Domain-oriented Verification Management

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. V. Basili stated twenty years ago that a software organization that manages quality should have a corporate infrastructure that links together and transcends the single projects by capitalizing on successes and learning from failures. For critical systems design, the verification tasks play a crucial role; when an unexpected situation is detected, the engineer analyzes the cause, performing a diagnosis activity. To improve the quality of the design, diagnosis information have to be managed through a well-defined method and with a suitable system. In this paper we present how a Verification Organizing System together with a problem-oriented method could achieve these issues. The key aspect of the approach is to follow a step-wise building of the solution, reusing known problems that are relevant for the system under study.

Keywords: Organizing System, Diagnosis, Problem Oriented Method

1 Introduction

D. Bjorner defines software engineering as a triptych: from descriptions of the *application domain* we construct prescriptions of the *requirements*; and from prescriptions of the requirements we *design* the software, i.e. construct specifications of software [7]. Our area of interest is software formal verification, especially with a model-checking approach. We start the model-checking process with a model of the system under consideration and a formal characterization of the property to be checked, i.e. two legs of the triptych : design and requirements. Then we run the model checker to check the validity of the property in the system model. When property is violated, the model checker provides us with a counterexample (a witness trace) that triggers a diagnosis activity to analyze the trace and to outline the error causes. Consequently, the system model is corrected and a new process cycle - verification, diagnosis, and correction - is repeated. The third leg is a domain-oriented verification defined as a process by which information used in verifying software systems is identified, captured, and organized with the purpose of making it reusable when modeling and verifying new systems.

Our research work is focused on methods and tools intended to ease the verification process, especially the diagnosis activities. Generally speaking, research addressing model-checking and diagnosis issues [2, 4, 6, 14, 19] are faced with the same difficulties. First, diagnosing the cause of abnormalities suffers from too detailed observations. It is hard, for instance, to localize the relevant parts in a detailed source-level trace when we look for the reasons a verification run failed [13]. Some techniques focus on linking low level information with more abstract information, like model-based diagnosis [29], or case-based reasoning [1]. Second, these techniques require a set of data that is not always available. In addition to the verification steps, the entire verification should be planned, administered, and organized. This is called verification organization by C. Baier and J.-P. Katoen [3]. During the engineering process, heterogeneous artifacts are produced, including requirements, system models, properties, runs, or diagnoses. As stated by T. C. Ruys [30], they are poorly managed and controlled. As a result, the expertise is poorly acquired by the verification engineers, and cannot be used for the above techniques. Third, 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 the single projects by capitalizing on successes and learning from failures [5]. 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.

Briefly stated, our approach aims to answer the issues above with a general diagnosis ontology [25], a Verification Organizing System [24], and a domain-oriented method, the latter being the subject of this paper. Some relevant parts of the ontology will be presented in section 3.1. The organizing system - an intentionally arranged collection of resources and the interactions they support [12] - makes easier the management of verification objects and supports reasoning interactions that facilitates diagnosis decisions; some features related to the method are drafted in section 5.1.

The method we propose in this paper relies on the idea of performing round trips between problem and solution spaces for improving the verification process. 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 background and related work. In section 3, we present the proposed method, its steps and a straightforward example. Section 4 shows the application of the method on the mutual exclusion problem. Section 5 discusses the knowledge base and its services, and section 6 concludes this study.

2 Background and Related Work

A. Newell and H. A. Simon introduced in [27] the problem-space hypothesis: *the fundamental organization unit of all human goal-oriented activity is the problem space.* M. Jackson introduced the concept of *problem frame* for presenting, classifying and understanding software development problems. Problem frames structure the analysis of the world in which the problem is located - the problem domain - and describe what is there and what effects one would like a system located therein to achieve [16]. A problem frame is defined in terms of its context and the characteristics of its domains, interfaces and requirements [20]. The problem frame approach allows engineers to build domain expertise and let practitioners gain experience from this knowledge base. POSE (Problem Oriented Software Engineering) [17] is an extension and generalization of problem frames. It is a representation and step-wise transformation of software problems to progress towards the solution. Software architecture [8], as well as development framework and design patterns [11] have same goals of knowledge construction, share and reuse. This kind of knowledge is generally attributed to the solution space. Compelling arguments justify an early understanding of stakeholders' requirements (focus on the problem). Equally compelling arguments justify an early construction of a suitable software-system architecture (focus on the solution).

