  Action -- an asset performs one or more actions
   }
   sig  Action {
         performedby:  some  Asset, -- an action is performed by one or more assets
         generates:      set  InputOutput, -- an action may generate zero or more input/outputs
         receives:       set  InputOutput -- an action may receive zero or more input/outputs
   }
   {no (generates & receives)} -- No action both generates and receives the same input/output
   sig InputOutput {
         generatedby:  set  Action, -- an input/output may be generated by zero or more actions
         receivedby:    set  Action  -- an input/output may be received by zero or more actions
   }
   -----Reciprocal Relationships-----
   fact {performedby = ~ performs}
   fact {generatedby = ~ generates}
   fact {receivedby = ~ receives}
```

Fig. 2. (a) Example instance of an architecture model schema subset with looping outputs and an isolated asset; b) Alloy code for an architecture model schema subset highlighting constraints to address these deficiencies.



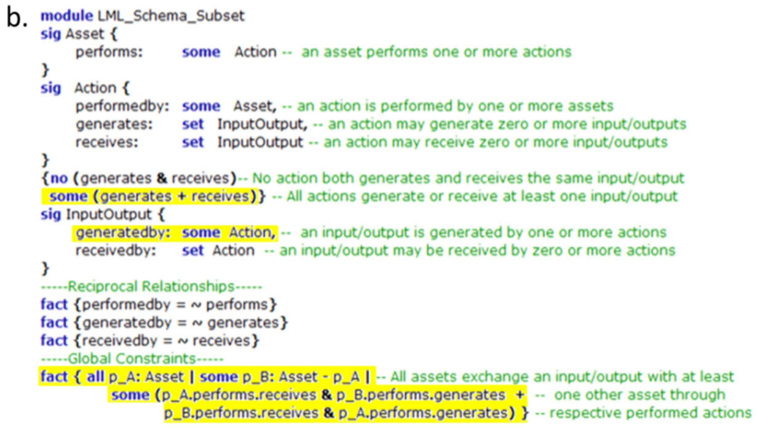

