A Problem-Oriented Approach to Critical System Design and Diagnosis Support

Vincent Leildé[2], Vincent Ribaud[1], Ciprian Teodorov[2], and Philippe Dhaussy[2]

[1] Lab-STICC, team MOCS, Université de Bretagne Occidentale, Avenue le Gorgeu, Brest, France Vincent.Ribaud@univ-brest.fr,
[2] Lab-STICC, team MOCS, ENSTA-Bretagne, rue François Verny, Brest, France firstname.lastname@ensta-bretagne.fr

Abstract. For critical software applications, dependability and safety are required features that should respect security principles. To cope with these constraints, the design activity should use new methods that foster knowledge sharing and reuse, in particular security problems and their solutions. In this paper, we present a problem-oriented method that follows a step-wise building of the solution. Problems are reused using various mechanisms, and a solution is conceived, verified and diagnosed. We illustrate the approach on the problem of building a secure SCADA architecture.

Keywords: Problem Oriented Method, Diagnosis, Security Patterns

1 Introduction

Critical software systems are pervading our daily lives and sustaining in many different domains (transportation, avionics, health-care or information management). To improve their dependability and safety, regardless their complexity, critical software design should be carried out with respect to security principles.

Over time, knowledge about security has been captured into patterns, a packaged solution to a recurrent problem in a specific context [7]. A security pattern is a reusable solution for a recurring security problem. It is used to analyze, construct and evaluate secure systems [6]. It provides detailed guidelines about the application of an architectural solution for a particular problem of security. Several research work address security issues using security patterns, and we exploit as a case study the approach set by F. Obeid [14]. The author secures SCADA systems through the composition of the SCADA architecture with security patterns. Safety and security requirements of the composition are then validated through model-checking.

Our research work is focused on methods and tools intended to ease this verification activity, especially the diagnosis activities. Briefly stated, our approach aims to answer diagnosis issues with a general diagnosis ontology [12], a management system to perform and enable verification and diagnosis activities [11], and a domain-oriented method [13].

Verification at the early stages of the engineering process prevents expensive defects from occurring in the final product. A software organization that manages quality should have a corporate infrastructure that links together and transcends single projects by capitalizing on successes and learning from failures [3]. These tasks require to manage past diagnosis experiences (gathering a set of heterogeneous artifacts) and to correlate discovered abnormalities with experiences. This can be achieved with a knowledge based system together with a well-defined method. This paper we present how our method and tools support the application of security patterns as set in the case study mentioned above.

To some extent, the method we use in this paper borrows the Twin Peaks idea of performing round trips between problem and solution spaces for improving the verification process [9]. It should help the engineer to bring closer high-level information and abnormalities observations. It focuses on a progressive constitution of a knowledge base, containing both problems and solutions, that can be reused. Solutions are packaging formal designs and verification runs, and problems are formalized with a set of properties together with a structure of various solutions.

Section 2 overviews the method. In section 3, we shows the application of the method to secure an application using security patterns. Section 4 discusses how problem cases is a support to design and diagnosis, and section 5 concludes this study.

2 Overview of the Method

The method focuses on a progressive understanding of the problem. First, this should help the designer to find rapidly a solution to his problem, by decomposing the problem in smaller subproblems, and reusing existing solutions. Second, it should help the verifier to understand the root causes of abnormalities for a selected solution, by providing diagnosis task with relevant information. This section describes the process used to formalize problems and the different steps of the process flow.

The step-wise method is presented by the activity diagram in figure 1. The method is reiterated until a satisfactory solution is achieved.

To illustrate the method, let us consider the following example. Suppose a board game with one board and two players. The board asks an infinite number of questions to each player, in a non deterministic manner. If the player has a right answer, it increases its score by one point, otherwise no point is awarded. The match ends when a player reaches 3 points. This model is not fair because in some case, the board can ask more questions to one player rather the other.

