Using Domain Model For Structuring Pattern Language

Veli-Pekka Eloranta, Marko Leppänen, and Kai Koskimies
{firstname.lastname}@tut.fi

Department of Software Systems
Tampere University of Technology
Finland

Abstract. Patterns have become a popular means for communicating knowledge about proven solutions, and collections of patterns have been proposed for various fields of software engineering, often referred to as pattern languages. As a pattern language, the patterns are organized according to a structure that facilitates the application order of the patterns. However, the principles and methods for creating such a structure for a pattern language are poorly understood. We propose a general approach to systematically derive a graph structure for a given collection of patterns, based on the use of a domain model describing the application area of the patterns. We demonstrate the approach by applying it in the context of embedded machine control systems in order to create a pattern language for the domain. We compare the result with an existing structuring of the same pattern language, proposed originally by the pattern creators on the basis of an intuitive idea of the use scenarios of the patterns. The results suggest that the new approach for deriving the language structure is viable, yielding a similar structure as the intuitive approach.

1 Introduction

Basically, a pattern language is an organized collection of patterns. A design pattern is usually defined as a general reusable solution to a commonly occuring problem. Alexander [1] defines a pattern as three-part rule, which describes relations between a certain context, a problem, and a solution. The term pattern language, which was also originally introduced by Alexander [1], has been the source of considerable confusion, which has led to the use of new, less demanding terms, like a pattern system [2] or pattern catalog [3], instead of the term pattern language. Much of the confusion is due to the ambiguity of the word "language": it can denote just an informal means of communication or a set of symbols, but also a formal construct with generative flavor.

The introduction of new terms has liberated the pattern language developers to use whatever structuring principles they consider appropriate for organizing the pattern collections and calling them pattern languages. This has, in turn, increased the confusion, and effaced common agreement about the ways patterns should be presented as collections. This makes it difficult to utilize pattern language organization, as the semantics of the structure is not clear. In addition, it makes it more difficult to communicate with patterns as terms are not unambiguous

Rather than introducing yet another term for a collection of patterns we use the original term pattern language, and clarify in the following our view of this concept.

In particular, we aim to be rather precise about the structuring principles of a pattern language and about the purpose of the structuring. Given this view of pattern languages, we will propose a method to systematically derive the required structuring for a pattern language intended for a particular domain. We believe that the clarification of the pattern language concept and the developing of a systematic method to derive a structure for such a language go hand in hand: the method both yields and requires a clear semantics for the notion of the pattern language.

One of the earliest definitions of the term pattern language in the context of software design patterns comes from Coplien [3] who defines a pattern language as a collection of patterns that build on each other to generate a software system. Although his definition is influenced by the early pattern movement concentrating on design patterns, it does capture one of the key characteristics of a pattern language: the ability to generate an artifact. We argue that this is actually the main property of a pattern language, and also the main motivation for providing the structure for the language. The structure should help the user of the pattern language to generate a solution for an artifact. In this work we are primarily interested in architectural software patterns, the artifacts being thus software architectures.

Along these lines, we view the organization of a pattern language basically as a partial order, indicating the possible or recommended application orders of the patterns when constructing a new artifact for the domain. Although in principle the pattern language can in this way generate a complete artifact, in practice patterns are typically used only for a small part of a system, and we argue that the language organization should also support this.

Given a set of patterns that have been collected from various sources and reviewed as independent entities, the derivation of any structure for the set is a non-trivial problem. This holds also for the application ordering structure. As far as we know, there are no systematic methods for producing such a structure; conventionally, the relationships between the patterns are based on intuitive views on the application scenarios, possibly matured and revised in the same way as the patterns themselves. We argue, however, that such a structure is mostly implied by the domain and by the relationships the patterns have to the domain concepts. Some concepts are more fundamental than others. Based on this idea, we propose a method to derive a pattern language structure using a domain model which has been annotated by the patterns. Given such a representation, the pattern language structure as a partial application order of the patterns can be produced algorithmically in a straightforward way. Since these ideas originate from a pattern mining project carried out in Finnish machine industry, we will use this domain as an example, but we emphasize that the method is expected to be applicable in any domain.

The paper is organized as follows. In the following section we briefly summarize the related approaches to organize a pattern language, and the benefits and drawbacks of those approaches. In Section 3 we demonstrate the construction of an appropriate domain model for our example domain, embedded machine control systems. Section 3 also explains how to associate the patterns with the domain model. Section 4 gives the method to produce the actual pattern language structure from the patternized domain model. We conclude with some final remarks on the work and its future directions.