Life-cycle model evolved from waterfall models to spiral models. Fine-grained spiral models are used by agile methods. The cornerstone of these processes is that developers craft a system's requirements and its architecture concurrently, and interleave their development [31]. Researchers from the Open University proposed an adaptation of the spiral life-cycle model, called the Twin Peaks Model to emphasize the equal status given to requirements and architecture [16]. The proposed model of software development is an iterative process during which problem structures and solution structures are detailed and enriched. In this context, the Open University team sees the use of architectural support as aiding the focus on the essential design requirements of the problem by allowing design concerns to be treated more abstractly and to be combined with behavioural requirements [16]. They extended problem frames towards this end.

In case of inadequate or unknown solution, problem-oriented approaches facilitate diagnosis. A well-known artificial intelligence approach, CBR (Case Based Reasoning), reduces the diagnosis effort by remembering a similar situation and by reusing information and knowledge of that situation. It proposes a method in four steps, retrieve, reuse, refine and retain. To some extent, the method we propose in this paper borrows the Twin Peaks idea of performing round trips between problem and solution spaces for leveraging diagnoses.

3 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 feeding diagnosis task with relevant information. This section describes formalization of problems, and the different steps of the flow.

3.1 Problem Formalization

According to V. Venkatasubramanian [32], Abnormal Event Management, a key component of supervisory control, involves the timely detection of an abnormal event, diagnosing its causal origins and then taking appropriate supervisory control decisions and actions to bring the process back to a normal, safe, operating state. Generally speaking, we have three main tasks; fault detection, diagnosis, and correction. Let see the tasks in a model-checking approach. Fault detection establishes that a system run raises an abnormal event: the exhaustive exploration encounters a state that violates the property under consideration, the model checker provides a counterexample, an execution path that leads from the initial system state to the violating state. Many researchers [4, 10, 13, 9] divide the diagnosis in two main tasks: isolation (localization) and causal analysis. Isolation extracts the subset of model parts that needs to be corrected. Causal analysis associates causes to the observed abnormalities. These are generally burden tasks, particularly due to a huge amount of unrelated information the engineer needs to understand and correlate. One example, known as the semantic gap, is the discrepancy between the formalisms used during design and low-level traces obtained during verification.

Reasoning on problem cases afford the advantage of raising the level of abstraction to a non technical level.

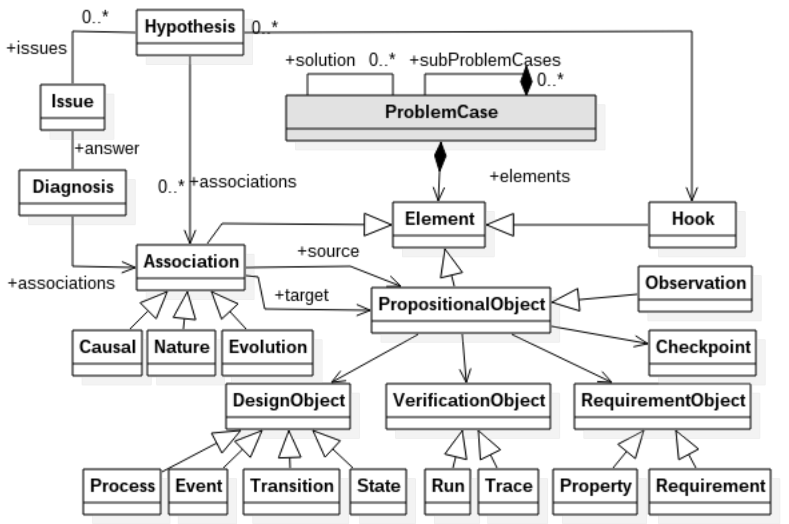

Fig. 1. Problem conceptual model

A conceptual model of the domain is given in figure 1. A problem is reified under a *problem case*. A *problem case* is composed of subproblem cases, that are made of *propositional objects*. *Propositional objects* are descriptions about the system components, such as *processes*, *states*, *runs*, *traces* or *properties*. Some *propositional objects* may be defined as *checkpoints*, key elements to be observed during the diagnosis activity. A *propositional object* is linked to other *propositional objects* by means of *associations* of different kinds: causal, nature, evolution. In the same way, *problem cases* are connected together with *associations*, through *hooks* exposing connection points. A set of *associations* between *problem cases* is an *hypothesis* of combination. This *hypothesis* may causes abnormalities, for instance when a problem case property is violated, and thus reveals an *issue*. The *issue* organizes results in a *diagnosis*. Finally *problem cases* may be reused as solutions for further designs.