(1) The problem is formulated as a set of properties and constraints (architectural, technical choices). The structure of formulated problem conforms to the conceptual model of figure 5 and is presented in section 4.1. For instance, "at the end of the game, each player has played the same number of times". (2) The problem is decomposed into subproblems, either known problems - called problem cases - selected from a knowledge base, or unknown situations. For in-

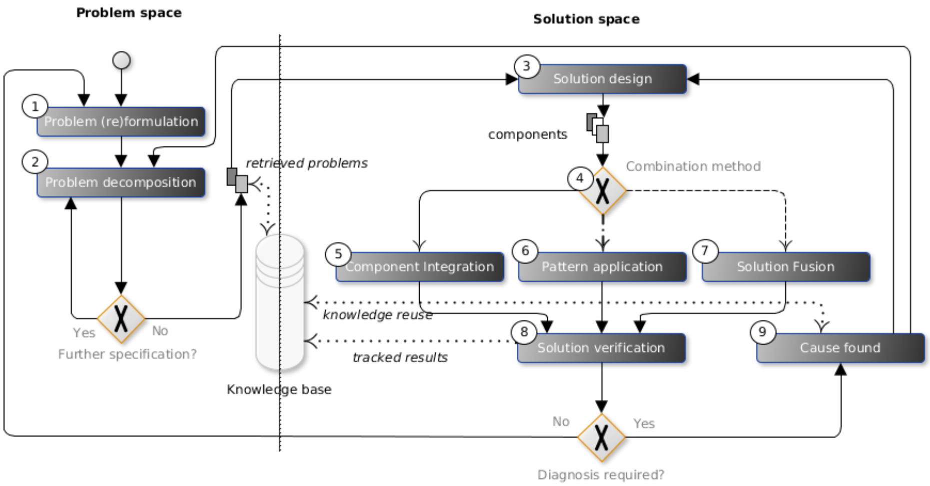


Fig. 1: Method steps

stance, we may decide that ”a turn mechanism is used”. (3) When the need for a concrete view occurs, we move towards the solution space. The potential solution elements are organized. For instance, we may decide that ”the turn problem case is introduced into the current solution”. (4) We consider how to combine the selected problem cases together. This solution may be either composed (5) with other problem cases, applied as a pattern (6), or problem cases may be merged (7) [3]. (8) At this point, we built a part of the expected solution; hence we are able to start a verification cycle. When abnormalities are observed, it triggers a diagnosis process. Verification results are stored in the knowledge base. (9) Diagnosis process is performed, problem cases can be used to enhance this process. The design is corrected, and the verification endeavor repeated. In some cases, the selected problem cases do not suit, hence we have to backtrack and rework the problem combination, and it might be useful to keep track of this failed attempt.

This step-wide method is repeated several times while useful components can be combined. The engineer is left with a reduced problem for which no known solutions exist and where a classical design and verification activity has to be achieved.

The method space is divided in two parts, the problem space, related to the problem elaboration, and the solution space, related to design and verifying the solution. Whereas the problem elaboration produces specification to the solution design, the resulting solution produces expanded specifications (from design choices) to the problem space. This is similar to the Twin Peak model

[3] Each kind of combination is represented with a particular arrow shape.

[9], a software iterative development process that focuses on the combination of problem structures and solution structures.

The method applicability is illustrated in the next section onto a critical system design. Let see now how to use the method on the securized SCADA case study.

3 Application

We reuse a case study extracted from the work of [14]. The approach aims at securing architectures by applying security patterns together with a security policy. The approach has been demonstrated on several kinds of architectures.

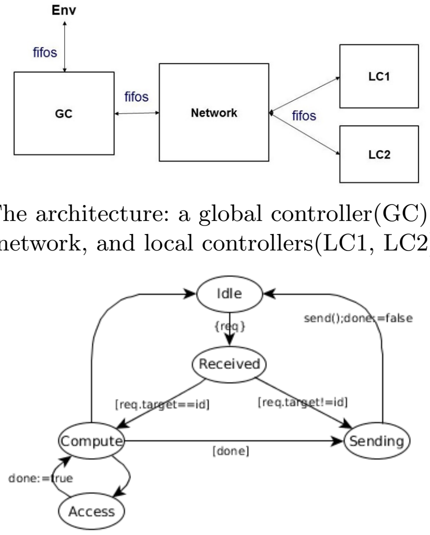
3.1 Problem Formalization

The figure 2 represents an unsecured architecture composed of four entities, a global controller (*GC*), two local controllers (*LC1 and LC2*), and a communication network (*NETWORK*) that links together *GC*, *LC1* and *LC2*. The local controller *LC1* owns the resource *RES1*, while the local controller *LC2* owns the resource *RES2*. Some *READ* and *WRITE* operations for accessing a resource are granted for an entity according to its role. Different roles are *ADMIN* (*READ* and *WRITE* access to *RES1* and *RES2*), *GCOWNER* (*READ* access to *RES1* and *RES2*), *LC1OWNER* (*READ* and *WRITE* access to *LC1*) and *LC2OWNER* (*READ* and *WRITE* access to *LC2*).