2 Structuring Pattern Languages

The different ways of organizing a pattern language can be roughly divided into problem-centered and solution-centered approaches. In the problem-centered approach, the domain imposes the structure of the language. If the domain is hierachical, this approach yields a basic hierarchical structure for the pattern language. The patterns are organized in such a way that the patterns dealing with major entities are considered first, and the patterns proposing solutions to the different parts of these entities are considered next. That is, the solution structure is generated mainly in a top-down manner, starting from larger entities and moving towards details. For example, Alexander [1] presents a pattern language for building community structures starting from cities and ending with fine details of rooms.

Alexander [1] describes patterns as sets of rules that are invoked by circumstances that the designer is facing. A hierachically structured pattern language provides patterns for different levels of design, and architects use the pattern language and apply patterns at each level according to the situation. The advantage of this approach is that the pattern language generates a consistent structure for the entire artifact. The drawback is that in many domains it is hard to apply the top-down approach. For example, the domain of embedded machine control systems discussed later in this paper does not impose a natural hierarchical structure.

Another problem-centered way to organize a pattern language is to use a multi-dimensional space of the domain as the basis. If the patterns can be associated to one or more points in this space, this space offers a tool to find the appropriate patterns, given the “coordinates” of the interesting aspect in the domain space. For example, Vesiluoma [4] structures a process pattern language for knowledge sharing in software processes by dividing the knowledge sharing related activities according to the parties involved (e.g. within a team, a project etc.), the nature of the work (value adding, managing etc.), and process phase (initialization, closure etc.). A similar approach has been taken in [5]. Assuming that the potential user of the pattern language can locate his or her problem in the space, the patterns possibly helping to solve the problem can be easily identified. The advantage is that the user is not bound to a hierarchical structure and can therefore more easily find individual, potentially useful patterns. The downside is that there is no guarantee that the patterns work together in a consistent manner. Especially in a more technical area this may be a significant drawback.

The solution-centered approaches for organizing a pattern language are based on the relationships between the solutions rather than on structuring the problem domain. The most common relationship is the application order: a pattern has this relationship with another pattern if the former pattern solution assumes the latter. In other words, the pattern language is organized according to a partial application order that can be represented as a graph. If the application order follows the hierarchical structure of the domain, this kind of pattern language organization comes close or equals to the hierarchical domain-centered approach, but in general the domain can be structured in a non-hierarchical way. This kind of graph-like structuring has been used e.g. in the pattern language for fault-tolerant systems [6]. The main advantage of this technique is that it provides a flexible, easy-to-use "map" of solutions with plausible application paths. A drawback is that it may be more difficult to come up with appropriate application

order relationships than in the domain-centered hierarchical case. Additionally, it may harder to find a suitable pattern for an individual problem in a particular system than in the problem-centered case.

In addition to the usage order relationship, other relationships can be used, too. Gamma et al [7] uses named connections between patterns. For example, the connection between TEMPLATE METHOD and FACTORY METHOD is named "often uses". Another practice can be found from [8] where the normal relationship between patterns is "refines", but in addition there is "specializes" relationship meaning that the one pattern improves the solution of the other. Noble et al [8] also uses "conflicts" relationship to inform that two patterns resolve similar forces in an incompatible way. Patterns are in many cases organized in a graphical representation that resembles a pattern language and describes relationships between patterns in a way described above, but the collection still can not be called a generative pattern language. This increases confusion on what a pattern language means.

In some cases also other types of relationships between the patterns are used, like generalization or specialization relationships: a pattern can be viewed as a general form of several more specific patterns. Although "abstract" patterns are strictly speaking not real patterns because they are not necessarily applicable as such, they may serve as a structuring device that helps the user to understand the commonalities between separate patterns.

Another solution-centered approach is to use implicit application order relationships, i.e. pre- and postconditions. A typical pattern format includes parts describing the context where the pattern is applicable, and the situation resulting from the usage of the pattern. If these parts are viewed more formally as pre- and postconditions, they implicitly determine the application order of patterns: a pattern can be applied when its precondition holds, and its application makes the postcondition true. The advantage of this technique is that the patterns do not need to explicitly refer to each other, making the patterns more independent. On the other hand, the independence of the patterns weakens the notion of the pattern language. Although actual pre- and postconditions have not been used formally for patterns as far as we know, many pattern formats in fact explain the meaning of certain parts of the pattern descriptions informally in a way that comes close to pre- and postconditions.

The concept of pattern languages has been widely adopted in software engineering where especially design patterns have been used successfully. Nevertheless, there has not been great success in building a generative pattern language for a certain domain that would actually produce the software architecture piece by piece. The problem is that in practice it is very hard to come up with a set of patterns covering all possible design situations emerging in practice, and to structure the pattern language in such a way that all relevant patterns are identified in the right order. The problem of application order is often visible in a pattern language in the form of an ambiguous start pattern, seeming to imply that the starting points are parallel and exclusive. However, this is not the case: such a pattern language may just be missing the actual pattern that should be applied first. In this case the language is not really a generative pattern language but it is a pattern catalog, or there are actually multiple languages that have been combined for a particular domain.

