A proposed meta-model for formalizing Systems Engineering knowledge, based on functional architectural patterns

François Pfister, Vincent Chapurlat, Marianne Huchard, Clémentine Nebut, Jean-Luc Wippler

A proposed meta-model for formalizing Systems Engineering knowledge, based on functional architectural patterns

F.Pfister[1,*], V.Chapurlat[1], M.Huchard[2], C.Nebut[2], and J.-L.Wippler[3]

[1]LGI2P, Ecole des Mines d'Alès, site de Nîmes, Parc Scientifique G. Besse, 30000 Nîmes, France
[2]LIRMM, CNRS – Université Montpellier 2, 161 rue Ada, 34095 Montpellier Cedex 5, France
[3]LUCA Ingénierie - 1 Chemin de Pechmirol - 31320 Mervilla, France

ABSTRACT

Design patterns and architecture patterns have been considerably promoted by software engineering. The software oriented tools and methods have been adapted for Systems Engineering, conforming to the model driven engineering paradigm proposed by the Object Management Group. However, designers of complex socio-technical systems have specific concerns, which differ from those of software designers. We propose a method of pattern implementation for Systems Engineering, based on a functional approach and relying on formal conceptual foundations in the form of a meta-model, which can be used for the management, application, and cataloguing of patterns specific to the field of Systems Engineering. A pattern instance in the field of control systems is proposed as an example application.

Keywords: design-pattern, systems engineering, model-based systems engineering, eFFBD

[*] Corresponding author: Francois Pfister : Francois.Pfister@mines-ales.fr, phone: (+33) 466 387 040

1. Introduction

Systems Engineering (SE) frames the design of systems characterized by a high degree of complexity, a hierarchical structure (consisting of subsystems), and strong interaction with their environment. Such systems, which meet stakeholders' needs by transforming inputs and consuming resources, consist of several types of components, including technical and physical components (mechanical, electronic), informational (software), and human (operators, pilots).

The design of such systems, even if they are very innovative, does not occur ex nihilo. The problem solving process carried out by the designers relies heavily on heuristic principles. Heuristics refers to experience-based techniques for problem solving, learning, and discovery. Heuristic methods are used to speed up the process of finding good solutions, when an exhaustive search would be impractical. In engineering, heuristics include leveraging design and architectural patterns, which are not reusable solutions as is, but must be adapted to the specific context of the problem. They are of value because they enable collective experience to be reused, and time to be saved in design processes.

So, even if the whole system seems to be an innovative solution to a given problem, its functions and components are, at a given level of granularity, consistent with patterns of known and tested solutions in response to known and recurring problems. In this way, a system may also be seen as a set of standard solutions interpreted and adapted to a given context. This "pattern vision" is known in many scientific and academic fields (for example in the SE domain, in which we talk about SE Patterns), but it is not always made explicit or formalized.

The questions we would like to address are: 1) What is an SE Pattern? 2) How is an SE Pattern different from other forms of patterns already known in other areas? 3) How can patterns for Systems Engineering be represented, capitalized upon, and reused? 4) Finally, how can they be identified and collected?

In addition to requirement patterns and physical allocation patterns, we hypothesize that an approach based on functional architecture patterns is equally important for knowledge reuse in Systems Engineering. It is therefore necessary to express these functional patterns with a language used by system architects, and propose a meta-model that is compatible and interoperable with those implemented by major SE tools, such as Core [Vitech, 2009]. Other problems raised by pattern implementation in SE tools are methodological issues: at what stage of the design process should patterns be used? These methodological issues also address the question of SE pattern formalization.

As described by Figure 1, a SE Pattern catalog requires a pattern language that formalizes problems, contexts, and solutions. When and if the language is available (first scientific obstacle indicated by Lock #1), a catalog must then be constituted. Using that language, system architects are able to translate their experience to form a catalog. Other non-formal or differently formalized catalogs, and technology watch, are potential sources for creating the catalog. Finally, SE Patterns may be embedded in existing models and should then be identified by pattern recognition mechanisms, which is the second scientific obstacle (Lock #2).

This article presents and illustrates a meta-model for implementing patterns tailored to Systems Engineering, able to manage pattern formalization, collection, and application. Such a SE Pattern meta-model enables both formal and informal pattern representation. It should also help in applying SE Patterns on models being developed. These models must necessarily be expressed using a Systems Engineering meta-model, which is how this paper proposes to make a contribution to this field of knowledge. Finally, our proposition is illustrated by a use case. That contribution is presented in Section 3, after a review of the literature in Section 2. Pattern identification mechanisms are briefly discussed in Section 4. Finally, Section 5 includes the conclusion and points to future work.