```
b. module LML_Schema_Subset
   sig Asset {
         performs:      some  Action -- an asset performs one or more actions
   }
   sig  Action {
         performedby:  some  Asset, -- an action is performed by one or more assets
         generates:      set  InputOutput, -- an action may generate zero or more input/outputs
         receives:       set  InputOutput -- an action may receive zero or more input/outputs
   }
   {no (generates & receives)-- No action both generates and receives the same input/output
    some (generates + receives)} -- All actions generate or receive at least one input/output
   sig InputOutput {
         generatedby:  some  Action, -- an input/output is generated by one or more actions
         receivedby:    set  Action  -- an input/output may be received by zero or more actions
   }
   -----Reciprocal Relationships-----
   fact {performedby = ~ performs}
   fact {generatedby = ~ generates}
   fact {receivedby = ~ receives}
   -----Global Constraints-----
   fact { all p_A: Asset | some p_B: Asset - p_A | -- All assets exchange an input/output with at least
             some (p_A.performs.receives & p_B.performs.generates  +  -- one other asset through
                   p_B.performs.receives & p_A.performs.generates) } -- respective performed actions
```

Fig. 3. (a) Example instance of an architecture model schema subset meeting constraints previously identified; b) Alloy code highlighting new constraints to address actions with no inputs/outputs, inputs/outputs with no source action that generates it, and non-interoperable assets.

The identification of constraints continues in this manner to iteratively filter out undesired scenarios. For example, the above constraints do not provide for actions that generate input/outputs without receiving any input/outputs (a violation of the law of conservation of matter and energy). This scenario (example shown in Fig. 4) can be filtered out with a local constraint on actions: (**some** generates **implies** **some** receives).

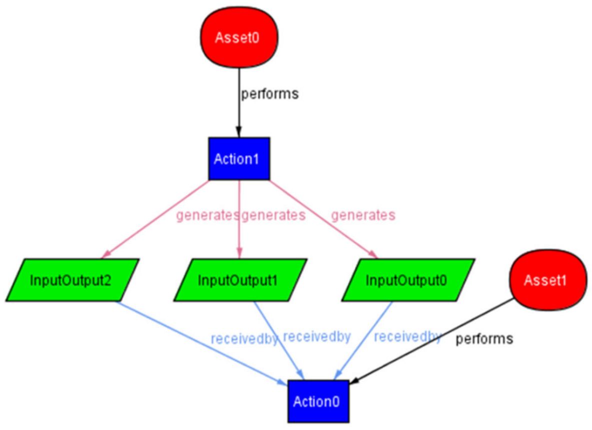


Fig. 4. Example instance of an architecture model schema subset that shows an action generating three input/outputs without consuming any input/outputs.

Appendix A contains the text of a more complete Alloy model. This text may be copied into the publicly available Alloy Analyzer tool [19] for further experimentation.

3.4. Formalize axioms

One may wish to formalize the constraints identified during the modeling process into axioms independent of a specific modeling language like Alloy, for sharing with a broader community of interest. First order predicate calculus is universal in its basis in mathematics, and thus an appropriate notation for expressing such axioms for the purpose of serving as the basis for implementation in any capable language or tool of choice. Tab. 1. contains a list of formalized axioms that an architect may be interested in using as general filters during the model verification process as the model matures. Model maturity is subjective, and appropriate axioms can be applied by individual architects or organization at different phases of a system's design. It is worth emphasizing that not all axioms identified need be enforced in all models, since not all may conform to the preferences and practices of every architect. Nor could one expect all axioms to hold true early in a model's development, as a model takes time to be populated with the correct entities and relationships. Therefore, the axioms are organized by the phase in which they are expected to apply. Tab. 2. contains example axioms that may be used to assess design maturity by negotiating and establishing formal definitions for concepts currently lacking formality or any means for assessment in an architecture model. In this case, the example axioms pertain to non-interoperability of a design. The design is considered non-interoperable (denoted by $\neg I$) if, for example, an asset generates an input/output that is not received by any other asset, if an asset receives an input/output that is not generated by any other asset, or if an input/output that is exchanged between any two assets is not transferred by some conduit that connects to those two assets. Additional axioms for non-interoperability may be developed by extending the scope of the data model considered [14]. Thus, this method offers a formal platform for experienced architects and engineers to debate the meanings of hard to quantify terms, and create a knowledge base of their collective experiences in the process.

Some relationships are computable by extension, as in (1) and (2). Auxiliary predicates may be defined as in (3), where asset p_1 generates an input/output to or receives an input/output from asset p_2.

$$generates(p,n) = (\exists a \in A)[performs(p,a) \wedge generates(a,n)] \tag{1}$$
$$receives(p,n) = (\exists a \in A)[performs(p,a) \wedge receives(a,n)] \tag{2}$$
$$exchanges(p_1, p_2, n) \tag{3}$$

Table 1. Example model maturity axioms, organized by application to notional phases.

Phase 1 Model Maturity		
1.1	No action generates and receives the same input/output.	$(\forall a \in A)(\neg\exists n \in N)[generates(a,n) \wedge receives(a,n)]$
1.2	If any action generates an input/output, it also receives an input/output.	$(\forall a \in A)(\exists n_1 \in N)[generates(a,n_1) \rightarrow (\exists n_2 \in N)receives(a,n_2)]$
Phase 2 Model Maturity (Phase 1 axioms, plus the following are satisfied)		
2.1	Any input/output is generated by some action.	$(\forall n \in N)(\exists a \in A)[generatedby(n,a)]$
2.2	Any input/output is received by some action.	$(\forall n \in N)(\exists a \in A)[receivedby(n,a)]$
2.3	Any action is performed by some asset.	$(\forall a \in A)(\exists p \in P)[performedby(a,p)]$
2.4	Each action generates or receives at least one input/output.	$(\forall a \in A)(\exists n \in N)[generates(a,n) \vee receives(a,n)]$
2.5	Each asset performs at least one action.	$(\forall p \in P)(\exists a \in A)[performs(p,a)]$
Phase 3 Model Maturity (Phase 2 axioms, plus the following are satisfied)		
3.1	Each asset is connected by at least one conduit.	$(\forall p \in P)(\exists c \in C)connectedby(p,c)$
3.2	Any conduit connects to at least two disjoint assets.	$(\forall c \in C)(\exists p_1 \in P)(\exists p_2 \in P)[connectsto(c,p_1) \wedge connectsto(c,p_2) \wedge (p_1 \neq p_2)]$
3.3	Any conduit connects to no more than two assets.	$(\forall c \in C)(\neg\exists p_1 \in P)(\neg\exists p_2 \in P)(\neg\exists p_3 \in P)$ $[connectsto(c,p_1) \wedge connectsto(c,p_2) \wedge connectsto(c,p_3) \wedge$ $(p_1 \neq p_2) \wedge (p_2 \neq p_3) \wedge (p_1 \neq p_3)]$
Phase 4 Model Maturity (Phase 3 axioms, plus the following are satisfied)		
4.1	If any two assets exchange some input/output, those assets are connected to at least one common conduit.	$(\forall p_1 \in P)(\forall p_2 \in P)(\exists n \in N)$ $[exchanges(p_1,p_2,n) \rightarrow ((\exists c \in C)connectsto(c,p_1) \wedge connectsto(c,p_2))]$
4.2	Every exchanged input/output between any two assets is transferred by some conduit that connects to those assets.	$(\forall p_1 \in P)(\forall p_2 \in P)(\forall n \in N)$ $[exchanges(p_1,p_2,n) \rightarrow$ $((\exists c \in C)transferredby(n,c) \wedge connectsto(c,p_1) \wedge connectsto(c,p_2))]$
Phase 5 Model Maturity (Phase 4 axioms, plus the following are satisfied)		