It is especially difficult to develop a generative pattern language for a wide and unclear domain. For example, it is probably hard to develop a generative pattern language for fault tolerant systems, but it might be possible to create such a pattern language for large maneuverable machines and use some of those fault tolerant patterns as a part of the language. In addition, the same fault tolerant patterns could be used for a different domain as well, for example in a pattern language for control systems with high availability requirements such as control systems of power-distribution networks.

In this paper we propose a novel problem-centered technique to derive a structure for a pattern language. We argue that the critical question is how to represent the problem domain the pattern language is intended for. Both a hierarchical and a multidimensional model discussed above have the problem that they give only a certain restricted viewpoint to the domain, but they cannot comprehensively describe all the relevant concepts belonging to the domain and the relationships between these concepts. A more complete way to describe a domain is to use a conventional domain model, that is, a conceptual model with entities and relationships. Typically, such a model can be given as a UML class diagram.

Using a domain model for structuring a pattern language has several benefits. Firstly, associating the patterns with a domain model supports the finding of the relevant patterns in the language, as the designer can study those patterns associated with the concepts the designer is currently working on. Secondly, as we will show in this paper, the domain model can be used to produce the application order relationships for the patterns as well, resulting in a more generative style of a pattern language.

3 Example Domain: Embedded Machine Control

We have carried out ten architectural evalutions in Finnish machine industry. Five different companies participated in the evaluations and same system was not evaluated twice. However, in two cases the different generations of the same system was evaluated. Besides the actual evaluation, we used the evaluation process for pattern mining: good solutions identified during the evaluations were recorded and formulated as patterns. From the gathered patterns we have created a pattern language of 35 patterns for the domain. A domain model for the maneuverable large machines was also created basing on the experiences from the evaluations. The domain model was created using traditional object analysis techniques.

Architecture evaluation was carried out using scenario-based method called architecture trade-off analysis method, ATAM [9]. The method bases on the elicitation of scenarios that describe situations or future changes that the system might face during its life cycle. A scenario should also explain how the system should respond to the situation or change. The evaluations that were carried out resulted in lists of scenarios and from these scenarios concepts and abstractions were searched by underlining keywords of the domain. This resulted in a list of keywords for the domain. From this list larger entities comprising of keywords were identified and relationships of these entities were discovered. Then the first version of the domain model was created. Finally the domain model was revised according to comments from the domain experts. This process resulted in the domain model that is depicted in Figure 1. As the domain model

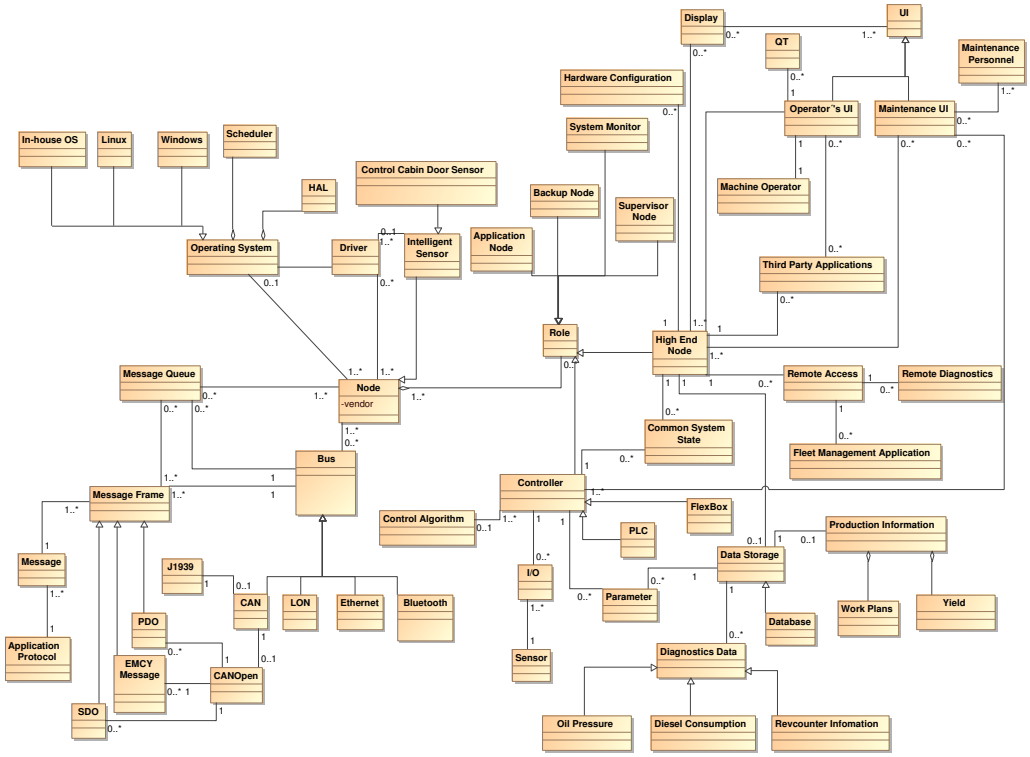

Fig. 1: Domain model for the embedded control system.

was created from scenarios that were elicited by designers, the process resulted in a somewhat technically-oriented domain model.