We find analogous concepts in the problem frame approach, a *problem case* is similar to a *domain*, and *propositional objects* are closed to *phenomena*.

3.2 Illustration

To illustrate how the conceptual model is used, 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. A player gives either a right answer, that increases its score by one point, or a wrong answer, and no point is awarded. The match ends when a player reaches 3 points. The behavior of the solution is presented in figure 2.

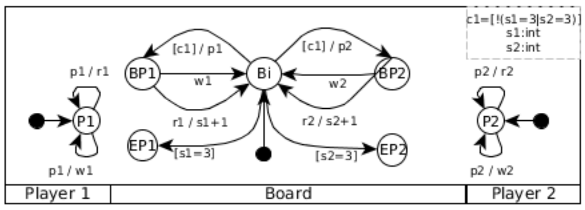

Fig. 2. Initial problem, first design

Transitions between states bear the Event-Condition-Action semantics represented as follow $Si \xrightarrow{\{Event\}[Condition]Action} Sj$. *Si* and *Sj* are *states*, arrows stand for *transitions*, labeled with *events* that causes the *transition* 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 taken only if the condition is true, performing the action.

The model is made of three *processes*, player one, player two and a board, and two variables, score one (s1) and score two (s2). Players share the same behavior. They wait for a question from the board (event p1 is a question from the board to player one, and event p2 is a question from the board to player two). Each player replies to the board. The response can be right (event r1 or r2) or wrong (event w1 or w2). When the board is in the idle state (Bi), it asks a question, either to player one or player two (respectively by sending events p1 or p2), if and only if none of the players have a score equals to 3 (c1 condition is false). If a question is asked to player one, the board goes in state BP1, and if a question is asked to player two, the board goes in state BP2. In these states, the board waits for a response, either a right response (event r1 or r2) and in that case the score is incremented, or a wrong response (event w1 or w2). When one of the players reaches a score of 3, the board goes in state EP1 or EP2, and the game is finished. Note that all the processes components (transitions, event or states) are represented by *observations*.

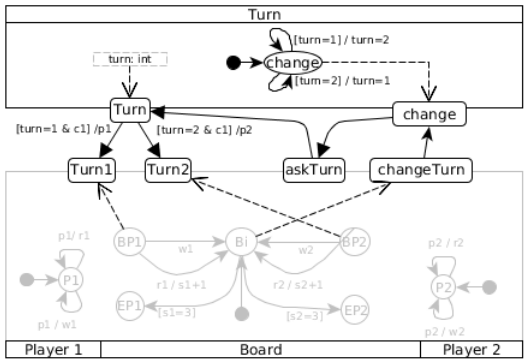

Fig. 3. Simple problem, second design

This model is not fair because in some case, the board may ask more questions to one player rather the other. To enable fairness, one can modify the design intuitively, or reuse a shared experience, represented as a *problem case*. Turn mechanism consists in memorizing the current entity that is authorized to do something. The authorized entity changes alternatively, thus, turn is a possible mechanism for fairness. As depicted in figure 3, a turn *problem case* contains a variable *turn*, representing the current turn memorized, and a mechanism to change the turn.

To combine turn with the current design, one relies on turn connection points called *hooks*. The turn *problem case* provides two *hooks*, one for changing the turn, and one for retrieving the turn variable. Combination can be achieved at the expense of updating the design and defining *hypotheses* of connection between the system design and turn *problem case*. Each connection between *hooks* are *association* with different semantics (*causal*, *nature*...).

The combination is a causal set of actions invoked from the *Bi* state. The turn change is invoked, then, a new turn value is retrieved, indicating the recipient for the next question. Since each question causes the change of the turn, fairness property is held.

This example presented how a *problem case* is defined, and how it can be combined with known problem cases to produce a new solution. In certain cases, when problem cases cannot be matched, combination through composition is impossible, and other mechanisms may be used. In the following section, we present the method and the various combinations mechanisms.

3.3 Method Steps

The step-wise method is presented by the activity diagram in figure 4. The method is reiterated until a satisfactory solution is achieved.