This architecture can be seen at a higher abstraction level, as a set of *NET* and *ACCESS* components. A *NET* is an abstraction of a *NETWORK* entity that forwards messages to other components. An *ACCESS* component is an abstraction of *LC1, LC2* and *GC* entities protect access to resources. An *ACCESS* component behaves as depicted in figure 2.

In this figure, the transitions between states bear the Event-Condition-Action expression represented as $Si \xrightarrow{\{Event\}[Condition]Action} Sj$. *Si* and *Sj* are *states*, the arrows stand for *transitions*, labeled with *events* that cause the *transitions* to be triggered. A *condition* is a boolean expression, and an *action* represents some variable assignments or events sending. When an *event* occurs, the guard condition is evaluated and the *transition* is fired only if the condition is true, performing the action.

Each *ACCESS* component begins with an *Idle* state, where it waits for a request (*req*). If a request is received and if the request is addressed to the component (*req.target==id*), the resource is accessed (*Access*) and the component replies (*sending*). When the request is not for the component (*req.target!=id*), the component forwards the request to other connected components through the network. In our case, when the environment *ENV* wants to access to the architecture, it sends a message to *GC* together with an indication about the target (either *RES1* or *RES2*), and the corresponding operation (*READ* or *WRITE*). *GC* receives requests from the environment(*ENV*). *GC* forwards the request to *LC1* or *LC2* through the *NETWORK*. *LC1* or *LC2* receives and processes the request from the *NETWORK*, and replies.



The architecture: a global controller(GC),
a network, and local controllers(LC1, LC2).

The behavior of GC, LC1 and LC2.

Fig. 2: Description of the application extracted from [14]

To guarantee integrity and confidentiality constraints regarding *LC1* and *LC2*, security mechanisms are applied. These security mechanisms must ensure: -*PRT1Init*, when a component sends a request that respects the access rights for accessing a resource, the access must be realized. -*PRT2Init*, any resource access must respect the access rights.

3.2 Domain Description

We suppose that a knowledge base has been built from previous experiences. The base contains a set of *problem cases* structured by security patterns. The author [14] defines a security pattern with a list of properties, its name, its functionalities and a description of the problem it is intended to solve, a static and a dynamic structure of the solution, formal properties and some examples of use.

The authorization ($AUTH$) pattern implements security measures for a resource (read, write, execution). $AUTH$ pattern ensures that a resource access by an entity *Ent*, for an operation *OpRes* is granted. When an access is authorized, the access is realized, otherwise counter-measures are triggered.

The structure of this problem case is depicted in figure 3. Function *hasRight (e: Ent, opRes: OpRes):Boolean* returns true if the entity *e* can perform the *opRes.oper* operation on the *opRes.res* resource (or said differently, if the resource is not protected for this operation or if the entity has an explicit permission).

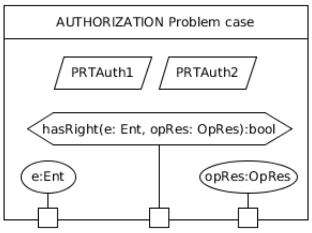

AUTHORIZATION Problem case
PRTAuth1
PRTAuth2
hasRight(e: Ent, opRes: OpRes):bool
e:Ent
opRes:OpRes

Fig. 3: Authorization Pattern Structure

In addition, the pattern is described with two formal properties. *PRTAuth1*, when an access request respects the access rights, the resource access is finally realized. *PRTAuth2*, a resource access must respect access rights.

