Software System Analysis with Graph Homomorphisms

C. V. KARAPOULIOS, G. K. ADAM, S. N. XANTHAKIS
Computer Science and Communications Department
Technological Institute of Larissa
Larissa Greece

Abstract: - In this paper we present a graph based formalism for the analysis and the behavioral envisioning of software based systems. We first try to identify and solve the main limitations of dynamic analysis approaches. Follow some results of a qualitative abstraction tool and some simple application examples. We then introduce the concept of a system qualitative graph, the role of graph homomorphisms for modeling a software, and more generally a system, as a continuous phase transition map operating on an abstract data space. We finish with the presentation of the underlying mathematical framework and some properties of graph homomorphism invariants.

Key-Words: - software engineering, qualitative reasoning, software testing, dynamic system analysis, graph homomorphisms, software phase spaces.

1 Introduction

The software engineering community has developed several methods, tools and concepts for dealing with the increasing complexity and size of software based systems. System reliability, maintainability and portability, constitute the main challenges for this engineering field. Artificial Intelligence (AI) concepts and ideas have been successfully applied in this field: automatic programming from examples, conceptual models for software design, automatic program understanding of programmer's intent, case based reasoning for software maintenance and reusability, etc [1], [2]. Qualitative reasoning (QR) [3] has proved to be an AI field with many successful contributions to system analysis, (physical and artificial systems). However, with some exceptions in model based reasoning [4], [5], [6] QR has not been widely applied to software and system engineering.

The difficulty of applying QR concepts to software engineering is due to the fact that the algorithmic behavior cannot be described as a physical system governed by a simple set of differential equations and some system parameters. Data types are heterogeneous and are not always ordinal: strings, records, lattices, vectors, trees, etc. In the same time, software tends nowadays to be integrated with hardware (hybrid systems). A more systemic approach is needed for addressing software based systems.

The most widespread software system analysis models are essentially based on formal static methods [7]. They have their own advantages and limitations [8] due to the inherent complexity of the software programming process. However those approaches cannot be considered as qualitative since they are too analytic (even if a certain level of data abstraction is operated) and do not propose a formalism that envisions software behavior as a whole. A software (or more generally a system) formalism (formalism) must be qualitative and, in our understanding, must respect the following specifications:

- It must propose a right level of a *dynamic behavioral* abstraction applicable to a wide range of *hybrid* systems,
- It must be able to express data type *heterogeneity* and system *compositionality* (outputs of a software module can be used by another module),
- For ordinal inputs, when present, this formalism must be able to envision software-system behavior when those inputs *change*,
- This formalism must contain the concept of *continuity* (even for non ordinal inputs) that is pervasive to any QR reasoning domain.

The paper is organized as follows. We first expose a motivating example of a simple piece of software source code and its corresponding software qualitative graph. This graph summarizes the global behavior of the software system in response of its inputs continuous change. The construction of such graphs is completely automated by a tool and can be easily generalized for any system. However, we shall concentrate our analysis here to software based systems. Qualitative graphs must respect some constraints that are independent of the internal structure of the system they envision. Those constraints are uniquely and strictly related to the metric properties of the input space and not only to its dimension. In other terms, qualitative graphs are homomorphic to the equivalence classes of the input space. This simple observation will constitute the grounds of a qualitative formalism based on graph homomorphisms (for oriented and not oriented graphs) that provide an elegant formal (and visual) framework for a qualitative envisioning of a system. Some basic properties of graph homomorphisms and their connection with our qualitative framework are given in the last section.

2 Motivation

Let's take a very simple piece of software source code written in the C programming language. For illustrative purposes the source code is given here, but we must stress the fact that we do not need to know the internal structure of a system for building its qualitative graph. All what we need to know is the inputs and their domain.

```
int prem,sec;
if (a >= b) prem = 1;
else prem = 2;
if (a >= 48) sec = 1;
else sec = 2;
if ((prem == 1)&&(sec == 1)) return 0;
if ((prem == 1)&&(sec == 2)) return 1;
 if ((prem == 2)&&(sec == 1)) return 2;
if ((prem == 2)&&(sec == 2)) return 3;
```

In our case we have two integer inputs a and b, which vary, say, from -100 to +100 (Fig. 1a). We suppose in the same time that our software, when compiled and executed, produces an observable result (given by the `return` statement). An automatic abstraction tool, developed by our team [9], determines heuristically input values that respect input domains and are situated at the frontiers of the state space regions. A state region is the set of inputs that yield the same output value. After several executions the following two dimension map with four distinct regions is built (Fig. 1a). The variable a increases horizontally, and variable b vertically. The point with coordinates, say, **(20, 80)** belongs to the region numbered 3, since the execution with inputs $a = 20$ and $b = 80$ produces the integer 3 as a return result. One can observe that the four regions are connected (and even convex) and separated by linear equations (automatically detected). This is due to the fact that the conditions appearing in the source code are linear functions of the inputs. It is often the case to have connected and even convex regions when we handle numeric parameters in software programming.