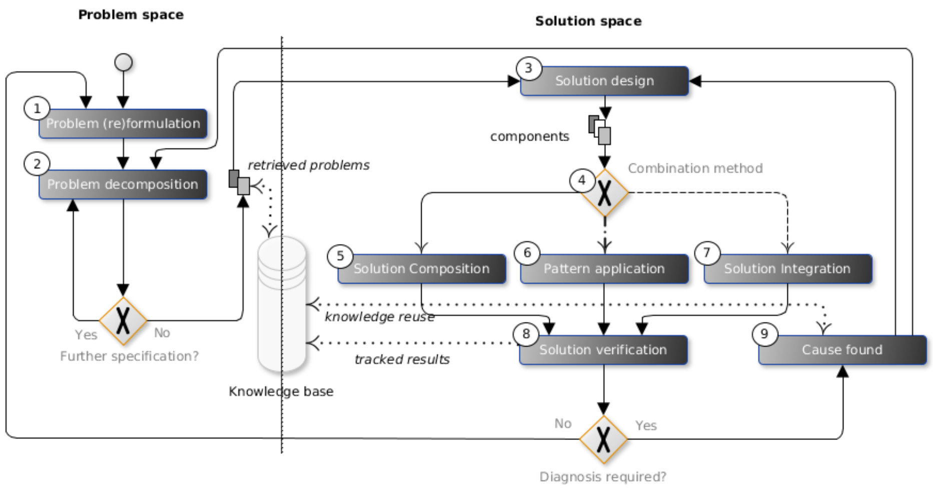

Fig. 4. Method steps

(1) The problem is formulated as a set of properties and constraints (architectural, technical choices), according to the conceptual model of figure 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 instance, "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, "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 integrated(7) [3]. (8) At this point, we built a part of the expected

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

solution; hence we are able to start a verification cycle. When abnormalities are observed, it triggers a diagnosis process. Verification results are tracked in the knowledge base. (9) Diagnosis reasoning process is performed, problem cases are used to enhance low-level observations, and the whole system provides the user with inference capabilities. The design is corrected, and the verification endeavor repeated. In some cases, the selected problem cases does 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 solution exist and where a classical design and verification activity has to be done.

The method performs roundtrips between two parts, the problem space, that consists in the problem elaboration, and the solution space, that consists in design and verifying the solution. While 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 proposal [16], a software iterative development process that focuses on the combination of problem structures and solutions structures.

The method applicability is illustrated in the next section onto a mutual exclusion problem design.

4 Method Application: Alice and Bob Share a Yard

We borrow our example from an invited talk given by Leslie Lamport [21] about two neighbors, Alice and Bob. Alice and Bob share a yard, but also have dogs, and naturally they want to let the dogs use the yard. The problem is that these dogs don't like each other, and they fight, so only one dog at a time can be in the yard. In the rest of the article we will talk only about Alice and Bob nor their pets. To demonstrate how our method can be applied, we build this example from some initial requirements, and from a minimal set of domain knowledge.

4.1 Domain Description

We suppose that a knowledge base as been defined from previous experiences. In particular, it contains the following problems extracted from [22][23]. For improving the readability, the i-th property is named Pi and each state (named Si) in a process (named PRi) is noted $PRi@Si$.

A concurrent system problem is composed of asynchronous processes, noted PRi. A basic property is that there be no deadlock; the set of processes must ensure the property noted $PDeadlock$.

In the mutual exclusion problem, each process of the collection, alternately executes a critical section noted $PRi@CS$ and a noncritical section noted $PRi@NCS$. Two processes cannot execute their critical sections concurrently. A process

structure is composed of the following states: - noncritical ($PRi@NCSi$); - trying ($PRi@Ti$); - critical ($PRi@CSi$); - exit ($PRi@Ei$). Both processes in figure 5 conform to the structure.
Mutual exclusion problem must ensure the following properties: (1) $PMutex$ defined : "For any pair of distinct processes PRi and PRj, no pair of operation executions $PRi@CSi$ and $PRi@CSj$ are concurrent". Equivalent in LTL as $\neg\Box PRi@CS \wedge PRj@CS$. (2) $PNolockout$ (starvation free): In every execution, if a process is in $PRi@Ti$, then later there is a configuration where the same process is in $PRi@CS$. Because the mutual exclusion is a concurrent system problem, it must ensure $PDeadlock$. If there is a deadlock, it means that "one or more processes are trying to enter $PRi@CS$, but no process ever does". There is also the possibility that a deadlock occurs because all the processes are stuck in their $PRi@Ti$ statements.

