Model-based Diagnosis Patterns for Model Checking

Vincent Leilde, Vincent Ribaud, Philippe Dhaussy

Model-based Diagnosis Patterns for Model Checking

Vincent LEILDE[1], Vincent RIBAUD[2], Philippe DHAUSSY[1]

[1] Lab-STICC, team MOCS, ENSTA Bretagne,rue François Verny,Brest
e-mail: firstname.lastname@ensta-bretagne.fr
[2] Lab-STICC, team MOCS, UBO,avenue Le Gorgeu,Brest
e-mail: ribaud@univ-brest.fr

1 Context and challenges

Model checking is a technique used to verify that a certain system's design satisfies its requirements. Given some models of the design and system's requirements formulated as formal properties, the system model can be checked [1], and if properties are violated, the model checker provides the user with counter examples that represent execution sequences (traces) leading to an unexpected situation. Then the engineer analyzes the cause of the problem, i.e. diagnosis activity, correct models or properties and carry out another verification endeavors. A verification process may include many verification endeavors gathering models and properties successively refined, which might be recorded in a dedicated form; stated deservedly by Ruys [2] to be a challenge. Diagnosing the cause of faulty properties is also a challenging task. Under the assumption formal properties are valid, and without neglecting the difficult problem to judge whether the formalized problem statement (model, properties) is an adequate description of the actual verification problem [3], we reduce here the scope to modeling errors. Model-based diagnosis (MBD) is a promising approach to diagnose modeling errors and consists in the interaction of observation and prediction [4] where observation indicates what the device is actually doing, and prediction indicates what it is supposed to do. “The interesting event is any difference between these two, a difference termed a discrepancy [5].” MBD presumes that “if the model is correct, all the discrepancies between observation and prediction arise from defects of the device [5].” Thereby diagnosis consists in identifying the faulty components responsible of the observed failure. When we apply this approach to model checking, the design is the system-under-study, and we need a correct model of the design to apply model-based reasoning. The diagnostician can be assisted by methods like Case-Based Reasoning (CBR) to dispose of a correct model. CBR consists in “solving a new problem by remembering a similar situation and by reusing information and knowledge of that situation [6]." Unfortunately these diagnostic methods/techniques are only possible if significant features about cases are identified and formalized. In conclusion, dealing with multiple data or diagnosing faults are challenges which require the verification's information to be well-defined and managed through time; to this intent we propose to define patterns facilitating information's formalization and sharing among engineers. We illustrate the idea through a verification scenario on a sample case, with a focus on diagnosis artifacts.

2 Diagnoses using patterns

2.1 Using patterns in model checking

The seminal book on design patterns defines design patterns as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" [7]. Issued from C. Alexander's work [8], design patterns describe object-oriented designs but one could be easily generalized in the context of model checking, as illustrated by Dwyer et al. [9] who provides form of pattern to specify formal properties. Verification process tools heavily depend on the structure and content of their case for searching, versioning, diagnosing, or adapting models, their description share goals of patterns as they must address a consensus about a topic, shared by a community. Relationships between cases such as uses or extends have also to be expressed in the patterns. In the model-checking community, benchmarks, i.e. collections including models, are used to assess tools performances. Benchmark repositories organize models, properties, verification runs and results. We used the structure of the BEEM benchmark [10] as a basis for describing cases. In the rest of the article the names of potential pattern's constituents are formatted in *italics*.

2.2 Bob and Alice share a yard

Lets us take a case borrowed from Lamport [11], with a viewpoint related to model checking. Alice and Bob are sharing a yard in an exclusive manner because their pets cannot be together in the yard. Lamport's solution uses two threads sharing only two Boolean variables (flags), each of which can be written by one thread and read by the other. For verification purposes, the system based on the mutual exclusion algorithm should be translated into a *model* describing how the system behaves, with clearly defined components and functions (required to a good diagnosis, i.e. finding the component/function source of the fault). Lamport's algorithm and a corresponding model in a concurrent automata-like language are given in Fig.1.

```
Alice:
   while (true) {
   flagAlice = up;
      while (flagBob == up)skip;
   catInYard;
   flagAlice = down;
   }
Bob:
   while (true) {
   flagBob = up;
      while (flagAlice == up) {
      flagBob = down;
         while (flagAlice == up)skip;
      flagBob = up;
      }
   dogInYard;
   flagBob = down;
   }
```

