Network Motifs in Computational Graphs: A Case Study in Software Architecture

Sergi Valverde¹ and Ricard V. Sole¹,²
¹ICREA-Complex Systems Lab, Universitat Pompeu Fabra, Dr. Aiguader 80, 08003 Barcelona, Spain
²Santa Fe Institute, 1399 Hyde Park Road, New Mexico 87501, USA

Complex networks in both nature and technology have been shown to display characteristic, small subgraphs (so called motifs) which appear to be related to their underlying functionality. All these networks share a common trait: they manipulate information at different scales in order to perform some kind of computation. Here we analyse a large set of software class diagrams and show that several highly frequent network motifs appear to be a consequence of network heterogeneity and size, thus suggesting a somewhat less relevant role of functionality. However, by using a simple model of network growth by duplication and rewiring, it is shown the rules of graph evolution seem to be largely responsible for the observed motif distribution.

PACS numbers: 05, 89.75

I. INTRODUCTION

Many natural and artificial systems are describable as networks of interacting components [1–4]. The network is a medium that allows resource sharing often involving an efficient transport of energy (metabolism, power grid), matter (highways, airport webs) or information (cellular communication, Internet). The architecture of complex networks can be explored at different scales, from the overall properties defined by average measures such as path length or clustering, correlations or degree distributions to the more fundamental features displayed by small subsystems. In this context, it has been shown that some special, small subgraphs -so called motifs- seem to be particularly relevant in describing the architecture of complex networks [5]. Motifs have been suggested to be the functional building blocks of network complexity. Are some subgraphs more common than others because their functional relevance?

An alternative view is that the rules of network growth can by themselves favour some subgraphs with no special relation to the underlying functionality. Actually, this seems to be the case for the structure of the protein-protein interaction map. In spite that proteins perform functions, the overall architecture of the protein network is easily reproduced by means of a simple model of node duplication plus rewiring [6]. Such properties include scale-freeness, small world features and even hierarchical organization and protomodularity [7]. Mounting evidence suggests that many key features of complex networks (including motifs) might be strongly tied to the global network structure [8].

If functional constraints to network architecture have to be considered, one particularly relevant aspect of network complexity is associated to the presence of some underlying computational process. Computation is a key ingredient of any complex adaptive system (CAS). By storing and processing information, CAS are able to predict (and adapt to) external fluctuations. Computation occurs in both natural and artificial systems [9], although the building process that creates the computational structure is different. This is actually one of the most important points here: are the rules of designed and evolved systems completely different? Biological networks are largely originated through tinkering [10–12]: new components are obtained by re-using old ones, mainly by duplicating them. In spite of the apparent limitations of such mechanism, it allows to discover good designs [13]. More complex computations can be developed as the network size is increased and new functions can emerge.

How is computation linked to network structure? Tentative answers, to be developed here, can be obtained by looking at a very important class of computationally-driven networks: software systems. They offer a unique opportunity of exploring different levels of complexity with well-defined functional traits. As opposed to most examples of evolving computational networks, extensive databases storing software evolution registers exist and involve a high degree of detail. Here we analyse the largest data set of software maps explored to date (83 different systems). The main goal of our study is to see if functionality, as opposed to network evolution, is a main constraint to the distribution of network motifs in real graphs. The paper is organized as follows. In section II an overview of the software systems analysed here is given. In section III, the statistical patterns of networks motifs in a large set of software systems is presented and the presence of scaling relations and the size-dependent frequency of motifs analysed. In section IV a model of duplication and rewiring is used in order to reproduce the structure of motifs of a large software map. In section V

FIG. 1: Examples of common network motifs with \( n = 4 \) elements found in software graphs. Here each node is a class and arrows indicate static dependencies among classes (see text).
a general discussion is presented.

II. SOFTWARE NETWORKS

Programming languages describe software systems [15]. Every computer program has a textual representation following syntactic rules dictated by a programming language [16]. The program is decomposed in a number of simpler software entities, or logical elements that are given a unique name. Software entities include things as data objects, instructions, subprograms, or modules. A hierarchy or natural ordering between software entities is prompted by modern programming languages. At the lowest level, a program is viewed as a sequence of simple software entities, or logical elements that are data objects, instructions, subprograms, or modules. A machine instructions. Sequences of related instructions are enclosed in subprograms. At the highest level, there are modules or logical containers grouping simpler software entities. Often, modules are defined as functional blocks but there are no strict principles driving module composition.

It is useful to depict the complex structures defined in computer programs by means of a graph [14], where nodes represent software entities and links represent syntactical relationships between modules, subprograms and instructions. In this paper, we will focus in a particularly interesting subset of software entities: the collection of modules and their static interactions (also called software architecture). While it is widely acknowledged that software evolution depends on its architecture, very little is known about the cause and effect relationships between design practices and evolution outcomes [24]. In order to understand the relation between structure and artificial evolution, we have envisaged a network model of software architecture, hereafter called software map or software network. Here, we show how the evolution of computer programs can be understood by recovering and analyzing their software networks at different stages of development.