5.1	Each asset generates an input/output to or receives an input/output from at least one other disjoint asset.	$(\forall p_1 \in P)(\exists p_2 \in P)(\exists n \in N)[\ (produces(p_1,n) \wedge consumes(p_2,n)) \vee$ $(consumes(p_1,n) \wedge produces(p_2,n)) \wedge p_1 \neq p_2\ ]$

Table 2. Example design maturity axioms for non-interoperability, organized by application to notional phases.

Phase 1 Design Maturity: Non-interoperability Axioms		
1.1	If there exists an asset that generates an input/output that is not received by any other asset, the design is not interoperable.	$(\exists p_1 \in P)(\exists n \in N)[\ generates(p_1,n) \wedge \neg((\exists p_2 \in P)receives(p_2,n))\ ] \rightarrow \neg I$
1.2	If there exists an asset that receives an input/output that is not generated by any other asset, the design is not interoperable.	$(\exists p_1 \in P)(\exists n \in N)[\ receives(p_1,n) \wedge \neg((\exists p_2 \in P)generates(p_2,n))\ ] \rightarrow \neg I$
Phase 2 Design Maturity: Non-interoperability Axioms		
2.1	If an input/output that is exchanged between any two assets is not transferred by some conduit that connects to those two assets, then the design is not interoperable.	$(\exists p_1, p_2 \in P)(\exists n \in N)$ $[exchanges(p_1,p_2,n) \wedge \neg(\ (\exists c \in C)transferredby(n,c) \wedge$ $connectsto(c,p_1) \wedge connectsto(c,p_2)\ )] \rightarrow \neg I$

4. Implementation

Once such axioms are codified in a capable tool compatible with the data model used for their expression, specific instances of noncompliance with the axiom in an actual architecture model can be flagged. Such tools implement these rules as filters or warnings that may be toggled on or off at each architect's or organization's discretion for automated assistance in finding instances in the model that fit a certain pattern. Implementation of this formal method for several example axioms has been accomplished in two different systems engineering tools to date (CORE® and Innoslate®). More details on implementation are provided in Rodano and Giammarco [17].

5. Conclusions and way ahead

Since the model maturity and model design axioms presented herein are expressed at such a high level of abstraction, they have applications to modeled architectures in many domains. Discovered patterns formalized as axioms can be distributed, codified in a tool of choice, and sought out in actual architecture models of systems using automation. These formalized axioms provide a basis for assessing development progress of modeled architectures, as exemplified in Tables 1 and 2, when implemented as shown in Rodano & Giammarco [17]. Each axiom is written to be minimally constraining, so that any combination of axioms may be used to restrict the design at given level of architecture maturity according to an architect's preference. This list of axioms is intended to grow as the user community identifies more model patterns for abstract capture.

As an outcome of formalization of such expectations, it becomes possible to establish formal measures and criteria for both model maturity and design maturity that can be tailored for different events, analytics, and decision points. Metrics tied to the results of queries based on these axioms can be used to provide a more quantitative and precise basis for assessing model and design characteristics, such as interoperability of a design, and to remove subjectivity from the assignment of descriptors, levels, and other qualitative instruments.

Although the LML specification served as the basis for the examples in this paper, other data models may be substituted for use with the method presented. Likewise, interoperability served as one example characteristic of concern; the same method could potentially be used for the assessment of other nonfunctional system-wide design characteristics such as security, safety, survivability, and other –ilities.

A key realization that resulted from constructing formal logic statements using an architecture model schema is that these expressions can provide systems engineers and integrators of complex systems with the ability to create clear, unambiguous, and tool-independent sets of axioms to embody certain expectations for models and designs. These axioms can be used to establish requirements and instructions that can be tailored for any system under design and its corresponding decision points. Formal axioms can be used to define what, precisely, a government, corporation, or other customer/stakeholder means by terms like interoperable, integrated, complete, mature, sustainable, and other quality attributes for which many principles, guidelines and policy statements exist in natural language [3,27,28], but for which formal specifications are largely absent. Formal axioms for architecture model qualities such as interoperability of a design provide clear, tailorable, testable requirements that can be used to assess and measure the degree to which architecture models comply with expectations for these quality attributes over time as the model matures. Architects may then use these axioms to conduct pre-assessments of their architecture models and ensure they match the formally defined expectations before delivery to and assessment by the customer. The formal method presented can be used to set and validate architecture model and design criteria using automation, giving the humans working on a particular system a collective body of knowledge and experience to draw upon and contribute back to, building an architecture model patterns repository in the process.

In formalizing our human experience with desired and undesired characteristics of systems into patterns, we externalize that experience so that others without direct involvement in the resulting lessons learned can benefit. In his ITEA journal article [29], RADM Dunaway states that "The bottom line is that when we have the discipline to execute our known best practices we minimize the cost to the taxpayer and the adverse impact to warfighting capability." Formal methods provide an elegant way to exercise that discipline, providing a vehicle to address our collective memory loss of best practices. Applied to architecture models, they provide a means to share patterns for architecture best practices and lessons learned with minimal error and high reusability. The axioms developed in this research are simultaneously abstract and precise: abstract enough to apply in multiple domains, and precise enough to provide deterministic assessment results on a given set of data when implemented in a tool. This formal method provides an analytical underpinning for assessment of models and designs based on architecture entities and relationships, and lays the groundwork for further formalization in the pursuit of precise and measurable definitions for system-wide characteristics that have thus far eluded our assessment instruments.

Acknowledgements

The author would like to acknowledge and thank Drs. Geoffrey Xie and Cliff Whitcomb for their advising of the author's dissertation on architecture model-based interoperability assessment, Dr. Mikhail Auguston for his mentoring in formal methods for architecture modeling, and the highly capable engineers at Vitech Corporation and SPEC Innovations who have brought many of the experimental axioms to life.

Appendix A. Example Alloy model of a subset of the architecture data model under analysis