Let's now replace each region by a graph vertex. A vertex x will be connected with a vertex y with an arc labeled *a* if there is a point belonging to the region represented by x from where an "infinitesimal increase" (in our case all input variables are integers so the minimal change is 1) of the input variable a may lead the program to reach the region y (since our variables are bounded we could draw an additional vertex, representing an infinity, an error or an out of specifications state). We obtain a qualitative graph illustrated in Fig. 1b. This graph contains the same relevant information than the map but in a more compact form that does not depend on the map dimension.

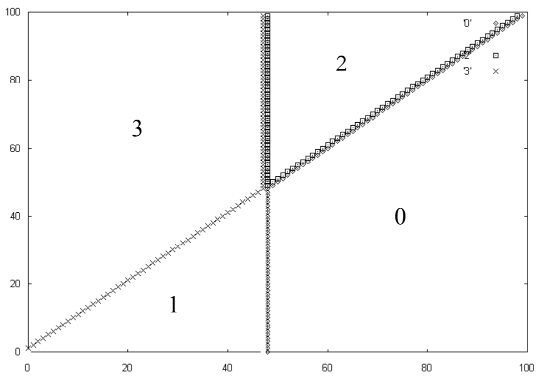

Fig. 1a

$$\begin{tikzcd}
3 \arrow[r, "a"] \arrow[d, bend right, "a"'] \arrow[dr, "a"] & 2 \arrow[d, bend left, "a"] \\
1 \arrow[u, "b"'] \arrow[r, "a"'] & 0 \arrow[u, "b"]
\end{tikzcd}$$

Fig. 1b

How this graph can be read? We can see for instance that from region 1 we cannot join directly the region 2: we must first visit region 3 by increasing both inputs. Sometimes we do not represent input labels on the arcs: in a non-oriented qualitative (Fig. 3) graph only the neighborhood information is represented. All graphs are reflexive (but we do not visualize loops on the vertices) since an infinitesimal change permits in most cases to stay in the same region. Regions with one isolated point (expressing sometimes an equality condition in the source code) do not admit loops and so constitute an exception but we don't wish to enter to those considerations in this presentation.

Visualizing by means of a graph the proximity of the different software functional areas provides a sort of software *phase space*, which permits a global understanding of the software behavior (software phase transitions). In some real time critical applications it is also important to know how the implemented software will react to some continuous modifications of its environmental inputs. Expressing a sort of topological representation of input variations is not only relevant for applications where inputs vary continuously. Functional frontiers and transitions are of paramount importance in software testing. One of the most common cause of programming errors [10] is a bad programming or misunderstanding of limit behavior.

In our example, a common programming error would consist for instance to write the first condition `(a<=b)`, instead of `(a>=b)` or to write a logical `or` instead of a logical `and` in a conditional statement. This sort of defects causes the deformation and the shifting of the surfaces separating the functional regions. Limit testing [8] consists in stressing the software with input values that are close or on to the separating surfaces.
In our qualitative terminology, limit testing means that we shall try to increase or decrease input data in order to visit all the vertices of our qualitative graph. Fig. 2 illustrates another automatic analysis result of a simple telecommunication protocol controller (with three control inputs).