4.2 First Solution

Problem Formalization The initial requirement are "Alice and Bob pets can reach the yard", and "Alice and Bob pets must not be together in the yard". The requirements are formalized as $P1 : \Box(Alice@Trying \rightarrow \Diamond Alice@Yard)$, $P2 : \Box(Bob@Trying \rightarrow \Diamond Bob@Yard)$, $P3 : \Box\neg(Alice@Yard \wedge Bob@Yard)$. In the rest of the example, we omit P1 and P2.

Problem Decomposition Following the method, we decompose our problem and we look for known subproblems. The P3 formula structure is similar to the abstract formula of mutual exclusion, $PMutex : \Box\neg(PRi@CS \wedge PRj@CS)$, considering that *PRi* is *Alice*, *PRj* is *Bob* and *CS* is the *Yard*. We make the assumption that they address the same problem, and thus new properties emerge from this assumption, $PDeadlock$ and $PNoLockout$, described as $\neg\Box\Diamond(Alice@CS \vee Bob@CS)$.

Design and Pattern Application The design is illustrated in figure 5. There are two processes *Alice* and *Bob*, and a shared variable $Yard$. Both processes use the mutual exclusion structure, thus we apply this pattern.

Each process (Alice and Bob) conforms to the structure presented in section 4.1. The behavior is the following: *Alice* tries to access the yard ($Alice@Trying$); then Alice goes into the yard ($Alice@Yard$), the *Yard* corresponds to the *CS*; and finally exit the yard($Alice@Exit$). The same goes for Bob.

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 [3]. 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

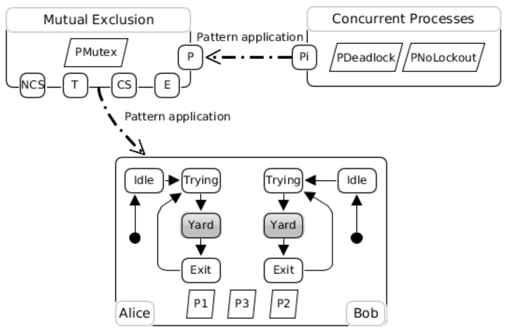

Fig. 5. Automata for Alice and Bob 1

observations of such traces. We use a model checker to check exhaustively the properties for this model.

PMutex is violated, indicating that Alice and Bob can be together in the yard at the same time. Since the structure of mutual exclusion problem is applied to Alice and Bob, each element of Alice and Bob can be understood from the point of view of mutual exclusion problem.

4.3 Second Solution - Turns

Problem Decomposition At this point, we need a mechanism to ensure the access to the critical section. Browsing the knowledge base, we can choose to pick the "turn" problem case. From the specification point of view, turn is defined by a (*turn*) variable, and two properties, *PChange*: after a process has finished its execution *turn* must be changed; *PTurn*: a process cannot be in execution if it is not its turn.

Design and Solution Integration The turn case is used to alternate the yard access. The design is presented with the automata in figure 6. The turn mechanism has to be combined with *Alice* and *Bob* structure ($NCS \rightarrow Tr \rightarrow CS \rightarrow Ex$). We suppose that the combination is complex enough to require an ad hoc integration. The process that had the last access is stored, using the *turn* integer variable. A process checks the value of the *turn* variable in *PRi@Trying* statement. If the value is equal to its personal turn, it is authorized to access the yard. Then, *PRi@Exit* statement sets the *turn* variable to the other process.

Verification and Diagnosis A new run is performed, the initial requirement *PMutex* is verified, but *PNoLockout* is violated.

Integration implies that the concepts of an integrated problem are widespread into the solution. Thus, parts of the problem are difficult to observe. According to the description of *PNoLockout*, "In every execution, if some processor is in

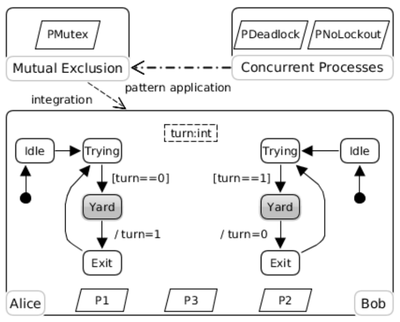

Fig. 6. Automata for Alice and Bob using Turn

the entry section in a configuration, then there is a later configuration in which that same processor is in the critical section". By analogy *Bob* is continuously in *Bob*@*Trying* state, and cannot pursue in the yard. Thus *Bob* has not the *turn* anymore, because *Alice* is never in *Alice*@*Trying*.