[Figure 1 here]

Figure 1: Systems Engineering Pattern Catalog

2. Literature Review

The historical uses of design patterns, their variants, objectives, and formalisms must be examined to determine whether or not they can be reused in the SE field, and to identify some of their limitations. This review is composed of four subsections. The first examines the different design patterns used over time, and in different areas. The second identifies their goals. The third

is about formal pattern representation using several tools and modeling environments, in particular UML and SysML. Finally, since the contribution described in this paper concerns functional modeling architecture, the fourth section reviews functional block languages, such as eFFBD.

2.1 History of Design Pattern Use

Design patterns in their current forms were proposed in the late seventies by architects and city planners, and adopted in the early nineties by software engineers. Ten years later, Systems Engineers began to promote their use.

A well-known promoter of patterns is Christopher Alexander. He is an English architect and urban planner who was born in 1936, having completed numerous architectural complexes in North America and Japan. He questioned the possibility of making original creations in the field of design. In his view, every cultural, artistic, technical, scientific, and legal form is the result of a transhistorical process, which adapts an "ideal" archetypal model. He argues that all creation is simply an imitation of an original pattern [Alexander, C. et al. 1977]. C. Alexander based his theory on functional requirements on the one hand, and cultural models on the other.

The design method promoted by Alexander has been adopted by software engineering [Gamma, E. et al., 1994] [Coplien and Schmidt, 1995] to optimize software architecture in terms of organization of classes. The basic goal is to improve quality, and facilitate code writing by adopting good practices, which was formalized in a book entitled *Design Patterns -- Elements of Reusable Object-Oriented Software*.

At that time, several communities began to manage pattern catalogs in various domains. For instance, in the field of software engineering, the Hillside Patterns Library started a collection of software patterns. These patterns are generally subject to a peer-review process through the PLoP (Pattern Languages of Programs) conference. The patterns submitted undergo a shepherding process [Harrison, N., B, 1999], in which they are analyzed and modified before being presented at the conference. The roles and procedures used to frame and make design patterns evolve are greatly influenced by the work of the Hillside Group. The PloP pattern language is essentially textual.

PloP authors emphasize the literary quality of the pattern descriptions; for example, problem statements are carefully crafted into well-defined patterns; if the solution is the heart of the pattern, the problem is its soul [Meszaros and Doble, 1997]. Problem statements for many patterns are too vague, and sometimes the problem statement is solution-oriented. Pattern authors

often begin by writing the solution, and the problem comes only as an afterthought. Alternatively, authors begin by writing the problem, and the way the problem is formulated tends to be vague and less precise. Pattern authors also emphasize the notion of forces. Breaking down the forces helps to formulate a clear and in-depth description of the problem solved by the pattern by decomposing that problem. In fact, the problem perceived is a symptom of something that is wrong. The forces at play give substance to the problem, and insight into what is behind the symptoms. The author should go back and forth between the forces and the problem statement to improve both of them.

The purpose of PloP is to promote software patterns, but it is also seeking subjects in domains beyond software. Topics covered include for instance Telecommunications Distributed Processing Patterns and Design Patterns for Avionics Control Systems, which are close to System Patterns.

Following Gamma and Coplien, Martin Fowler proposes analysis patterns [Fowler, 1996], which are not about how code is organized, but about the structure of models that represent real systems. In the area of workflow management [Van der Aalst, W. et al., 2003] identifies 21 different patterns that describe the behavior of business processes. These patterns, which are described on a website, represent configurations encountered in business processes in general. They were originally expressed with a specific language (YAWL), but can be transformed into UML Activity diagrams [Bock, 2006] or BPML diagrams. [White, SA, 2004].

However, Van der Aalst patterns are abstract structure patterns (choice, sequence, parallelism, choice, interleaved parallel routing, milestone ...), but they are not as such reusable in concrete business cases or scenarios. These patterns describe generic transverse configurations.

Meanwhile, other patterns, which reflect concrete processes in a particular domain, such as the PDCA pattern [Appleton, B., 1997] [Cloutier, R. & Verma, D., 2007], documents a pattern that can be implemented in the field of process management.

In the field of Systems Engineering, patterns have been the subject of many publications within INCOSE. The first article on this subject [Barter, HR 1998] proposed a language of patterns for SE. A pattern language was subsequently proposed for writing [Haskins, C., 2003] a Systems Engineering manual: the Systems Engineering Book of Knowledge (SEBOK). Inter-pattern relations and the establishment of a pattern map have been proposed by the same author [Haskins, C., 2005]. A proposal for an engineering paradigm based on patterns (PBSE) [Schindel, WD, 2005] was followed by other studies [Cloutier, R., 2006] [Cloutier et al., 2010]. The degree of formalization of these proposals is not yet sufficient, which is what motivated our research and

this paper in particular.