3 A qualitative formalism based on graph homomorphisms

It is often the case in computer science, and especially in real time applications, to have an input domain with a natural metric relation. For instance, when a software processes many scalar inputs, the input graph can be considered as a sort of multidimensional grid. In some applications, inputs have a poset or lattice structure with a closeness relation *immediately greater than*. In other cases, before testing, software engineers partition the input domain into separate classes, choose a representative test vector in each class, and execute the software system. Here too one can say that some classes are close when they share some common attributes. The structure of the classes depends on the problem we are solving. A proximity relation can equally be extended to structures like letters, words, trees, since one often wishes to see what happens when he “smoothly” changes an input (i.e. changing a letter to another alphabetically close letter, a small tree is appended to another tree, etc.). Here, “smoothly” means that one performs the minimal change (given by the specification) to an input. Finite state machines as well as lattices can equally be considered as oriented graphs with a natural proximity relation. *All those observations allow one to consider the input domain as a graph, the input graph, with specific, domain dependent interconnections and its natural distance.* In order to illustrate our purposes let's now suppose that, before executing our software, we have found five kinds of input values. In Fig. 3, the input graph G is transformed in an output qualitative graph H by means of an algorithm ***f***. The edge (1, 2) means that we can "smoothly" change inputs to jump from class 1 to class 2. For instance, we could decide that the class 2, groups all input values that are negative but not both zero, class 4 could group values which are positive but not both zero, class 5 could group the point (0, 0), etc. This is a sort partition of the bidimensional plane. Of course, G could represent a partition of any input space of any dimension. The edge *(x, y)* means that in an execution of ***f*** (taking an input from input class 1) we get a result belonging to the class x.

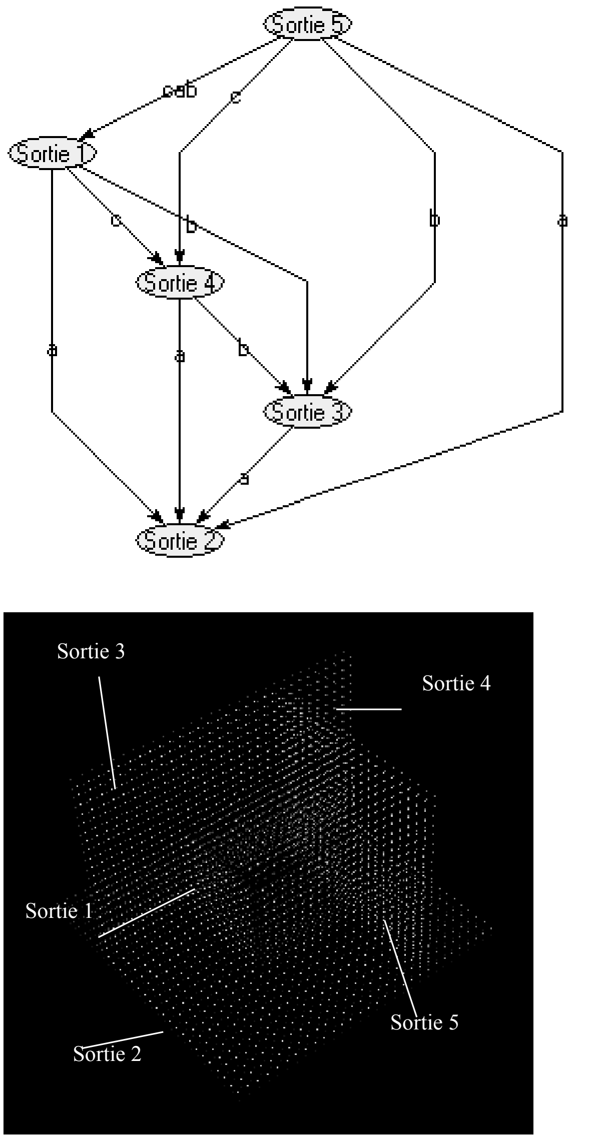

Fig. 2

After what, we change "continuously" the input from the class 1 to the class 2 and, in a second execution, we obtain an output belonging to the class y. After several executions (or physical observations of the system) we build the output graph H. All the executions are *independent* and *are not* (cannot be) *exhaustive*. The software system here has no memory. The software map ***f***, transforms an input graph G to a qualitative one. H contains the equivalence classes of the input graph G. Two vertices x and y of G are equivalent when *f(x)= f(y)*. This natural equivalence relation means that the output qualitative graph H is isomorphic to the quotient graph ***G/f*** which is homomorphic to the input graph G by the homomorphism naturally induced by the equivalence relation. In other words, the qualitative H is built (by observation) in such a manner, that becomes homomorphic to G and the software ***f*** (or, more generally, a system) is a graph homomorphism. Graph homomorphisms allow one to endow with the concept of continuity an inherently discontinuous field like software programming.