$$PRTAuth1 : \forall c \in Auth, \forall e \in Ent, \forall opRes \in OpRes,$$
$$[evtVerify(c, AccReq(e, opRes)) \wedge right(c, e, opRes) \Rightarrow \Diamond evtAccess(c, e, opRes)]$$

$$PRTAuth2 : \forall c \in Auth, \forall e \in Ent, \forall opRes \in OpRes,$$
$$[evtAccess(c, e, opRes) \Rightarrow right(c, e, opRes)].$$

3.3 Problem Decomposition

Following our method, the problem is decomposed into smaller subproblems, so that a global solution emerges from a combination of smaller solutions. The structure of the two initial properties *PRT1Init* and *PRT2Init*, that have to be fulfilled, is similar to the structure of *PRTAuth1* and *PRTAuth2*. The $AUTH$ problem case is retrieved for reused.

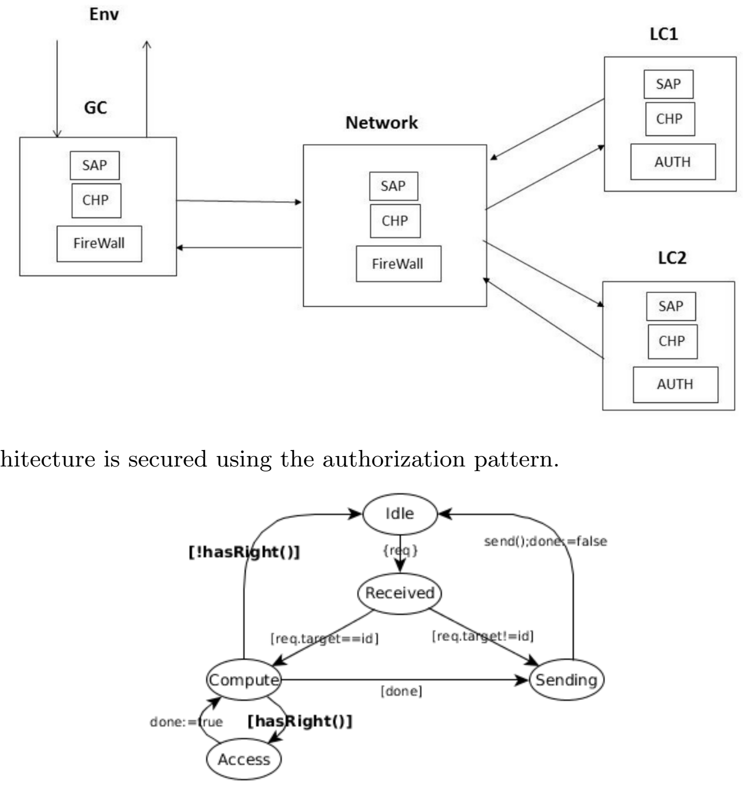
3.4 Solution Design

Then the solution is realized by combining the initial problem with the $AUTH$ problem case. The combination strategy can be a *composition* of self-contained and separated components, an *application* of patterns, or a *specialization* of a problem case and its solutions.

Because the $AUTH$ problem case is structured as a pattern, the most appropriate combination mechanism is the pattern application. It is realized in two steps, first, verification of the initial conditions, and second, the application of transformations. At the first step, some hypotheses have to be fulfilled. For instance one must identify an *Access* state in the design, where the authorization mechanism has to be introduced. In the case study, the following hypotheses must be respected: - *Hypothesis 1*, the reception of a message is carried out by reading the input fifo. This happen in a transition from the *Idle* state to the *Receive* state; *Hypothesis 2*, the sending of a message is carried out by a write into the output fifo, along a transition from the *Sending* state to the *Idle* state;

- *Hypothesis 3*, each transition to the *Compute* state has a source Receive state;
- *Hypothesis 4*, the *Access* state has only one source state, *Compute*.

If these hypotheses are respected, the second step produces a solution according to the pattern definition, as depicted by figure 4. The transformation rules are not presented in the article, but can be found in [14].



The architecture is secured using the authorization pattern.

This finite state machine represents the secured behavior of LC1 and LC2.