As the patterns were discovered during the architecture evaluation, there was a possibility to map patterns to scenarios where the solution described by a pattern was found. Using this information of the origin of patterns, keywords underlined from scenarios and refined pattern descriptions we mapped the patterns to the domain model. For example, SINGLE POINT OF UPDATING pattern was mapped to the Node entity in the domain model, as the common solution found from the systems was to create one node that updates other nodes in order to make the updating process easier. Each pattern is presented as a collaboration between entities in the domain model. Part of the domain model with patterns presented as collaborations is shown in Figure 2.

It was found out that the connection was not always unambiguous: the pattern could have different kinds of relationships to entities in the model. Consider the VARIABLE MANAGER pattern that describes a solution on how to add a component to each node to enable the nodes to share consistent state information, as an example. It obviously has a relationship to Node entity in the domain model and to Common System State entity. Nevertheless, it is clear that these two relationships are of different nature. Having Node in the system is more like a precondition for the applicability of the pattern whereas

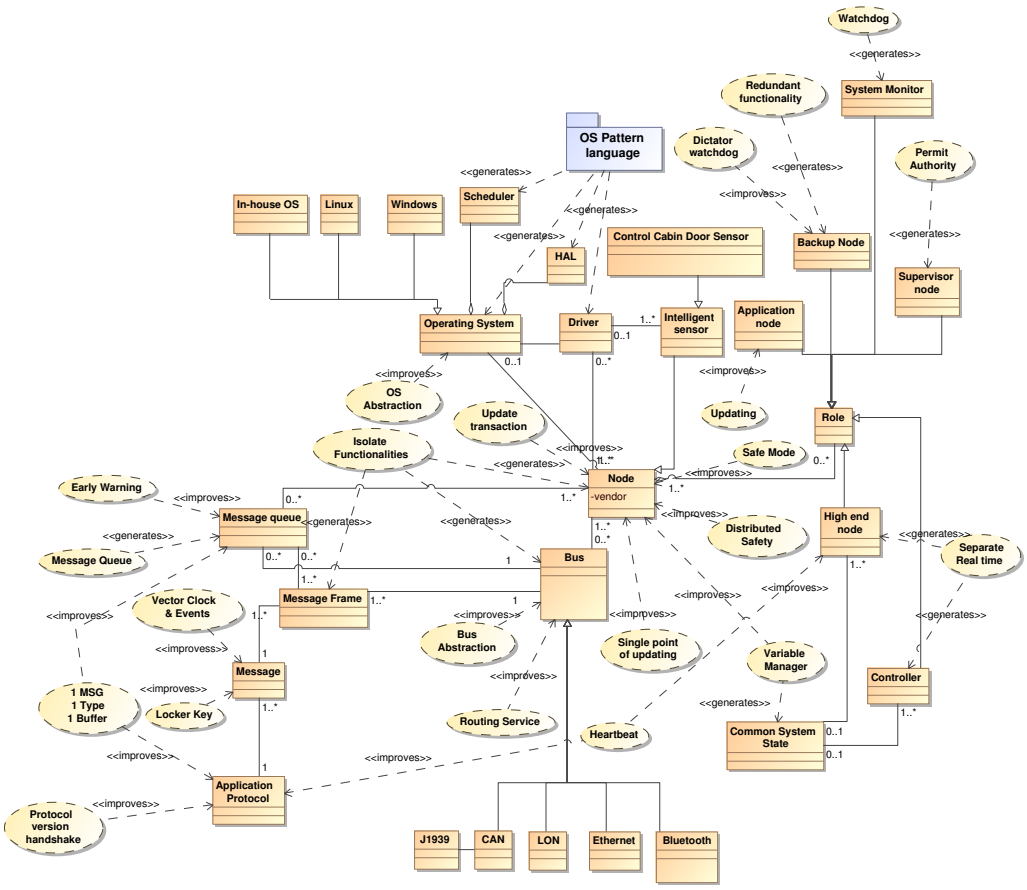

Fig. 2: Partial domain model annotated with patterns discovered during evaluations.

Common System State is more like a post condition. This resembles the context and resulting context parts of the pattern description.