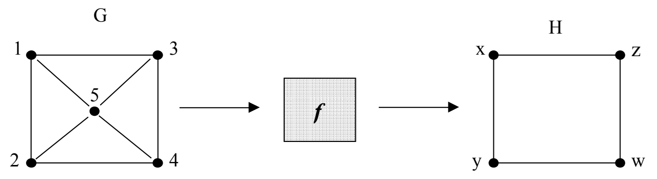

Fig. 3

The output qualitative graph H envisions the global dynamic behavior of the algorithm structure and, since it is homomorphic to the input domain, it integrates the topological input constraints independently on the way the algorithm has been implemented. More formally, the qualitative graph must preserve homomorphism invariants that are present in the input graph: they allow one to use non homomorphism properties: *if a graph does not respect an invariant, some vertices, labels or edges are lacking, or are misplaced*. For example, in Fig. 3, did one observe all the possible changes of behavior (all the edges of H)? Can one be sure that he will never have a transition from x to w, or y to z? Suppose, for instance, that x is the normal initial state of the system and that w is the error state. Can one be sure that before visiting the error state he will always visit the warning states y or z? As we shall see in the next paragraph, graph invariants show that H cannot be homomorphic to G since it does not respect an important homomorphism invariant (maximal *hole* number). One can formally conclude that the software, whatever is its implementation, and because of the topology of its input classes, must and will exhibit, in a future operational phase, a forbidden transition without visiting the warning states.

Suppose now that a system admits two integer inputs, a and b, and exhibits four possible classes of output behavior. Are all qualitative graphs with four vertices admissible? Can one observe the qualitative graph of Fig. 4? Without delving into further details (homomorphism invariants for oriented graphs are discussed in [11]) we can say, for that example, that the region 3 must be necessarily connected with a label *a* to the region 2. In fact, the qualitative graph H is homomorphic to a bi-dimensional oriented grid which has the isotropy property: two vertices which are connected by a path expression of the form $a^n.b^m$ must be also be connected by a path expression $b^m.a^n$. Isotropy is an homomorphism invariant which is not the case for the regions 1 and 2. So a label *a* is lacking between the regions 3 and 2, since this constitutes the only way to connect correctly the regions 1 and 2. We conclude that the tester must design test cases in order to exhibit the specific output transition after an increase (with a sequence of independent executions) of the input a.

$$\begin{tikzcd}
3 \arrow[r, "b"] & 2 \\
1 \arrow[u, "b"] \arrow[r, "a"'] & 0 \arrow[u, "b"']
\end{tikzcd}$$

Fig. 4

As we said before an input graph G could abstract a partition of the bidimensional plane. Since all computer values are discrete, the real plane is a huge grid where values are connected when there is no an intermediate value between two decimal points (the grid vertices). We conclude that even input graph G must be homomorphic to that original grid. Homomorphism composition can be further used in the case where the output graph H is an input graph of another system, say ***s***, providing a new qualitative graph J. We see in that manner that graph homomorphisms provide an elegant and coherent framework for data abstraction and system composition. The next paragraph gives a more formal flavor to those observations with some basic properties of homomorphisms of non-oriented graphs.

4 Mathematical framework

We adopt conventional notations for graphs G(X, U) with X the set of vertices and U the set of edges. We note *x~y* the adjacency of the two vertices. Graphs are connected and reflexive but we do not visualize loops. We note $d_G(x, y)$ the natural distance in a connected graph G that is, the length of the shortest path, linking x to y. We note I_n as the path of length n. Grids noted $G_{m,n,p\ldots}$ are cartesian products of paths.

An **homomorphism** [12], is a map $h{:}G\to H$ preserving adjacency: i.e. *x~y* implies *h(x)~h(y)*. Our graphs being reflexive, this definition is equivalent to a non expanding map: $d_G(x,\ y) \geq d_H(h(x),h(y))$. Homomorphisms will always be onto. When G = H we say that we have an *endomorphism*. Idempotent endomorphisms are also called **retractions**. So retractions are homomorphisms which leave invariant a subgraph G', called a retract of G. Graph homomorphisms, as well as retractions constitute a very active area of research in graph theory [13], [14].

A **contraction** is an onto homomorphism *h:G→H* where the inverse image of every vertex of H is a connected subgraph of G. We note ***G/h*** the quotient graph induced by the kernel of h. A partition is elementary when all the equivalence classes contain only one element, except one class that contains exactly two adjacent vertices. More particularly, a contraction is elementary when it induces an elementary partition.

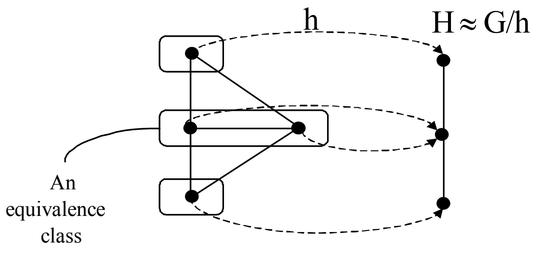

Fig. 5

An elementary contraction can be viewed as gluing two adjacent vertices following their common edge. Fig. 5 illustrates an elementary contraction h and its kernel ***G/h***.