2.2 Design Pattern Goals and Typology

All these variants have similar objectives: a design pattern is a way practitioners can represent invariant knowledge and experience in design. It can help humans to identify and solve problems by drawing or imitating such knowledge and experience. The objectives are:

- to improve performance (comprehensiveness, relevance), reliability (proven solutions, justified and context-based),
- to gain economic value (time savings) and,
- to facilitate collaborative work by sharing design pattern repositories.

These objectives can be achieved by leveraging and integrating knowledge and good practices.
A design pattern is a simple and small artifact, rarely isolated and therefore correlated with other ones; not a creative method (by definition, it exists only if the solution it proposes is well known and frequently used in the field and, therefore, is not innovative). In the same manner, it is not a reusable component. It is destined to be imitated and adapted to a particular context.
One characteristic of design patterns is that they do not alter the functionality of the models to which they apply. Indeed, a design pattern improves the quality of the current model of a system under design. A design pattern improves non functional features, quality of service and systemic properties. [Manola, F., 1999] names "-ilities" the operational and support requirements a system must address (availability, maintainability, vulnerability, reliability, supportability …). Often, gaining some "-ilities" is done at the expense of some others: e.g. *The Spitfire was designed with an elliptical wing, giving greater speed and maneuverability (perhaps the most critical "-ility" of all for a warplane). But this came at a price: 13000 man-hours per airframe. Willy Messerschmitt had optimized the German Bf 109 for speed and manufacturability at only 4000 man-hours per frame, but the Bf 109 was no faster than the Spitfire and was consistently out-turned by it. The elliptical wing had been considered but ultimately rejected as too difficult to manufacture* [Alexander, I., 2005].
Different categories are used to construct taxonomy of patterns:

1. Idiomatic patterns. Describe low level elements that structure a model, govern associations between these elements, and define aggregation and containment strategies (e.g. Composite Pattern). In the software engineering domain, GOF patterns [Gamma et al., 1994] could be classified as such. Workflow patterns, such as the ones proposed by [Van der Aalst, W. et al., 2003], also belong to that family.

2. Generic patterns. This category of high level patterns cuts across several areas. e.g. command-control loop, active redundancy, event queuing, error detection and correction. The example described below illustrates generic patterns.

3. Domain specific patterns. This category of patterns is specific to particular industrial or organizational areas. These patterns, although related to particular technical fields, are however, at a high level, the responsibility of system designers. Many patterns coming from the Hillside repository, as well as Fowler analysis patterns, are domain specific patterns.

Patterns may apply to the system of interest (SOI), or to systems engineering activities (SEA) themselves (which aim to produce a model of the system of interest). In the first case, we speak about SOI patterns, in the second about SEA patterns. Leveraging patterns is efficient for each of three main Systems Engineering activities - requirement engineering, functional architecture design, and physical architecture design. We argue that functional architecture patterns are key elements for knowledge reuse.

[Cloutier and Verma, 2007] proposed a system architecture taxonomy, in which patterns are broken down into:

1. Structural patterns - provide a physical pattern to follow when designing a part of the architecture.
2. System Requirements patterns - prescribe the format of a properly formed requirement, or a collection of requirements that can be reused to describe desired functionality.
3. Systems Engineering Activities patterns - also described as systems engineering process patterns, indicate how the process of architecting or systems engineering is performed.
4. Systems Engineering Roles patterns - help describe how the architecting/engineering role is performed.
5. System Process patterns - capture how the system does what it does by using control loops, algorithms, etc.

Cloutier's taxonomy seems to lack consistency due to a mixing of SOI patterns and SEA patterns: type 1 and type 2 are relevant to SOIs, type 3 and type 4 are relevant to SEAs, and type 5 is relevant to SOIs.

This paper addresses only SOI patterns and focuses on functional architecture (Type 5 in Cloutier's taxonomy), whether generic or domain specific targets.

2.3 Design pattern formalization

Design patterns have been formalized to varying degrees. Often, this formalization is only a

standard text template [Portland Pattern Repository]. In contrast, other software design tools based on UML have implemented design pattern catalogs equipped with a true meta-model [Galic, M. 2003].

Within models, design patterns apply a crystallization process [Jézéquel et al., 2005 in French]. Impacted entities fit together in a configuration to meet specific roles defined in the pattern. Design patterns are defined in UML or SysML as parameterized collaborations. They specify a set of classes and objects that have specific roles and interactions. Generally speaking, a parameterized collaboration is used when the actual classes work in the same way as the collaboration classes, but the classes and operations are named differently. Collaboration is a name given to the interaction among two or more classes. Typically this is an interaction as described in an interaction diagram [Larman, 2001]. Mechanisms such as inheritance, delegation, and implementation are used to give rise to these collaborations, which will be captured by use cases as well as through interaction diagrams. [Jacobson, 1997] [Sunyé 2000].