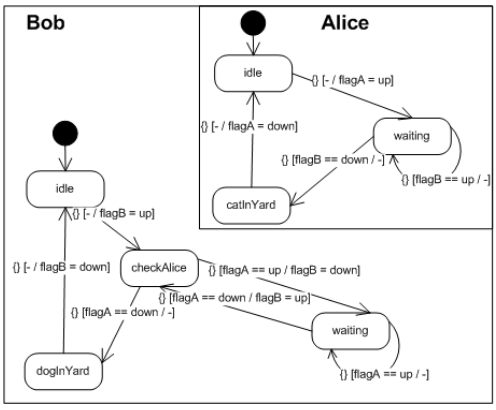

Fig. 1. Lamport's mutual exclusion algorithm

To prove the model is correct, a verification by model checking is conducted according *properties*: - Mutual Exclusion (P1), Bob and Alice must not be together in critical section (CS); - Deadlock Freedom (P2), it never happens that Alice or Bob are trying to enter their CS, but never succeeds; - Lockout Freedom (P3), from a fair path Alice or Bob try to access the CS, it will terminate; - Fairness (P4), it is always possible for Alice or Bob to access the CS. We reach a valid model through several *verification endeavors* until all properties are satisfied. A verification endeavor is composed of different versions of behavioral models coupled with properties and *run results*. *Models* are expressed with different *implementations* (such as in Fig.2) and *properties* are checked on these implementation during *verification runs*.

```
bool flagAlice,flagBob;
#define csa alice@CS; #define csb bob@CS; #define mtx !(csa && csb)
#define ecsa alice@E_CS; #define ecsb bob@E_CS;
#define lcsa alice@L_CS; #define lcsb bob@L_CS;
```

```
active proctype alice(){
E_CS: printf("Entering CS\n");
do::flagAlice=true;
   do::!flagBob-> break
   ::else->skip;
   od;
CS:  printf("CS\n");
L_CS:
  atomic{
      flagAlice=false;
     printf("Leaving CS \n");
   }
od;
}
```

```
active proctype bob(){
E_CS: printf("Entering CS\n");
do::flagBob=true;
   do:: flagAlice->flagBob=false;
      do::!flagAlice
        ->atomic{flagBob=true;break;}
      ::else->skip;
      od
   :: !flagAlice->break
   od
CS:  printf("CS\n");
L_CS: atomic{
      flagBob=false; printf("Leaving CS\n");}
od;}
```

Fig. 2. A Promela implementation of the concurrent automata of Fig. 1.

Verification runs are performed in a *model checker* with a *configuration*, give results about the status of properties (see Fig. 3), and *traces* are produced raising diagnosis, corrections and new verification cycles. The SPIN model-checker is used here [1].

Property	**Implementation**	**Run result**
P1	never {// ![]mtx T0_init: do:: atomic { (! ((mtx))) -> assert(!(! ((mtx)))) } :: (1) -> goto T0_init od; accept_all: skip}	**errors: 0**
P2	Automatically checked by SPIN	**errors: 0**
P3 (for Alice, the results are the same for Bob)	never {// !([]<>csa->[](ecsa-><>csa)) T0_init: do:: (! ((csa)) && (ecsa)) -> goto T0_S69 :: (1) -> goto T0_init od; T0_S69: do:: (! ((csa))) -> goto T0_S69 od;}	**errors: 0**
P4 (for bob, no errors found for Alice)	never { // !([]<>csb) T0_init: do:: (! ((csb))) -> goto accept_S4 :: (1) -> goto T0_init od; accept_S4: do:: (! ((csb))) -> goto accept_S4 od;}	Acceptance cycle **errors: 1**

Fig. 3. Properties implementation and verification in SPIN

Each property is expressed as a never claim and checked successively by SPIN, almost all properties are verified (errors:0), excepted P4 for Bob, the fairness property which is falsified meaning that it doesn't give to Bob a “fair” chance to try to enter its CS, and thus we are faced with an unexpected situation, and one have to find the origin of the fault. Following a MBD approach, a model that predicts how the system behaves can be used to find the component or function that causes the problem. Even though in model checking this model is not available, CBR approach can luckily be applied to retrieve a similar case. A CBR system gathers previous cases (that we call Problem Cases), as a basis to reason on a new case (called Sample Case), and helps to isolate the faulty element that leads to the discrepancies between observation (Sample case) and prediction (Problem case). CBR process is conducted in four steps [5], retrieve the most similar cases from the case base, reuse these cases to solve the new problem, revise the proposed solution, and retain the new problem and solution in the case base for later use.

2.3 **Reuse the problem case and solution**.

Once the sample case has been assessed, the most similar problem case is searched in the case base. Our case addresses the classical *problem case* of Mutual Exclusion as it supports almost the same properties. A problem case is a sample case that has been reified in the case base. It provides a specification in a form of a *structure* and a set of *abstract properties* that prescribes the system's behavior. A specification addresses the problem space (not the solution space) and is by definition independent from any implementation language, which is rather given by solution implementations. A problem case provides a structure, for Mutual Exclusion we state that each process’s program may be written as follows (see Fig. 4): The entry statement represents all the operation executions between a noncritical section (NCS) and the subsequent critical section (CS); The exit statement represents all the operation between a CS and the subsequent NCS; The initial declaration describes the initial values of the variables. Abstract properties might be expressed in temporal logic like LTL (see Fig.5) which is based upon the propositional calculus, where formulas are composed of atomic propositions and operators [1], by contrast to assertions that impose control point in the system and cannot express all properties (deadlock, starvation). *Abstract properties* are inspired by Lamport's work: -Mutual Exclusion (P1), For any pair of distinct processes i and j, no pair of operation executions CSi and CSj are concurrent i.e. $\Box\neg(CSi \wedge CSj)$; -Deadlock Freedom (P2), deadlock appends if it is impossible to reach a state in which some processes are trying to enter their CS, but no process is successful; -Lockout Freedom (P3), if a process in a fair path tries to execute its CS then eventually it succeed, i.e. for i $\Box\Diamond CSi \rightarrow \Box(E\_CSi \rightarrow \Diamond CSi)$. -Fairness (P4), it should be possible for each process to access the CS , i.e. $(\Box\Diamond CSi) \wedge (\Box\Diamond CSj)$.

```
initial declaration;
repeat forever
     noncritical section;
     entry ;
     critical section;
     exit ;
end repeat
```