An **homomorphism invariant** is a non negative real valued function ∂ verifying: ∂(G)≥∂(h(G)) for any homomorphism h. The number of vertices, edges as well as the diameter are trivial invariants. It is easy to observe that any contraction is the commutative composition of elementary contractions. That means that if a property is an invariant for any elementary contraction it is also, by induction, a contraction invariant. An immediate property of that observation is that contractions preserve planarity. For a connected subgraph G' of G, we define *discon*(G') as the number of connected components (possibly a single vertex) that we obtain when we remove G'. We call it the *disconnecting capacity* of G'. It is easy to prove that for any contraction h we have: *discon*(G')≥*discon*(h(G')). This property yields an interesting corollary (that can easily generalized for higher dimensions): any bi-dimensional grid of an odd size m (i.e. $G_{m,m}$), with m≥3 cannot be contracted to any path I_{m+1}. In Fig. 6 we illustrate this: the grid $G_{2,2}$ cannot be contracted to the path I_3. To have an idea of the general demonstration note, in Fig. 6, that the central vertex c has a disconnecting capacity of 1 since its removal does not disconnect the graph. At the same time it is at a maximal distance of 2 from all the other vertices, so it cannot be homomorphically mapped to the two outer vertices of I_3. So, if a contraction exists, its image h(c) is necessarily a vertex in the middle of I_3, which disconnects I_3, thus increasing its disconnecting capacity, which is impossible.

The maximal disconnecting capacity of a graph, ***mdc*(G)**, is the maximum *discon*(G') that we can obtain from a subgraph G'. Since *discon*(G') does not increase, *mdc*(G) is a contraction invariant. In Fig. 7, we have *mdc*(G)=2 and *mdc*(H)=3.

A cycle contains a **chord** when two not subsequent vertices of the cycle are connected. Chordless cycles that are also retracts are called **holes**. For instance, in Fig. 3, the cycle [1, 2, 4, 3] is a chordless cycle since opposite vertices (like 1 and 4) are not adjacent, but is not a hole, since there is no possible retraction on this cycle. Let ***hole*(G)** be the greatest length of a hole in G. For instance, in Fig. 3, *hole*(G) = 3 and *hole*(H) = 4. The only holes of grids are the cycles of length 4. So the hole number of any grid, of any dimension, is 4. It can be proved by induction that elementary contractions do not increase the hole number, so *hole*(G) is also a contraction invariant.

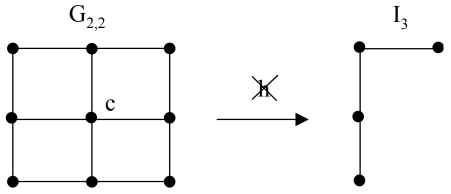

Fig. 6

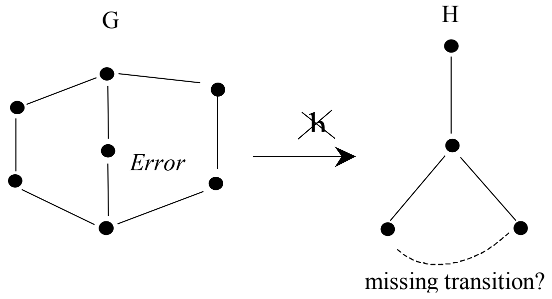

Fig. 7

As we said in the previous sections, if we assume the connectivity of functional regions, contraction invariants can be used to constraint qualitative graphs. For instance, in Fig. 6, the non homomorphism property based on the disconnecting capacity invariant says that if we observe a software with two integer inputs partitioned in 9 classes of a bidimensional plane (combination of negatives and positive coordinates) it is impossible to observe a completely linear behavior. The *mdc*(G) invariant, in Fig. 7, permits to say that when inputs of a software system follow a cyclic finite state machine with a central *Error* state, then the observed output region transitions cannot have a star-like topology. A transition edge is missing. The hole number permits to conclude that any system with any number of scalar inputs cannot exhibit a 5-cycle behavior without, at least, a missing transition among the states of the cycle.