It seems that the relationships between patterns and entities of the domain model are twofold: the pattern either introduces new entities and *generates* a solution for the entity in the domain model or *improves* the solution that is applied by using generating entity. In this way, a new system can be thought of as a *tabula rasa* until the first pattern, generating the basic artifacts of the system, is applied. The complete domain model with patterns attached to it can be seen as a blue print of any system of the domain. After the first pattern is applied, other patterns could be applied to improve already generated solutions or adjacent patterns from blue print domain model could be applied to generate new entities and solutions to the designed system.

Initially we had a third kind of relationship between patterns and entities as well: *is related*. For example, MESSAGE QUEUE pattern is obviously related to messages,

in this case Message Frame entity, but the pattern does not have a corresponding relationship drawn in the domain model. Initially we marked these relationships with connection named *is related*, but we found it unneccessary. ISOLATE FUNCTIONALITIES generates Message Frame entity to the domain model. MESSAGE QUEUE pattern can not be applied before the Message Frame entity exists in the domain model. This defines the pattern applicability order and there is no need to use *is related* relationship between the MESSAGE QUEUE pattern and Message Frame entity as the connection is already available from the domain model itself. The domain model informs that these two entities are related.

One could argue why we use *improves* relationship. For example, the STATIC SCHEDULING pattern that proposes that static scheduling and compile time verification should be used in order to meet hard real time requirements of the system. The pattern changes the scheduler of the system. Depending on the point of view, it improves or worsens the scheduler. Nevertheless, we use *improves* term from the pattern's context point of view. The pattern improves the artifact in the context that is set by a pattern. In other words, it solves the forces given in the pattern description in given context.

The upper left part of the domain model contains a package OS pattern language. The package is connected to the domain model because we see few of our patterns as a part of a greater pattern language. Operating systems are so complex domain that they have their own pattern language(s) and patterns. PARALLEL EXECUTION and HARDWARE ABSTRACTION LAYER that we have found during the pattern mining come from that domain and are part of a pattern language for operating systems. These are taken in our language to provide the most important leverage points to modify the performance of an embedded control system. This package is just added to our language using the same rules. As the OS pattern language *generates* Operating System, HAL, Driver and Scheduler entities to the domain model, the example patterns from that language (PARALLEL EXECUTION and HARDWARE ABSTRACTION LAYER) must be applied first to allow OS ABSTRACTION to be applied.

Another benefit of connecting patterns and domain model is that it is an easy way to check the validity of a domain model and completeness of a pattern collection. If there are artifacts in the domain model that do not have *generates* relationship with any pattern, there might be a pattern missing. Moreover, usually the patterns that are identified tend to become too large and general when they are refined in the pattern workshops after the identification. The lack of focus of the pattern can be easily detected when trying to connect the pattern with a domain model: it seems that the pattern does not directly relate any of the entities of the domain model or it could be attached to many or all entities of the domain. The LIMP HOME pattern is a good example of this.

Initially the LIMP HOME pattern described that if a non-critical part of the system fail, it should still be possible to use the machine in degraded service quality mode. As such the pattern can not be connected to any entity in the domain model. Nevertheless, by focusing the pattern more, it could be connected. Currently the pattern deals with a situation where one or more high end nodes offering supporting services such as remote access can crash and the machine should remain maneuverable. In the end, the pattern was formulated so that it improves SEPARATE REAL TIME pattern. This means that the pattern should be connected to High End Node and Controller entities with *improves*

relationship. In this process, the pattern description was also reformulated. After the pattern description was modified, it also correnssponded more accurately to systems where it was found.

When the patterns are attached to the domain model successfully, a pattern language graph can be systematically derived from the resulting extended domain model. Constraints that this model must meet and the algorithm that generates the pattern language graph are presented in the next chapter in more detail.

4 Deriving Pattern Language Structure Using Domain Model

4.1 Method

The driving principle behind the process of building a pattern language using the domain model as a basis is to identify what entities a pattern creates when applied. This is the *generates* relation. Some patterns don't generate entities, but rather improve or modify them. These are the *improves* relations. The generating patterns must be applied first to have all required entities and solutions in place to improve them, so they must be placed closer to the root of the language graph. The improving patterns depend on the generating patterns as they improve the solution that is achieved by using a generating pattern. This clearly sets an ordering principle between the patterns.

In order to create a pattern language graph using the domain model, the model has to be built according to a few simple rules. When combined with a pattern set, the domain model must satisfy the following two rules. A new entity E can be created to the domain model directly with a generating pattern or by generating the aggregate entity that holds E as a composite part. In other words, creating an aggregate entity naturally creates all its parts from which it consists of. There can be no two separate ways to generate E. This means that *generates* relation can be linked to the domain entity at most one. If the composite entity can be generated, no further *generates* relations are allowed to the subparts.