Fig. 4. Mutual Exclusion Structure

Operator	not	and	implies	always	eventually
LTL	¬	∧	→	□	◊
SPIN	!	&&	->	[]	<>

Fig. 5. Correspondences between notations

The structure and properties of problem cases have to be bound to the sample case including the difficult task to identify and bind parts of the model to equivalent parts of the problem case structure. Parts of the implementation model might be annotated (E_CS, CS, L_CS) in order to apply the properties given the problem structure. The problem case is used to predict the expected behavior, and thus fairness by comparing the observed model of Alice and Bob and deducted which component is not fair. To this intent, a problem case should provide a set of potential *causes* and *solutions* to a failed property, for instance fairness property P4 is violated by Bob, it results the cause "Bob is too kind with Alice", and a solution have to be found to this cause that will update processes accordingly.

Thus a problem case may propose a set of *solutions,* for mutual exclusion many solutions exist with different properties, in our example a possible solution might be the Peterson's algorithm [12] given in Fig.6. Then the engineer applies the adequate solution to its wrong model, to do that he/she must understand precisely how to bind the solution to his sample case either by copying or adaptation [6] techniques. Copying is the trivial type of reuse where the solution is directly transferred to the case as its solution while adaptation is relevant if differences are taken into account, it consists in either reuse the past case solution (transformational reuse), or reuse the past method that constructed the solution (derivational reuse). When a solution is applied to the sample case, a new verification must be performed. If the properties are still not satisfied, the diagnostic of the cause become clearer, and revised solutions can be brought again. Finally the new solution is retained in the case base, together with sample case reified as a problem case. The most relevant features of the new problem case have to be revealed to facilitate further identification.

```
bool flag[2]; flag[0] = false; flag[1] = false; int turn;
```

```
P0: flag[0] = true; turn = 1;
    while (flag[1] && turn == 1){
        // entry
    }
    // critical section
    // exit
    flag[0] = false;
```

```
P1: flag[1] = true; turn = 0;
    while (flag[0] && turn == 0){
        // entry
    }
    // critical section
    // exit
    flag[1] = false;
```

Fig. 6. Peterson's algorithm

3 Conclusion

Model checking relies on a large collection of heterogeneous artifacts and diagnosing faults is generally a tedious task which should benefit from a knowledge base system to collect and provide access to well-formalized verification artifacts. In the formal method community, patterns are mostly understood in reference to the work by

Dwyer et al. [9], and actually there is no common agreement on a formalization or organization for the whole verification process artefacts, such as models, traces or sessions of verifications. Hence this paper asks the question about unmet needs for such patterns and their possible relationships, with a particular focus on patterns required for diagnosis techniques such as problem case or solution in CBR.

4 References

1. Ben-Ari, M. (2008) Principles of the Spin Model Checker. London: Springer.
2. Ruys, T. C., & Brinksma, E. (2003). Managing the verification trajectory. International journal on software tools for technology transfer, 4(2), 246-259.
3. Baier, C., & Katoen, J. P. (2008). Principles of model checking (Vol. 26202649, pp. 19-82). Cambridge: MIT press.
4. Davis, R. (1984). Diagnostic reasoning based on structure and behavior. Artificial intelligence, 24(1), 347-410.
5. Davis, R., & Hamscher, W. (1988). Model-based reasoning: Troubleshooting. In Exploring Artificial Intelligence: Survey Talks from the National Conferences in Artificial Intelligence (pp. 297-346). Morgan-Kaufmann.
6. Agnar Aamodt and Enric Plaza (1994). Case-based reasoning: foundational issues, methodological variations, and system approaches. *AI Commun.* 7, 1 (March 1994), 39-59.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
8. Alexander, Christopher, Sara Ishikawa, and Murray Silverstein (1977). A Pattern Language: Towns, Buildings, Construction. New York: Oxford University Press.
9. Dwyer, Matthew B., George S. Avrunin, and James C. Corbett (1999). Patterns in property specifications for finite-state verification. In Proceedings of the 1999 International Conference on Software Engineering. IEE.
10. Pelánek, R. (2007). BEEM: Benchmarks for explicit model checkers. In Model Checking Software (pp. 263-267). Springer Berlin Heidelberg.
11. Lamport, L. (1985). Solved problems, unsolved problems and non-problems in concurrency. ACM SIGOPS Operating Systems Review, 19(4), 34-44.
12. Peterson, G. L. (1981). Myths about the mutual exclusion problem. Information Processing Letters, 12(3), 115-116.