Applying a design pattern amounts to generating or correcting part of a model by applying a prototype [Barcia 2006]. Transposed to SysML, a design pattern could be described as an internal block diagram, which comes with a sequence diagram. An activity diagram (functional view) and a state diagram (behavioral view) can be supplied. When an actual model is in the design phase, a part of it, revealing a design challenge, may require the application of a design pattern. Once the design pattern is chosen from a model repository by the domain expert who is currently working on this model, a pattern instance is parameterized with model entities involved (i.e. blocks, components, objects). As soon as the design pattern instance is applied to the model, the model entities are reconfigured to comply with the design pattern (their structure and composition may be amended, supplemented or renamed, constraints can be added). The process takes place as a local model modification: the latter mimics the design pattern applied. The design pattern instance persists within the model as a parameterized collaboration, and can be used at a later point (e.g. for the purposes of documentation).

UML has achieved a good level of formalization for software and analysis design patterns, based on class diagrams and object interaction diagrams. This paper aims to build upon this work to apply a similar approach adapted to the functional design of non software systems.

[Figure 2 here]

Figure 2. A System Pattern Meta-model

2.4 Function-block Languages

One of the problems is to fit a pattern language to a Systems Engineering language: eFFBD is a widely used formalism in SE [Bock, 2003] [Vitech, 2009]. An eFFBD (Enhanced Functional Flow Block Diagram) is a functional flowchart. An eFFBD can be considered as a superposition of an FFBD (Functional Flow Block Diagram) and a DFD (Data Flow Diagram) [Long, J., 1995].
FFBD formalism has been used for a long time in Systems Engineering. It was developed in the 1950s by TRW Incorporated, and applied in the 1960s in the fields of aeronautics and space [Blanchard, B. & Fabrycky, W., 1990]. It shows the functions of a system and their execution sequence, but does not mention data streams, whether single or triggering.
A DFD [DeMarco, T., 1979] represents system functions and the data flows processed by these functions, which absorb input data and produce output data after processing. The DFD does not represent the control structures that drive the activation and sequencing of the functions. An equivalent formalism is IDEF0 [Ross, DT & Brackett, JW, 1976], which refines the types of data streams by distinguishing inputs, outputs, mechanisms, and controls.

[Figure 3 here]

Figure 3. A Systems Engineering Meta-model

An eFFBD represents functions, a control flow in the form of an FFBD, overlaid on a data flow in the form of a DFD. An eFFBD distinguishes triggering- and non-triggering data entries. Triggering data have control implications. Non triggering data are represented with a single arrow, while triggering data are designated with a double arrow. An eFFBD has an operational semantics allowing it to run as a discrete event model: a function must be validated (by the completion of the previous function execution in the control structure) and triggered (if a data entry is triggering data). The designer, therefore, has the ability to specify executing conditions of the functions, either by the use of control structures, or by triggering data, or a combination of the two semantics. The semantics of an eFFBD is completed by the notion of execution duration (min, max, average), as well as execution costs or resource constraints. [Seidner, C. et al, 2008] propose to translate eFFBD into Temporal Petri Nets [Petri, C., 1962].
This review of the literature does not reveal any form of design pattern well suited to functional architecture in the area of Systems Engineering. Adapting design patterns to SE (SE Patterns) begins by developing a supporting meta-model. SE patterns must be function oriented, thus such a meta-model has to include function graph structures. In a Systems Engineering perspective,

functions compose operational scenarios, are allocated to components, and related to requirements which are themselves related to needs. Thus a minimalist Systems Engineering meta-model comes as an infrastructure for the SE Pattern meta-model.

3. Contribution

The goal of this section is to formalize patterns for Systems Engineering, going beyond the stage of textual documentation, by proposing a SE Pattern meta-model (Figure 2) extending a Systems Engineering meta-model (Figure 3). An example of a SE Pattern is then given as illustration (Figure 4). In the following subsections, terms in italics refer to entity names found in Figures 2, 3, and 4.

3.1 SE Pattern meta-model

Our domain of interest is the representation of system engineering projects. A *Project* consists of several models each being a representation of the system under study and of a catalog which lists system patterns. A *SystemPattern* formally and informally identifies and documents a *Solution* addressing a *Problem* in a given *Context* that has been tested and deemed to be safe. These three concepts: *Problem*, *Solution*, *Context*, constitute a triangular pattern definition, which represents the core meaning of a *SystemPattern*. If any of the three elements is missing, this will result in a trivial pattern [Gaffar, A. & Moha, N., 2005]. A *SystemPattern* is defined by at least the following characteristics: a unique identifier, a short but evocative name, alternative names as aliases, a creation date, a textual description and an author.