The inheritance has two different interpretations in the domain model and these must be handled differently. The first case is that inheritance is used to show different concrete alternatives of a domain entity. This means that a certain solution for the parent class entity can be generated by applying a pattern associated with the entity. Then patterns can not generate the subclasses. For example, in our domain model Bus entity is generated by ISOLATE FUNCTIONALITIES pattern and therefore the subclasses, e.g. Bluetooth, are not generated. The subclasses just provide concrete examples of the parent entity. The second case is that the parent class is used to classify different entities under an abstract common name. This means that generative patterns must be associated with the subclasses because the parent class is only an abstract classifying entity. In this case, all subclasses should have *generates* relationship with a pattern. Multiple levels in inheritance hierarchy should be handled in the same way.

The combined pattern and domain model has four kinds of relations: dependencies from patterns to entities, associations between entities, and inheritance and aggregation relations between entities. To derive a partial order for the patterns that is required for the pattern language, we need rules to order the entities. We use the multiplicities to

infer the directions between the entities. Note that multiplicities are only one possible source for inferring directions between associated entities. If the directions of associations are given explicitly in the model, multiplicities are not needed.

The associations are called *equal connections* in case the multiplicities are such that both ends of the association have at least 1 as the minimum multiplicity or the relation is aggregation, because both ends of the relation are then always needed together. This means that we can argue that in the domain model these entities form one unseparable whole and are therefore generated together. The associations are *subordinate connections* if the multiplicity on one side has 0 as the lower bound and on the other side at least 1. In this case we conjecture that the optional entity is subordinate to the mandatory entity, and therefore the mandatory entity must exist before the optional one. If the lower bounds in both ends are 0, no direction information can be extracted, but we postulate that these relations can be seen as equal connections.

Let E denote an entity and P a pattern. Then the formal requirements for a patternized domain model are as follows:

1. Each entity E_i must be generated at most once.
2. If pattern P generates an entity E which aggregates other entities $\{E_1, E_2, ..., E_n\}$ all aggregated entities are implicitly generated by pattern P.
3. If pattern P generates an entity E which has child entities $\{E_1, E_2, ..., E_n\}$ then child entities can not be generated by any other pattern.
4. If E has subclass entities $E_1, ..., E_n$, either E is must be generated by a pattern or all subclass entities $E_1, ..., E_n$ must be generated, but not both.

Assuming a model satisfying the above requirements, we can produce a partial order for the patterns by inferring order relationships (<) between the entities and patterns according to the following rules:

1. If E is generated by P, E = P
2. If P improves E, E < P
3. If E is subordinately connected to E', E < E'
4. If E is equally connected to E', E = E'

4.2 Application

To clarify the rules, we present an example to create a partial language from our domain model annotated with patterns (Fig. 2). We take the part of the domain model consisting of entities: Node, Role, Backup Node, High End Node, Controller and Common System State to describe the pattern language building process. The following patterns that are attached to the forementioned entities are used as an example. These patterns are ISOLATE FUNCTIONALITIES, REDUNDANT FUNCTIONALITY, DICTATOR WATCHDOG, SEPARATE REAL TIME and VARIABLE MANAGER. If we follow the four rules presented in Section 4.1, we can deduct the following facts:

– ISOLATE FUNCTIONALITIES = Node, by rule #1
– Node > Role, by rule #3
– High End Node < Node, by rule #4

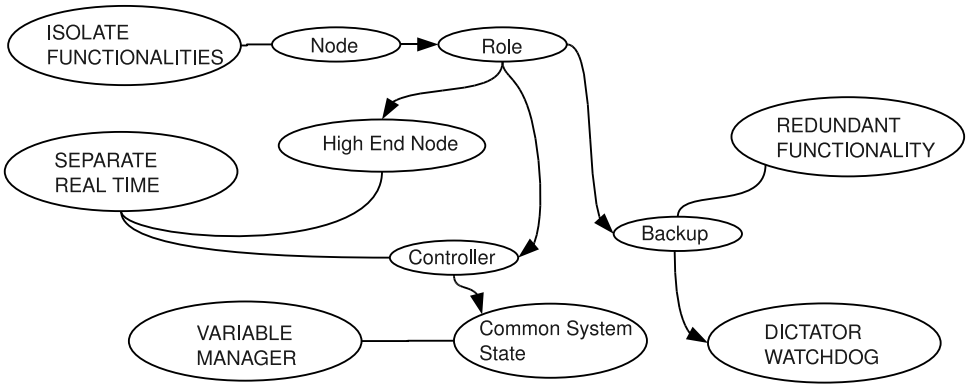

Fig. 3: The ordered tree. Pattern names are written with capital letters.