```
module LML_Schema_Subset

sig Asset {
            performs:      some Action -- an asset performs one or more actions
}
{lone (performs)} -- All assets perform no more than one action

sig   Action {
            performedby:   some Asset, -- an action is performed by one or more assets
            generates:     set InputOutput, -- an action may generate zero or more i/o
            receives:      set   InputOutput -- an action may receive zero or more i/o
}
{no (generates & receives)-- No action both generates and receives the same input/output
 some (generates + receives) -- All actions generate or receive at least one input/output
  (some generates implies some receives)} -- If an action generates i/o, it also receives i/o

sig InputOutput {
            generatedby:   some  Action, --  an i/o is generated by one or more actions
            receivedby:    some  Action  -- an i/o is received by one or more actions
}

-----Reciprocal Relationships-----
fact {performedby = ~ performs}
fact {generatedby = ~ generates}
fact {receivedby = ~ receives}

-----Global Constraints-----
-- All assets exchange an input/output with at least one other asset through respective
performed actions
fact { all p_A: Asset | some p_B: Asset - p_A |
       some (p_A.performs.receives & p_B.performs.generates  +
            p_B.performs.receives & p_A.performs.generates)}

-- All assets perform disjoint actions
fact { all disj p_A, p_B: Asset | no (p_A.performs & p_B.performs) }