4.4 Problem Reformulation

Turn approach is formalized and stored in the knowledge base as follows. Turn is defined by a (*turn*) variable, followed by the properties: *PChange*: after *PRi*@*Exit*, *turn* must be equals to 1 and After *PRj*@*Exit*, *turn* must be equals to 0; *PTurn*: *PRi* cannot be in the *PRi*@*CS* if *turn* is not equals to 0 and *PRj* cannot be in the *PRj*@*CS* if *turn* is not equals to 1.

4.5 Using Flags

Problem Decomposition The turn problem case is interesting if we not consider the *PNoLockout* property. Another idea consists in sharing the intention of Alice and Bob to access the yard. This intention can be captured using two flags, one for Alice and one for Bob. A raised flag means that the person wants to go in the yard, and reciprocally, a lowered flag means the person doesn't want to.

Design of a new Solution The new solution is based on an array named *flag* of two booleans. The first boolean indicates if *Alice* want to access to the *Alice*@*Yard* or not. The second boolean indicates if *Bob* want to access to the *Bob*@*Yard* or not. *Alice* can access to the *Yard* if and only if *Bob* hasn't raised his *flag*. The same goes for *Bob*. When *Alice* or *Bob* are in the yard, he/she raises the *flag*. Finally the *flag* is lowered in the *Exit* state. The design is illustrated in figure 7.

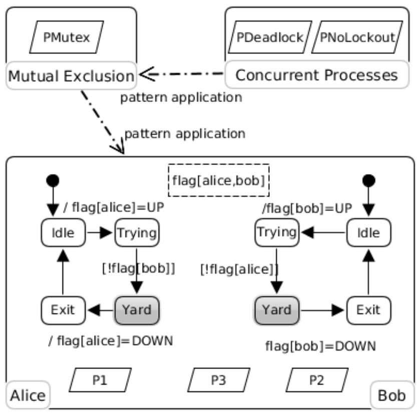

Fig. 7. Automata for Alice and Bob using Flag

Verification and Diagnosis A new run is performed, and as a result, a deadlock occurs.

Suppose that *Alice* and *Bob* are interrupted in their $Trying$ section. At this point each have claimed for entering in the yard but is not yet sure if the *Yard* is in use. Then each of them sees the $flag$ of the other one and wait. Each is waiting indefinitely for the other, a deadlock has occurred.

4.6 Flag Problem Formulation

We formulate the new approach using $flags$. It contains an array of two boolean called $flags$, and a set of following properties: (1) $PRaise$, PRi raises its $flag$ in $PRi@CS$ state, and PRj raises its $flag$ in $PRj@CS$ state; (2) $PLower$, PRi lower its $flag$ in $PRi@Exit$ state and PRj lower its $flag$ in $PRj@Exit$ state; (3) $PWait$, PRi cannot access $PRi@CS$ if $flag$ of PRj is raised, and PRj cannot access $PRj@CS$ if $flag$ of PRi is true.

4.7 Taking Turns and Raising Flags

Problem Formulation Now, we decide to design a solution that solves all the problems mentioned above. We know that the property $PMutex$ is fulfilled either with a turn or a flag. But the first solution violates *PNolockout* property, and the second violates *PDeadlock* property. We try to combine these problem cases together to fulfill all properties.

Design and Solution Integration Flag problem supposes that a flag raised by a process indicates its intention to enter in the critical section. Turn problem supposes a priority to enter in the critical section. We combine these two mechanisms; entering to the critical section is granted for PRi if PRj does not want to enter the critical section, or if PRj has given up priority to PRi by setting turn to PRi. The design is given in figure 8. For readability sake, only the automaton of *Alice* is depicted.

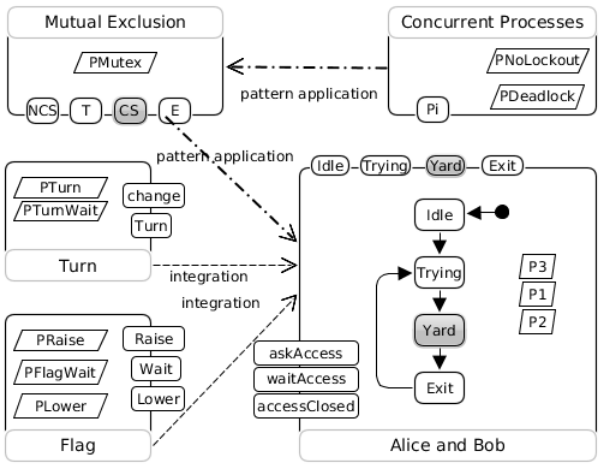

Fig. 8. Automata for Alice and Bob using Flags and Turns

Verification and Problem Reformulation It turns out that the whole set of properties are verified, thus the solution is acceptable. Finally, we formulate our problem of Alice and Bob sharing a yard, as a combination of Concurrent Process, Mutual Exclusion, Turn and Flag problem cases.