– Controller < Node
– Backup node < Node
– SEPARATE REAL TIME = High End Node
– SEPARATE REAL TIME = Controller
– VARIABLE MANAGER = Common System State
– VARIABLE MANAGER < Node, by rule #2
– High End Node < Common System State
– Controller < Common System State
– REDUNDANT FUNCTIONALITY = Backup node
– Backup node < DICTATOR WATCHDOG

The order the facts above are generated is not important: we can go through all entities in arbitrary order. A noteworthy fact is Controller < Node. This fact can be deduced from the domain model as Role is subordinately connected to Node entity and Controller is inherited from Role entity. Therefore Role is smaller than Node and also Controller is smaller than Node. Same applies to Backup node entity.

The facts deduced from the annotated domain model can be used to form a tree as depicted in Fig. 3. The arrows show the ordering and the undirected connections mean equality. For example starting from pattern ISOLATE FUNCTIONALITIES we draw undirected connection to Node entity as they are marked equal by facts derived from the annotated domain model. An arrow is drawn from Node to Role as Node is marked to be smaller than Role.

The VARIABLE MANAGER < Node relation can be seen as redundant, as equality to Common System State entity convey more exact information. Now the equal nodes in the tree can be renamed according to their counterpart pattern names and this directly creates the pattern language graph. For example ISOLATE FUNCTIONALITIES

and Node form one node in the pattern graph and SEPARATE REAL TIME, High End Node and Controller form another node. If there is an alternative path corresponding to an arc in the graph, the arc is removed as redundant. For example there will be two arcs from ISOLATE FUNCTIONALITIES to SEPARATE REAL TIME. The other arc is removed as it is redundant.

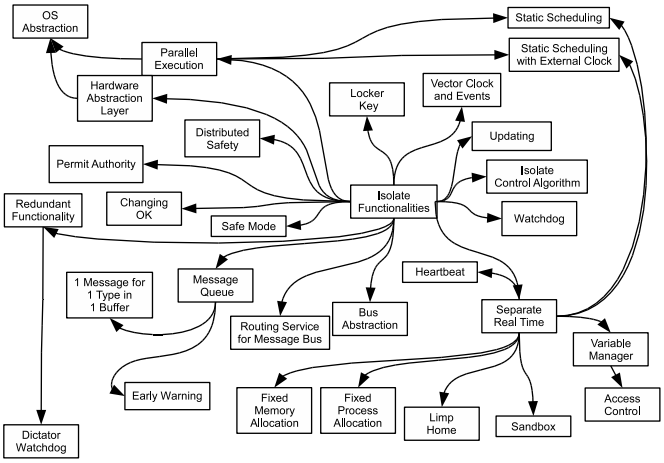

Fig. 4: Excerpt of the pattern language.

Now we have constructed a part of the pattern language for the embedded control system domain. If we continued building the pattern language using the method we would get the pattern graph depicted in Fig. 4.

We have previously constructed a graph structure for the same pattern language using a more intuitive method. The structure was achieved by simulating the development process of an imaginary example system and observing the order in which the patterns presumably are taken into use. The most fundamental pattern is the starting point of the language graph. This resulted in the structure depicted in Fig. 5.

The language construction using method described in this paper is quite straightforward, but the language graph becomes somewhat flat using the current patternized domain model. This can be due to the missing generating patterns. The most clear benefit of the method is that it provides a systematic way to create generative pattern language for a given domain when the patterns have been mined. The method gives a starting point where the pattern language can be further developed using intuitive methods. Another advantage of the technique is that the missing parts of the pattern language can be seen easily from the domain model, so that the pattern mining efforts can be directed towards these parts of the domain. If there is a part in the domain model with no patterns attached to entities with *generates* relationship, it probably indicates that there are some patterns missing from the collection as all entities should be generated by a pattern.

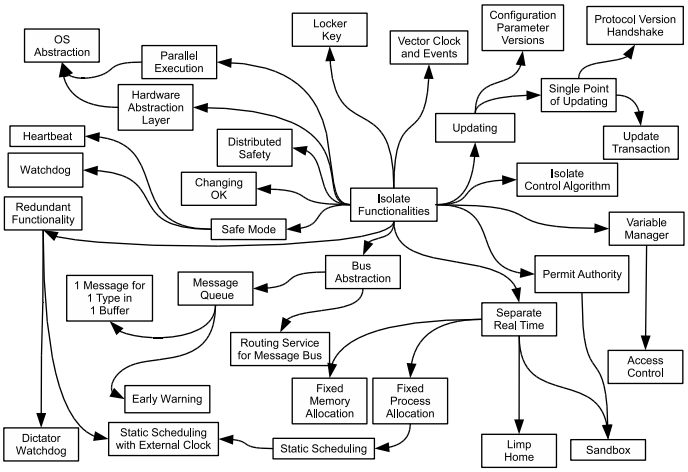

Fig. 5: Intuitively structured pattern language.