A *Problem* formally or informally describes the design problem motivating the *SystemPattern*. Each *SystemPattern* addresses one and only one *Problem*. A *Problem* is characterized by an informal description, a *Feature* to be optimized, a set of competing *Forces*, a use case *Model* showing a trivial or a poor functional and/or organic architecture. A *Force* is a competing constraint in a System: design problems arise from a conflict between those different interests or "Forces". The *SystemPattern* application decision depends on arbitration between the Forces. [Alexander, C. et al., 1977] gives the example of a conflict between the need for a sunny environment in a building, but not to be overheated in the summer. In this example, the Force is described by a *challenge* (the need for a sunny environment), a *constraint* (not to be overheated in the summer) and a *ProblemType* (Fluid, Field, Structure, Security...). A *Feature* is an extra functional characteristic, also named “-ilities” by [Manola, F., 1999]: availability, maintainability, vulnerability, reliability, supportability…).

A *Solution* contains a pattern *Model*, which is a parameterized system architecture. It represents a design solution as a response to a *Problem* considering the given *Context*. There is only one solution for one pattern, but one *Problem* may have many solutions through several patterns by using equivalent-patterns and/or related-pattern relations. A *Solution* is illustrated by a use case showing a better architecture, resulting from the pattern application. The solution has an *Impact* that characterizes the influence of a *SystemPattern* on a model to which it applies. Impact(s) are quantified with a *VariationSense* (increase, decrease, equals) and a *value* on a *scale*. They are similar to post conditions, after a pattern is applied. The impact is measured on a feature. The *Feature* to optimize is often gained at the expense of another. For example, security may be improved (in this case the attribute *variationSense* of feature F1 is set to "increase") at the expense of manageability (the attribute *variationSense* of feature F2 is set to "decrease").

The *Context* expresses the core meaning of the pattern i.e. may be interpreted as a (set) of pre-condition(s). It indicates the situation to which the *Solution* may be applied, and the required conditions that must be checked before the pattern can be applied (informally, in the *description* attribute).

The other main relations between a *SystemPattern* and other concepts from the meta-model are:

- Since a *SystemPattern* is a parameterized architecture, each of its *Parameters* associates one of its own *ModelElements* to a *ModelElement* belonging to the model under work, e.g. Function, Component, Item, Interface, DataFlowConnection, Need, Scenario or Requirement.
- A *SystemPattern* is legitimated when mined in several well known applications (defined as *knownUses*).
- A *requestedPattern* is a *SystemPattern* required when applying a given *SystemPattern*. All *requestedPatterns* are also *relatedPatterns*.
- A *relatedPattern* is a *SystemPattern*, often present when a given pattern is applied. Within a triangular association *Problem Context Solution*, related patterns often have the same context, but *relatedPatterns* exclude *antiPatterns*.
- An *antiPattern* is in opposition with the *SystemPattern* of interest in a given case. Within a triangular association *Problem Context Solution*, anti patterns have the same problem and the same context.
- *Equivalent patterns* are patterns that have the same problem and the same context. In this case, the textual description may be more formalized in the solution/model/needs/description.

The *Domain* identifies a specific area in which a *SystemPattern* can be applied or is relevant e.g.

mechanics, electronics, software, civil engineering, organization & service, security, pedagogy...

The *Rationale* justifies the *SystemPattern* by an explicit description and the associated argumentation that justifies applying the pattern. It is different from *knownUses* in several *Application*, which are statistical observations.

Problem, *Solution*, *Context*, *Application* and *Rationale* are *Indexable* objects, described by *keywords*.

[Figure 4 here]

Figure 4. Application of a System Pattern to an Incompletely Designed Model

3.2 Example of how an SE Pattern is Applied

The example in Figure 4 aims to show the *SystemPattern* instantiation and application mechanisms (however it is a simplified example and is no a reference in the field of regulation). The example shows a *SystemPattern* that is applied in three phases:

- The model under work in which there is the problem (A)
- The SE Pattern chosen to be applied (B)
- The model under work after the SE Pattern has been applied (C).

The three models are represented using eFFBD notation, which is supported by the meta-model shown in Figure 2 and Figure 3. These models could be transformed into SysML activity diagrams using the eFFBD profile, the two notations are isomorphic.

The *Fx* titled rectangles are system functions (*Function* meta-class), connected by a control flow (solid line) (*ControlFlowConnection* meta-class). The rounded rectangles represent the nature of the processed data flow (dashed line) (*Item* meta-class). The graph formed by the control flow and the functions is structured by control nodes (*ControlFlowConnection* and *ConnectionType* meta-classes).

The model containing the problem (A) is an instance of the *Model* meta-class, associated with the *Project* meta-class by the *modelUnderWork* meta-association.