5 Tool Support

This approach applies when problems and known solutions are available. It supposes to combine the method with appropriated tooling for creating, storing, querying and retrieving problem cases.

5.1 A Verification Organizing System

In a previous article [24], we presented the Verification Organizing System (VOS) "an Organizing System is an intentionally arranged collection of resources and the interactions they support [12]." The VOS is a three-layered infrastructure made of a storage tier, a knowledge tier, and an access tier.

The storage tier is characterized by a variety of sources, heterogeneous with respect to several dimensions concerning form and content properties. It is based on a Software Configuration Management (SCM) system that controls versioned artifacts produced. It includes no exhaustively verification endeavors (run, traces), properties or models, andproblem cases.

The knowledge tier, a logic-based, knowledge-rich level, plays the central role of a shared language to connect people to people, people to information, and information to information, represented as an ontology. It allows for knowledge creation, query and inference.

The access tier is used for diagnosis tools interoperability. Heterogeneous tools can interoperate by the underlying mechanism of model federation [15].

These tools can be classified in three categories, model-based, process-history-based and interaction-based. Model-based tools assumes that a model of the system is available [29] allowing to localize the subset of system's constituents generating abnormalities. Process history-based tools relies on the availability of large amount of historical process data, and thus, can be used for extracting knowledge [26], or reasoning [1]. Interaction-based tools allow for observing, controlling, understanding and altering the system execution. Examples includes omniscient debuggers [28], or visualization tools [18].

5.2 Knowledge and Inference

Figure 9 illustrates the various artifacts produced at each step of the method. Problems (1) are progressively decomposed into other problem cases (2). Given the problem case structure, the organizing system can be used for querying and retrieving problems cases that are the most relevant. This may be achieved, for instance, according to three tasks, search, initially match, and select [1]. Combining problem cases (4) is achieved with a certain level of automation. Composition (5) is the most automated way. Problem cases are on the shelf solutions connected through well defined connection points. For pattern application (6), the degree of freedom lies in the rules of application between the problem pattern and the given solution. Integration (7) is a completely ad-hoc combination. (8) It shows how the previous combination techniques may affect the diagnosis task. Errors are represented with black stars. Composition mainly results in connections errors, pattern application mainly generates errors in the rules of pattern application, and integration may produce widespread errors. Finally, a solution is kept in the organizing system. It involves the selection of relevant information from the new problem case to keep.

6 Conclusion

Designing a solution for a given 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. When a new solution is designed, for which abnormalities are observed, problem cases enhance the initial observations with more intelligible information. This work paves the way to the elaboration of problem-centered diagnosis tools, proposing adapted views and relevant checkpoints fostering the diagnosis activity.