---Run with up to 3 elements from each signature, but with exactly 2 assets (small scope)
run {} for 3 but exactly 2 Asset
```

References

1. Georgia Institute of Technology. Why HSI?. http://hsimed.gtri.gatech.edu/hsi_info/hsi_intro_why.php, accessed Sept. 2012.
2. Lifecycle Modeling Languiage (LML) Specification, version 1. http://www.lifecyclemodeling.org/specification/, accessed Aug. 2013.
3. Department of Defense. DoD Architecture Framework, Version 2.0. Washington, DC: ASD(NII)/DoD CIO, 2009.
4. Object Management Group. Unified Profile for DoDAF and MODAF (UPDM), Version 1.1. Needham, MA: Object Management Group (OMG), 2011.
5. Gamma E, Helm R, Johnson R, and Vlissides J. Design patterns: Abstraction and reuse of object-oriented design. Springer Berlin Heidelberg, 1993.
6. Bosch J. Design patterns as language constructs. JOOP 11, no. 2 (1998): 18–32.
7. Haskins C, Application of patterns and pattern languages to systems engineering, Proc INCOSE 15th Annu Int Symp, Rochester, NY, Jul. 10–13, 2005.
8. Cloutier RJ & Verma D. Applying the concept of patterns to systems architecture. Systems engineering 10, no. 2 (2007): 138–154.
9. Rebovich Jr. G & DeRosa JK. Patterns of Success in Systems Engineering of IT-Intensive Government Systems. Procedia Computer Science, Volume 8, 2012, Pages 303–308
10. Brownsword L, Albert C, Place P, and Carney D. Software acquisition patterns of failure and how to recognize them. Monterey, California. Naval Postgraduate School, 2013.
11. Hashizume K, Yoshioka N, and Fernandez EB. "Three misuse patterns for Cloud Computing." Security Engineering for Cloud Computing: Approaches and Tool, 2013.
12. Weilkiens T. Two kinds of patterns. http://model-based-systems-engineering.com/2013/02/07/two-kinds-of-patterns/, accessed Nov. 2013.
13. Friendshuh L & Troncale L. Identifying fundamental systems processes for a general theory of systems (GTS), Proceedings of the 56th Annual Conference, International Society for the Systems Sciences (ISSS), July 15–20, San Jose State Univ. (electronic proceedings: Go to http://journals.isss.org), 22 pp, 2012.
14. Giammarco K. Architecture model based interoperability assessment. Doctoral thesis, Naval Postgraduate School, Monterey, CA. Jun. 2012.
15. Giammarco K, Xie G, Whitcomb C, A formal method for assessing interoperability using architecture model elements and relationships. Proceedings of the 9th Annual Conference on Systems Engineering Research, Redondo Beach, CA. Apr. 2011.
16. Giammarco K. Formal methods for architecture model assessment in systems engineering. Proceedings of the 8th Annual Conference on Systems Engineering Research, Hoboken, NJ. Mar. 2010.
17. Rodano M & Giammarco K. A formal method for evaluation of a modeled system architecture. Procedia Computer Science, Volume 20, 2013, Pages 210–215.
18. Jackson D. *Software Abstractions: Logic, Language and Analysis*. Cambridge, Massachusetts: The MIT Press, 2006.
19. Jackson D. Alloy: A language & tool for relational models. http://alloy.mit.edu/alloy/, accessed Nov. 2013.
20. Department of Defense. Joint communications system. Joint Publication 6-0. Washington, DC: Joint Chiefs of Staff, 2010.
21. Department of Defense. Department of Defense dictionary of military and associated terms. Joint Publication 1-02. Washington, DC: Joint Chiefs of Staff, 2010, as amended through October 15, 2011.
22. Department of Defense. *Universal Joint Task List (UJTL).* Chairman of the Joint Chiefs of Staff Manual 3500.04D. Washington, DC: Joint Staff, 2005
23. Department of Defense. Procedures for interoperability and supportability of information technology (IT) and national security systems (NSS), DoD Instruction 4630.8. Washington, DC: ASD(NII)/DoD CIO, 2004.
24. Institute of Electrical and Electronics Engineers (IEEE). Standards glossary. 2011. http://www.ieee.org/education_careers/education/standards/standards_glossary.html#sect9, accessed Jan. 2012.
25. Healthcare Information and Management Systems Society (HIMSS). *HIMSS Dictionary of Healthcare Information Technology Terms, Acronyms and Organizations, Second Edition*. Chicago, IL: HIMSS, 2010.
26. Department of Defense. DM2 conceptual data model (DIV-1) diagram. DoD Chief Information Officer (DoD CIO). (n.d.). http://cio-nii.defense.gov/sites/dodaf20/conceptual.html, accessed Jan. 2012.
27. Buede D. *The Engineering Design of Systems.* New York: John Wiley and Sons, Second Edition, 2009.
28. Maier M & Rechtin E. *The Art of System Architecting*. Boca Raton, Florida: CRC Press, 3rd Edition, 2009.
29. Dunaway, David A. OPTEVFOR: Organizing to accomplish the OT&E mission (Integrated testing through mission based test design). ITEA Journal, Vol. 32 (2011): 146–148.