Contraction invariants can also be used to deduce some important properties of the input graph. Let's take an interesting example. Suppose one has a system (like a robot) that he controls with two integer variables forming a planar input graph. Suppose now that this robot exhibits a good behavior (this is a vertex of the output qualitative graph H). One wants to be sure that small perturbations on input variables (the graph G) will not suddenly change robot's behavior. That is, we want to be sure that whatever would be the output behavior, there is the possibility of maintaining the robot in the same configuration while smoothly changing its control variables (a sort of controllability). That necessitates all the control surfaces of our system to be connected since we wish to visit all the points of a region without jumping into another. In other words there must be a contraction between G and H. Suppose now that, during the testing of our robot, we observe a graph H containing a 5-cycle hole, or, a graph with so many transitions that make it no planar. In both cases, we can conclude that there is no possibility of contraction. Thus, inverse images are not connected and the system is not controllable.

5 Conclusion

Qualitative reasoning has been applied for the testing of protocol oriented software components and proved to be very beneficial. Software designing, programming, testing and debugging are very complex and error prone human activities. Conventional software engineering methods and tools are very powerful but lack of a global qualitative formalism to help the engineer to understand the system behavior. A formalism based on graph homomorphisms has been presented. Software is viewed as a sort of continuous map transformation between two abstract data spaces likewise a phase transition system. The qualitative graph envisions the global of the software system and respects some constraints that are independent of its internal structure. Those constraints, called invariants, express topological properties of graph homomorphisms. They can be used to infer the possible shapes of the qualitative graph. An automatic abstraction tool has been presented. Many questions may arise: can we abstract all common input data structures with a proximity relation? Is it possible to express more quantitative information in the qualitative graph labels? How do we handle state machines, time and memory? How this formalism can be extended to physical and/or artificial systems? Do endomorphisms or retractions express some specific classes of software behavior? Can we classify software applications according to the properties of their endomorphisms (that is, the properties of the generated monoid)? How the map composition of homomorphisms can express system integration? Is it possible to express some software errors as the composition of the correct map with an error map that one could study in more details? All those questions are open but we think that the main contribution of our formalism resides in the fact that ***it proposes a bridge between qualitative system analysis and a very seminal area of applied mathematics.***

References:

[1] Charles Rich. Artificial intelligence and software engineering: the programmer's apprentice project. *In Proceedings of the 1984 annual conference of the ACM*, 1984.

[2] Althoff K.-D. Case-Based Reasoning. In *Handbook on Software Engineering and Knowledge Engineering.* Vol. 1 "Fundamentals", Chang, S. K., Editor, World Scientific. pages 549-588, 2001.

[3] Bert Bredeweg and Peter Struss. Current Topics in Qualitative Reasoning. *AI Magazine*, Winter 2004.

[4] Antoine Missier, Spyros Xanthakis and Louise Trave-Massuyes. Qualitative Algorithmics using Order of Growth Reasoning. *In Proceedings ECAI* 94, pages 750-754, 1994.

[5] Wolfgang Mayer and Markus Stumptner. Model-Based Debugging using Multiple Abstract Models. In Procs of the 5th IWAAD, pages 55-70, 2003.

[6] Louise Travé-Massuyès et al. *Le raisonnement qualitatif pour les sciences de l'ingénieur*, chapter 12, Editions Hermès, France, 1997.

[7] Flemming Nielson, Hanne Riis Nielson. Principles of Program Analysis, Springer, 1998.

[8] Spyros Xanthakis, Pascal Régnier and Constantinos Karapoulios. *Le test des logiciels*, Etudes et logiciels informatiques. Editions Hermès, France, 2000.

[9] Virginie Guiraud. Visualisation du comportement dynamique des logiciels numériques. *Rapport de stage, société SOPRA*, 2001-2003.

[10] Steven J. Zeil Faten H. Afifi Lee J. White. Detection of linear errors via domain testing. *ACM Press*, New York, NY, USA, 1992.

[11] Constantinos Karapoulios. *Raisonnement Qualitatif Appliqué au Test Evolutif des Logiciels*. Thèse de Doctorat, I.R.I.T, Université Paul Sabatier, Toulouse, France, Juillet 1999.

[12] Pavoll Hell and Jaroslav Nesetril. *Graphs and Homomorphisms*. Oxford Lecture Series in Mathematics and its Applications, Oxford University Press, 2004.

[13] G. R. Brightwell and P. Winkler, Gibbs measures and dismantlable graphs, *J. Graph Theory* 11 (1987) 71-79.

[14] Gena Hahn and Gary MacGillivray. *Graph homomorphisms: computational aspects and infinite graphs.* Research report, Université de Montreal, June 2002.