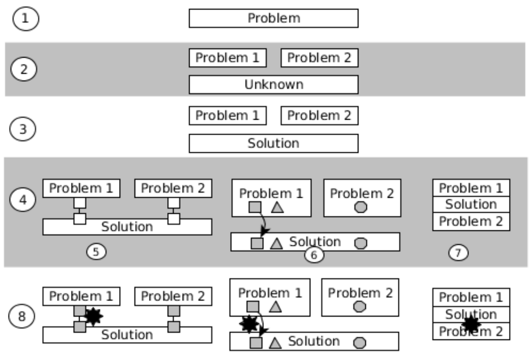

Fig. 9. Artifacts produced

References

1. Agnar, A., Enric, P.: Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. AI Communications (1), 39–59 (1994)
2. Alrajeh, D., Kramer, J., Russo, A., Uchitel, S.: Automated support for diagnosis and repair. Communications of the ACM 58(2), 65–72 (2015)
3. Baier, C., Katoen, J.P.: Principles of model checking. The MIT Press, Cambridge, Mass (2008)
4. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: ACM SIGPLAN Notices. vol. 38, pp. 97–105. ACM (2003)
5. Basili, V.R., Caldiera, G.: Improve software quality by reusing knowledge and experience. MIT Sloan Management Review 37(1), 55 (1995)
6. Bertoli, P., Bozzano, M., Cimatti, A.: A symbolic model checking framework for safety analysis, diagnosis, and synthesis. In: International Workshop on Model Checking and Artificial Intelligence. pp. 1–18. Springer (2006)
7. Bjrner, D.: Software Engineering 3. Texts in Theoretical Computer Science An EATC Series, Springer-Verlag, Berlin/Heidelberg (2006)
8. Buschmann, F. (ed.): Pattern-oriented software architecture: a system of patterns. Wiley, Chichester ; New York (1996)
9. Clarke, E.M., Kurshan, R.P., Veith, H.: The localization reduction and counterexample-guided abstraction refinement. In: Time for verification, pp. 61–71. Springer (2010)
10. Cleve, H., Zeller, A.: Locating causes of program failures. p. 342. ACM Press (2005)
11. Gamma, E. (ed.): Design patterns: elements of reusable object-oriented software. Addison-Wesley professional computing series, Addison-Wesley, Reading, Mass (1995)
12. Glushko, R.J.: . Foundations for Organizing Systems. The Discipline of Organizing, edited by Robert J Glushko (2012)
13. Groce, A., Visser, W.: What went wrong: Explaining counterexamples. In: Model Checking Software, pp. 121–136. Springer (2003)
14. Gromov, M., Willemse, T.A.: Testing and model-checking techniques for diagnosis. In: Testing of Software and Communicating Systems, pp. 138–154. Springer (2007)

15. Guychard, C., Guerin, S., Koudri, A., Beugnard, A., Dagnat, F.: Conceptual interoperability through Models Federation. In: Semantic Information Federation Community Workshop (2013)
16. Hall, J., Jackson, M., Laney, R., Nuseibeh, B., Rapanotti, L.: Relating software requirements and architectures using problem frames. pp. 137–144. IEEE Comput. Soc (2002)
17. Hall, J.G., Rapanotti, L., Jackson, M.: Problem Oriented Software Engineering: A design-theoretic framework for software engineering. pp. 15–24. IEEE (Sep 2007)
18. Hamou-Lhadj, A., Lethbridge, T.C.: A survey of trace exploration tools and techniques. In: Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research. pp. 42–55. IBM Press (2004)
19. Holzmann, G.J.: The Theory and Practice of A Formal Method: NewCoRe. In: IFIP Congress (1). pp. 35–44 (1994)
20. Jackson, M.: Problem frames: analysing and structuring software development problems. Addison-Wesley [u.a.], Harlow (2001), oCLC: 247895444
21. Lamport, L.: Solved problems, unsolved problems and non-problems in concurrency. ACM SIGOPS Operating Systems Review 19(4), 34–44 (Oct 1985)
22. Lamport, L.: The mutual exclusion problem: part Ia theory of interprocess communication. Journal of the ACM (JACM) 33(2), 313–326 (1986)
23. Lamport, L.: The mutual exclusion problem: partIIstatement and solutions. Journal of the ACM (JACM) 33(2), 327–348 (1986)
24. Leilde, V., Ribaud, V., Dhaussy, P.: An Organizing System to Perform and Enable Verification and Diagnosis Activities. In: International Conference on Intelligent Data Engineering and Automated Learning. pp. 576–587. Springer (2016)
25. 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)
26. Liu, Y., Xu, C., Cheung, S.: AFChecker: Effective model checking for context-aware adaptive applications. Journal of Systems and Software 86(3), 854–867 (Mar 2013)
27. Newell, A., Simon, H.A., et al.: Human problem solving, vol. 104. Prentice-Hall Englewood Cliffs, NJ (1972)
28. Pothier, G., Tanter, ., Piquer, J.: Scalable omniscient debugging. ACM SIGPLAN Notices 42(10), 535–552 (2007)
29. Reiter, R.: A theory of diagnosis from first principles. Artificial intelligence 32(1), 57–95 (1987)
30. Ruys, T.C., Brinksma, E.: Managing the verification trajectory. International Journal on Software Tools for Technology Transfer (STTT) 4(2), 246–259 (Feb 2003)
31. Swartout, W., Balzer, R.: On the inevitable intertwining of specification and implementation. Commun. ACM 25(7), 438–440 (Jul 1982)
32. Venkatasubramanian, V., Rengaswamy, R., Kavuri, S.N.: A review of process fault detection and diagnosis. Computers & Chemical Engineering 27(3), 313–326 (Mar 2003)