Fig. 4: Description of the secured application

3.5 Solution Verification and Diagnosis

Verification can be done using several techniques such as static analysis, theorem proving, or model checking. The later is a formal technique that, given a formal model of the system and a set of properties, explores all possible system states in a brute-force manner [1]. If abnormalities are detected in the design, counter examples are produced, i.e a trace from the initial state to an unexpected situation. Then diagnosis is triggered based on the observations of such traces. We use a model checker to check exhaustively the properties for this model. The verification validates a solution for the problem, but note that other solutions may also exists for this problem.

3.6 Iterating Through Problem and Solution Spaces

Assume that a new security policy is required for the same architecture, some counter-measures must be triggered in case of a security violation. According to the method, existing problem cases can be retrieved. The checkpoint pattern (*CHP*) allows to apply specific regulations, and defines properties that are closed to the new security constraints. The pattern is retrieved, and applied by taking into account the ($AUTH$) solution found previously, thus $AUTH$ is now a part of the problem.

The *CHP* problem case has been applied, and the solution verified. The resulting solution gathers two solutions, the checkpoint solution and the authorization solution. It can be retained in a knowledge base as a new reusable component named for instance *SECACCESS*.

We suppose that the architecture must evolves because a new local controller *LC3* is required. The new problem can rapidly be solved if one reuse the *SECACCESS* component.

4 System Design and Diagnosis Support

In this section we will present the conceptual model that sustains the approach applied on the case study, as well as tools we built [11]. The model has two parts, the former related to the reuse of verified designs 4.1, and the latter related to diagnosis management and support 4.2. Problem cases are the links between both parts.

4.1 Problem cases for System Design

Decomposing a complex problem into smaller problems that are more manageable and easier to solve, is a natural way to reduce the design complexity. When past experiences are available, this method can be improved by analogical reasoning, i.e. reusing past known problems. At the same time, it raises the issue of the way to capture different problems.

A problem is reified as a *problem case*. A conceptual model is provided as an illustration (figure 5). A *problem case* is either atomic or a combination of subproblem cases. An atomic *problem case* is made of *problem elements*. A *problem case* can be of different kinds, a *component*, a *pattern*, or a *solution*. Each *problem case* gathers a set of *problem elements* that depends on its kind. For example, *states*, *ports* and *functions* are all *problem elements*.

Some concepts are analogous to the problem frame approach [10], a *problem case* is similar to a *domain*, and *problem elements* are closed to *phenomena*.

To be reused, *problem cases* are combined together. Combination can be of different natures, for instance *composition*, *application* and *specification inheritance*. *Composition*, is the most easy way for reusing a problem case, as it only requires few adaptations. When the component is too generic, the counterpart is a lack of efficiency. Besides, the *application* of a pattern generally requires

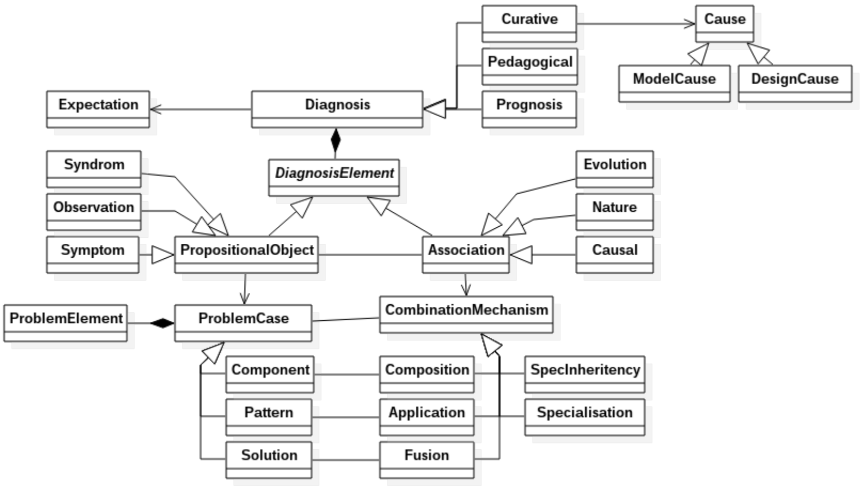


Fig. 5: Problem conceptual model

manual implementations. A pattern is a well-defined guidance for a recurring problem, but the solution must be adapted to the context. With *specification inheritance*, only specifications are reused, and the solution must be fully implemented. Thus, problem cases impact the amount of reuse.