When applying the pattern on the model (A) to produce the model (C), the original version (A) is preserved and remains accessible by the *priorVersion* meta-association. The *SystemPattern* model (B) is an instance of the *Model* meta-class, associated with the *Solution* meta-class by the *patternModel* meta-association.

Model containing the problem (A)

The example of Figure 4 (A) shows a fragment of an industrial food process. Raw materials are placed in a container where a given environment is maintained, with parameters such as temperature and hygrometry. An air flow is provided by blowing air. The material to be processed is subjected to lactic fermentation, and also desiccation. The lactic fermentation is intended to provide the final product organoleptic qualities and to lower the pH to improve conservation. This process fragment is connected to a global process, which is not shown, by the *maturation start* entry link as well as by the *maturation finished* output link. It consists of four simultaneously active functions (*and* nodes before and after their parallel activity).

To achieve a goal of *raw product* maturation, four functions must work together:

- *Contain*: the *raw product* must be contained. The *Contain* function is fed by *raw product* and *air stream*. At the end of the process (*refinement finished),* the *Contain* function ends, and the processed *raw product* is outputted (*refined product*).
- The input flow (*raw product*) and the output flow (*refined product*) are discrete flow (path in the Figure 3 meta-model: *DataFlowConnection, carries, Item, flow_type, discrete*).
- Flows as *air stream*, *residual air stream*, *calorific energy*, and *residual heat* are continuous flows (path in the Figure 3 meta-model: *DataFlowConnection*, *carries*, *Item*, *flow_type*, *continuous*).
- *Blow*: This function converts *electrical energy* into an *air stream*. This outputted air *stream* is absorbed by the *Contain* function. The latter retransmits it as a *residual air stream* which is the input of the *Blow* function, and this is a closed circuit.
- *Heat*: This function absorbs *electrical energy* to provide *calorific energy*. The latter is provided as input to the *Contain* function. The *Heat* function also consumes a given *heating command*. The latter is used by the function to modulate its *calorific energy* output.
- *Control Temperature*: this function produces the *heating command* which will be absorbed by the *Heat* function. It absorbs a *Contain* output: *residual heat*, to compute, given the *temperature profile* and *cycle mode* inputs, the *heating command* output. This function also determines the end of the process.

When all parameters are met (for example time * temperature * mass), this function ends and forces termination of the three remaining functions in the *and* parallel branch.

The problem in this model is one of incompleteness. To finalize this model, the *Control Temperature* function has to be decomposed. This function delivers a control data proportional to

the difference between the *residual heat* output and the *temperature profile* setpoint so that the *residual heat* temperature reaches the setpoint.

Pattern solution (B): control-Command pattern semantics

The *Control-Command* pattern describes a regulation process in which an *Actuate* function delivers an *effect* output which is directed to an external function. This *effect* output modifies the external environment, of which, in return, one dimension is reinjected into the *Measure* function as the *position* Item.

Until the process represented by the model is not ended, *Actuate* and *Measure* functions are repeated in loop structures.

Another cycle in this model consists of three sequential functions: *Acquire Measure, Compute Command, Drive Actuator*. The *Acquire Measure* function absorbs *measured value* data to transmit it after conversion as *acquired measure* to the *Compute Command* function. The *Compute Command* function compares a *set point* with *acquired measure* to issue a *command* output; c*ommand* is fed into the *Drive actuator* function, together with a given *energy*, to produce a *command to apply* output. The latter is fed into the *Actuate* function that produces the *effect* as an output.

This pattern is a high level pattern, independent of the food processing area. It implements a control loop based on the feedback of the difference between a *setpoint* and a *position*, and the resulting action is proportional to that difference. This pattern (which is here a simplified version) is proven and recognized by all regulation experts (Bennett, S. 1986) in various fields.

Pattern parameterization and application

A, B, and C models are described in Figure 4. The application of *SystemPattern* (B) to source model (A) containing the problem results in a transformation from the model (A) into a target model (C) containing the *SystemPattern* solution.

The target model mimics the *SystemPattern* model. It is a parameterized collaboration of *ModelElement*. Some *ModelElement* belonging to model (A) will play given roles in the *SystemPattern* model (B).

A model transformation taking as input model (A) to be corrected, model (B) to imitate, and the parameter list given in Table 1, will produce the target model (C). Some elements of the *SystemPattern* model are renamed in the resulting target model. *ModelElement* related to *Parameter* has the names assigned to the *concreteRole*. Additional elements are renamed according to Table 2. Elements whose names are unchanged are shown in Table 3. Finally, one

element, the *ControlTemperature* function, is removed after the *SystemPattern* is applied.

Model containing the solution (C).

Model (C) containing the solution provided by the *SystemPattern* (B) imitation is a reformulation of model (A) in which there is the problem. The overall function of model (C) is identical to that of model (A), but non-functional features are improved; namely, temperature control is optimized.