The intuitive method resulted in a suprisingly similar language construction to that built using the domain model method. The other language construction ends up in having patterns that affect similar aspects of design as branches of the language graph. Some clear differences however can be seen. For example, SEPARATE REAL TIME is more central pattern in the generated language than in the intuitively structured language. This may indicate that it has been misunderstood in the intuitively built language and this pattern description needs some focusing. Other difference is that the algorithmically generated language is more flat, i.e. the graph does not have long paths, this is mainly due to the missing generating patterns.

As the method is based on clear rules it could be possible to create a tool to process the annotated domain model and create pattern language from it automatically. Nevertheless, we have not explored the possibility of creating such a tool yet. The tool could be implemented as a plug-in for an existing modelling tool, but this is a subject of future research.

5 Conclusions

We have presented a technique to derive a graph structure for a pattern language, describing the recommended usage order of the patterns. The technique is based on exploiting a model of the intended application domain of the patterns. By extending the domain model with the patterns, and inferring the usage order on the basis of the relationships in such an extended domain model, the pattern language structure can be derived in a systematic - even partly automated - way. This both facilitates the pattern language construction process and gives a clear semantics for the concept of a pattern language.

An obvious drawback of the technique is the need for an appropriate domain model. To be useful in this context, the concepts in a domain model should be associated to the patterns, that is, the domain model has to be expressed with terms that can be easily related to pattern descriptions. This seems to be the case if the model is constructed in the way we did it in the example case, namely by analyzing the scenarios describing the "test cases" of architectural evaluation. Since the scenarios are assumed to test the architectural solutions, the scenarios are typically expressed in domain terms that are relevant for the solutions, like patterns. However, using other sources it might be possible to come up with a different model for the same domain with less appropriate concepts for our purposes. As Leppänen et al [10], we conclude that architectural evaluation and pattern mining are a natural mix, not only to identify the relevant patterns but also to derive the structure of the pattern language.

Our approach emphasizes the generative nature of a pattern language in the sense that the patterns are assumed to generate concepts. In fact, if the pattern language is complete enough, it could be actually used to produce the entire architecture, or at least its central parts. Given that each pattern has a specific structural solution, the designer could follow the pattern language graph structure, and by selecting certain paths in the graph enforce certain structural solutions in the architecture. This process could be even at least partly tool supported. The domain model could be used also to limit the set of applicable patterns: if the domain model is first pruned to match with the desired system, only those patterns relevant for the concepts of the target system remain, giving a sublanguage of the original language. These topics are among our future work.

Note that the concept of a pattern in our context is very general. In fact, any architectural solution could here serve in the role of a pattern, even a company-specific or system-specific solution. Company-specific pattern mining has been actually found useful in many cases, storing the design know-how of the company in an easily accessible form. Especially if the patterns are system-specific, the concept of a pattern language is changed, however, because it loses its generative character. Nevertheless, in the context of an architectural evaluation of a particular system, it might be worthwhile to record *all* solutions as pattern candidates, and derive a solution map for the system as described in this paper. In such a solution map, the relationships describe the mutual dependencies of the solutions, which is crucial information in many situations: if a particular solution has to be changed for some reason, which other solutions rely on it? In our future work we plan to explore the generalization of the technique for architectural solutions and for solution-centered documentation of software architectures.

References

1. Alexander, C., Ishikawa, S., Silverstein, M.: A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series). Oxford University Press (August 1977)
2. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. John Wiley & Sons (August 1996)
3. Coplien, J.O.: Software Patterns. SIGS Books and Multimedia (1996)

4. Vesiluoma, S.: Knowledge Sharing Pattern Language. (Eds. Berki, E., Nummenmaa, J., Sunley, I., Ross, M. and Staples, G.). Software Quality in the Knowledge Society. The British Computer Society (2007)
5. Välimäki, A., Kääriäinen, J., Koskimies, K.: Global Software Development Patterns for Project Management. To appear in: Proceedings of EuroSPI 2009 (2009)
6. Hanmer, R.: Patterns for Fault Tolerant Software. John Wiley & Sons (2007)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley Professional (January 1995)
8. Noble, J., Weir, C.: Small Memory Patterns: Patterns for Systems With Limited Memory. Addison-Wesley (2001)
9. Kazman, R., Klein, M., Clements, P.: ATAM: Method for Architecture Evaluation. (2000)
10. Leppanen, M., Koskinen, J., Mikkonen, T.: Discovering pattern language for embedded machine control systems during architecture evaluation methods. Manuscript submitted for publication (2009)