4.2 Diagnosis Support

Many researchers [2, 5, 8, 4] divide the diagnosis in two main tasks: isolation (localization) and causal analysis. Isolation extracts the subset of elements, part of models, that needs to be corrected. Causal analysis associates causes to the observed abnormalities. These tasks are burden, particularly due to the huge amount of unrelated information the engineer needs to understand and correlate.The semantics gap is an example that is a discrepancy between the formalisms used during design and low-level traces obtained during verification. Reasoning on problems afford the advantage of raising the level of abstraction to a non technical level.

In [12], we proposed a formalization of diagnosis. Based on the formalization, a diagnosis can be captured together with *problem cases*, allowing design or model causes inference (figure 5). *Diagnosis* is made of (*diagnosis elements*), either *propositional objects* or *associations. Propositional objects* are descriptions about the problem parts (*states*, *traces*, *properties*, *problem cases* ...). Some descriptions are *symptoms* of errors (*counter-examples*). A *propositional object* is linked to other *propositional objects* by means of various natures of *associations* (causal, nature, evolution). A *diagnosis* may be *curative* if it aims to find the cause of an error. In our method, a *cause* can be a *model cause* or a *design cause.* A *model cause* happens when the selected problem is valid (for instance the $AUTH$ pattern is the good choice), but its implementation is not valid (for instance the $AUTH$ pattern is badly implemented). It locates the cause in the application of the pattern. Conversely, a *design cause* happens when the combination of the problem is valid (for instance the $AUTH$ pattern is correctly implemented), and the combined problem case is not appropriated, or incomplete.

5 Conclusion

Designing a solution for a given security problem, and diagnosing possible faults in the proposed solution, are tedious tasks. It is mainly due to poorly understood problem, and poorly managed information, that results in a lack of diagnosis support and solution reuse. Our hypothesis is that a method is required for analyzing the current problem, storing relevant information, and reusing known solutions as much as possible. This work paves the way to the elaboration of a verification management tool.

References

1. Baier, C., Katoen, J.P.: Principles of model checking. The MIT Press, Cambridge, Mass (2008)
2. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: ACM SIGPLAN Notices. vol. 38. ACM (2003)
3. Basili, V.R., Caldiera, G.: Improve software quality by reusing knowledge and experience. MIT Sloan Management Review 37(1), 55 (1995)
4. Clarke, E.M., Kurshan, R.P., Veith, H.: The localization reduction and counterexample-guided abstraction refinement. In: Time for verification. Springer (2010)
5. Cleve, H., Zeller, A.: Locating causes of program failures. p. 342. ACM Press (2005)
6. Fernandez, E.B., Larrondo-Petrie, M.M.: Designing Secure SCADA Systems Using Security Patterns. pp. 1–8. IEEE (2010)
7. Gamma, E. (ed.): Design patterns: elements of reusable object-oriented software. Addison-Wesley professional computing series, Addison-Wesley, Reading, Mass (1995)
8. Groce, A., Visser, W.: What went wrong: Explaining counterexamples. In: Model Checking Software, pp. 121–136. Springer (2003)
9. Hall, J., Jackson, M., Laney, R., Nuseibeh, B., Rapanotti, L.: Relating software requirements and architectures using problem frames. IEEE Comput. Soc (2002)
10. Jackson, M.: Problem frames: analysing and structuring software development problems. Addison-Wesley [u.a.], Harlow (2001), oCLC: 247895444
11. Leilde, V., Ribaud, V., Dhaussy, P.: An Organizing System to Perform and Enable Verification and Diagnosis Activities. In: IDEAL. pp. 576–587. Springer (2016)
12. Leilde, V., Ribaud, V., Teodorov, C., Dhaussy, P.: A diagnosis framework for critical systems verification. In: 15th International Conference on Software Engineering and Formal Methods, SEFM 2017. pp. Short–Papers. Springer (2017)
13. Leilde, V., Ribaud, V., Teodorov, C., Dhaussy, P.: Domain-oriented Verification Management. SUBMITTED TO : 8th International Conference on Model and Data Engineering (MEDI 2018) (Oct 2018)

14. Obeid, F.: Validation Formelle d Implantation de Patrons de Securite. Ph.D. thesis, ENSTA-Bretagne (2018)