ModelElement	**Parameter patternRole**	**Parameter concreteRole**
ControlFlowConnection	process start	maturation start
ControlFlowConnection	process end	maturation finished
Function	Actuate	Heat
Item	effect	calorific energy
Item	position	residual heat
Item	operational mode	cycle mode
Item	command	heating command
Item	set point	temperature profile
Item	energy	electrical energy

Table 1. System Pattern Parameters

ModelElement	**Name in SystemPattern**	**Name in model after pattern application**
Function	Acquire measure	Acquire temperature
Function	Compute command	Compute heating
Function	Drive actuator	Drive heating
Function	Measure	Measure temperature
Item	acquired measure	acquired temperature
Item	command to apply	heating power
Item	measured value	measured temperature

Table 2. ModelElements added to Model (A) after Application of SystemPattern (B)

ModelElement	Name
Function	Contain
Item	air stream
Item	raw product
Item	refined product
Item	residual air stream

Table 3. ModelElements Unchanged in Model (A) after Application of SystemPattern (B)

4. Pattern identification

This section is a brief discussion of pattern identification. This point is the second lock identified earlier in this article; it will be developed in a subsequent paper.

To be listed in catalogs, patterns must first be identified (mined) within models or systems in which they were applied, and where they are buried. Reverse engineering can also motivate the need for pattern identification. In order to identify buried patterns, models can be searched for within occurrences of particular identified patterns, in order to verify whether or not these patterns were applied. One can also look for any recurring structures, assuming they are patterns. But then there is no evidence that they are not anti-patterns.

The difficulty lies, first of all, in the heterogeneity of notations and languages. In addition, MBSE techniques that facilitate model analysis are relatively recent and formal SE models have only been available for a few years.

Pattern mining techniques differ depending on whether the models to investigate are formal models (eFFBD, SysML, UML), for which there is a meta-model, or on the contrary textual or graphical unparsable models.

Product models, which are not SE models, but technical models, such as schematic diagrams or mechanical models produced by different computer aided design (CAD) tools, can also be analyzed to identify standard architectures.

In the case of older and less formal models, pattern recognition is done by experienced practitioners, who identify recurrent structures using reverse engineering processes. These patterns are then formally added to catalogs.

In the case of formal models, underpinned by meta-models, mining relies on techniques of subgraph recognition.

The matches are not exact: there will be similarities that will be measured. The techniques are

those of model alignment [Euzenat and Valtchev, 2004], [Melnik et al., 2002]. Models to be aligned are, on one hand, models in which patterns are supposed to be buried, and on the other hand, pattern models from existing catalogs.

The alignment algorithms seek to match the entities of each model based on semantic similarities (names), obtained directly or indirectly through the use of glossaries or dictionaries, and also on structural similarities, seeking similar paths between similar nodes within two graphs under study [Noy and Musen, 2001].

In the case of patterns based on functional notation such as SysML Activity Diagram or eFFBD, it will be necessary to construct graphs by transforming the functions or activities into nodes, and the control flow and data flow into edges. It will then be possible to apply the appropriate efficient alignment methods developed by the knowledge engineering community.

5. Conclusion

Systems Engineering conforms to the Model Based initiative [Estefan, J., 2008], and adopts languages and tools derived from those used by software engineering. However, design patterns, as understood in software engineering, are not adapted to the field of Systems Engineering. Indeed, the SE approach should be a functional approach, and such patterns should be modeled as parameterized functional micro-architectures. This paper proposes to develop a specific approach for Systems Engineering, based on the eFFBD formalism which is widely used in this area. A meta-model that is interoperable with the main tools used by system architects has been designed.

Future work will consist in the following tasks: first, we will create an editor through which a catalog of System Patterns can be built, and develop mechanisms (based on model transformations) for applying patterns. Then, the presented approach will be developed to extend these System Patterns to allocation patterns that may help experts design better physical architectures. Finally, pattern detecting mechanisms based on model alignment will be implemented to complete the project.

6. References

Alexander, C. et al., 1977. A pattern language: towns, buildings, construction, Oxford University Press.

Alexander, I., 2005. Systems Engineering: -ilities for Victory. Available at: http://easyweb.easynet.co.uk/iany/consultancy/systems_engineering/ilities_for_victory.htm.

Ambler, S., 1998. Process Patterns building Large Scale Systems using Object technology Cambridge University Press., SIGS Books.

Portland pattern repository. Available at: http://c2.com/ppr [Accessed Feb 9, 2011].

Appleton, B., 1997. Patterns for Conducting Process Improvement. In Proceedings of the 4th Annual Conference. Pattern Languages of Program Design (PLoP'97). Urbana Champaign.

Barcia, R. & Gerken, C., 2006. Get started with model-driven development using the Design Pattern Toolkit. IBM WebSphere Developer Technical Journal.

Barter, R.H., 1998. A Systems Engineering Pattern Language. In Proceedings. 8th Annual International Symposium of the International Council on Systems Engineering. Vancouver.

Bennett, S., 1986. A history of control engineering, 1800-1930, Institution of Engineering and Technology.

Blanchard, B. & Fabrycky, W., 1990. System Engineering and Analysis, Prentice Hall.

Bock, C., 2003. UML 2 activity model support for Systems Engineering functional flow diagram. Systems Engineering, 6, p.249–265.

Cloutier, R.J., 2006. Applicability of Patterns to Architecting Complex. Hoboken, NJ: Stevens Institute of Technology.

Cloutier, R.J. & Verma, D., 2007. Applying the concepts of patterns to systems architecture. In Systems Engineering. Wiley, p. 138-154.

Coplien, J.O. & Schmidt, D.C., 1995. Pattern Languages of Program Design, Addison-Wesley Professional.

DeMarco, T., 1979. Structured Analysis and System Specification, Upper Saddle Rive, NJ, USA: Prentice Hall.

Estefan, J.A., 2008. Survey of Model-Based Systems Engineering (MBSE) Methodologies, Seattle, WA - U.S.A.: INCOSE.

Euzenat, J. & Valtchev, P., 2004. Similarity-based ontology alignment in OWL-Lite. In European Conference on Artificial Intelligence - ECAI04. Valencia, Spain, p. 333–337.

Fowler, M., 1996. Analysis Patterns, Addison-Wesley Professional.

Gaffar, A. & Moha, N., 2005. Semantics of a Pattern System. In TEP International Workshop on Design Pattern Theory and Practice (IWDPTP05). Budapest.

Galic, M., 2003. Applying Pattern Approaches, IBM.Com/Redbooks.

Gamma, E. et al., 1994. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley.

Harrison, N., B, 1999. The language of shepherding. A pattern language for shepherds and sheep. In Proceedings. 7th Pattern Languages of Programs Conference PLoP.

Haskins, C., 2005. Application of Patterns and Pattern Languages to Systems Engineering. In INCOSE

15th Annual International Symposium.

Haskins, C., 2003. Using Patterns to Share Best Results – A Proposal to codify the SEBOK. In Proceedings. 13th Annual International Symposium of the International Council on Systems Engineering. Washington DC.

Jacobson, I., Griss, M. & Jonsson, P., 1997. Software Reuse: Architecture, Process and Organization for Business Success, Addison-Wesley Professional.

Jézéquel, J.M., Plouzeau, N. & Le Traon, Y., 2005. Développement de logiciel à objets avec UML, Université de Rennes 1.

Larman, C., 2001. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, Prentice Hall PTR.

Long, J., 1995. Relationships between Common Graphical Representations in System Engineering. In 1995 INCOSE Symposium.

Manola, F., 1999. Providing Systemic Properties (Ilities) and Quality of Service in Component-Based Systems. Available at: http://www.objs.com/aits/9901-iquos.html.

Melnik, S., Garcia-Molina, H. & Rahm, E., 2002. Similarity flooding: A versatile graph matching algorithm. In Proceedings. 18th International Conference on Data Engineering. San Jose, CA, U.S.A: IEEE Computer Society, p. 117-128.

Meszaros, D. & Doble, J., 1997. A Pattern Language for Pattern Writing. In Pattern languages of program design. Addison-Wesley, p. 529-574.

Noy, N.F. & Musen, M.A., 2001. Anchor-PROMPT: Using Non-Local Context for Semantic Matching. In IJCAI - workshop on ontology and information sharing. Seattle, WA, U.S.A., p. 63-70.

Petri, C.A., 1962. Kommunikation mit Automaten. Bonn: Institut für instrumentelle Mathematik.

Ross, D.T., 1977. Structured Analysis (SA): A Language for Communicating Ideas. , 1(3), p.16-34.

Schindel, W.D. & Rogers, G.M., 2000. Methodologies and Tools For Continuous Improvement of Systems. Journal of Universal Computer Science, p.289-323.

Seidner, C., Lerat, J.-P. & Roux, O.H., 2008. Formal Verification in System Design Process: from EFFBDs to Petri nets. In 18th International Symposium of the INCOSE (IS2008). Utrecht.

Sunyé, G., Le Guennec, A. & Jézéquel, J.M., 2000. Design Patterns Application in UML. In ECOOP. Sophia Antipolis, France: Springer.

Van der Aalst, W.M.P. et al., 2003. Workflow Patterns. Distributed and Parallel Databases, 14(3), p.5-51.

Vitech, 2009. CORE Architecture Definition Guide (DoDAF v1.5).

White, S., A., 2004. Workflow Patterns with BPMN and UML. IBM.