Following [14, 20], figure 2 shows the text of an incomplete C++ program (see fig. 2a) and its corresponding software map (see fig. 2b). The program text reads from left to right and top to bottom. The software network $\Omega = (V, L)$ of this C++ program is recovered by means of a very simple lexical analysis. First, we identify the vector of all module names (also class in C++) given by $W = (w_i)$ is a module definition associated to node $v_i$ and $w_l \notin \{w_1, w_2, w_4, w_7\}$ is a module reference associated to node $v_l$. A directed link $\{v_i \rightarrow v_j\} \in L$ signals a dependency from module definition $w_k$ to module reference $w_l$. Link directionality reflects name ordering in the C++ program, that is, $k < l$. There are two types of module dependencies: association (also “has-a” relationship) or inheritance (or “is-a” relationship). The purpose of these dependencies is to establish a logical organization of the system. However, our analysis is centered in the study of topological patterns and does not take into account detailed relationship semantics. In an association, referenced node $v_j$ is nested in the C++ block of module $v_i$. This block is always bracketed by the symbols { and }. In an inheritance, referenced node $v_j$ always follows the C++ sequence “: public” after the referencing node $v_i$ (see fig.2a). Repeated links are not considered in the following analysis.

Software maps capture the topology of complex software systems. In particular, these maps provide a quantitative approach to the evolution of technology. They are actually evolving entities and somewhat inhabit an intermediate zone between computing machines and neural structures. We have shown software networks to be scale-free and small world [14, 20, 22]. Software networks can be described under a statistical physics perspective.

class Point {  
  int row, column;  
};

class Chessmen {  
  Point pos;  
  int &getValue();  
};
class Move {  
  Point start_pos;  
  Point end_pos;  
  bool isAllowedMove();  
};
class Pawn : public Chessmen {  
  Move *moves;  
  int &getValue();  
};

FIG. 2: (a) A piece of C++ code from a chess-playing program and (b) the corresponding software map or network model displaying the collection of modules and their logical dependencies. The only information required to recover the software map is the set of module names (highlighted in bold in (a)) and their relative locations in the C++ program (see text). Notice how nodes are labelled with their names and links are decorated with relationship type.

nodes $v_i \in V$ in the software network.

Design of any non-trivial function involves the interaction of at least two modules [18]. Static module interactions can be depicted from relative positions of names in $W$. Let's assume that $w_k \in \{w_1, w_2, w_4, w_7\}$ is a module definition associated to node $v_i$ and $w_l \notin \{w_1, w_2, w_4, w_7\}$ is a module reference associated to node $v_l$. A directed link $\{v_i \rightarrow v_j\} \in L$ signals a dependency from module definition $w_k$ to module reference $w_l$. Link directionality reflects name ordering in the C++ program, that is, $k < l$. There are two types of module dependencies: association (also “has-a” relationship) or inheritance (or “is-a” relationship). The purpose of these dependencies is to establish a logical organization of the system. However, our analysis is centered in the study of topological patterns and does not take into account detailed relationship semantics. In an association, referenced node $v_j$ is nested in the C++ block of module $v_i$. This block is always bracketed by the symbols { and }. In an inheritance, referenced node $v_j$ always follows the C++ sequence “: public” after the referencing node $v_i$ (see fig.2a). Repeated links are not considered in the following analysis.

Software maps capture the topology of complex software systems. In particular, these maps provide a quantitative approach to the evolution of technology. They are actually evolving entities and somewhat inhabit an intermediate zone between computing machines and neural structures. We have shown software networks to be scale-free and small world [14, 20, 22]. Software networks can be described under a statistical physics perspective.
In this section, we extend our previous topological studies by analyzing software networks at the level of network subgraphs, or subsets of connected nodes in a network. The statistics of subgraphs provides important information about network structure. It has been claimed that overrepresented subgraphs (i.e., motifs) signal key building blocks of networks [5]. This might be the case for regulatory networks, where specific subgraphs (i.e., feed-forward loops) perform information processing functions [29]. A particular class of subgraphs, cycles, have received considerable interest. Cyclical dependencies in software maps imply that a module is related to itself, which may be acceptable, unacceptable or required [21]. Ambiguity in the functional meaning of cycles suggests that subgraphs in software graphs are not strictly related to well-defined functions. The ubiquity of subgraphs in software networks seems to be a consequence of top-down mechanisms of software organization and not a consequence of selective pressures.

Following the method outlined in previous section, we have recovered and analysed a large dataset of software maps \( \Omega \) from 83 reverse-engineered C++ software systems. A given graph can be characterized by a degree sequence. For the whole graph \( \Omega \) each node has a degree sequence given by the indegree list \( \{K_i\} \) and the outdegree list \( \{R_i\} \) (with \( i = 1, \ldots, N \)). The lists would be completed by the so called mutual edges \( \{M_i\} \), i.e. cases where there is a pair of edges in both directions between two nodes. For each subgraph \( \Omega_i \subset \Omega \) of size \( n \) (here \( n=3,4 \)), another degree sequence would be provided by two new lists, now \( \{k_j\} \) and \( \{r_j\} \) for the in- and outdegrees, respectively. For example, for the left subgraph in figure 1, we would have \( \{k_j\} = \{0, 1, 1, 2\} \) and \( \{r_j\} = \{2, 1, 1, 0\} \).

Network motifs are defined in terms of subgraphs which appear much more often than expected from pure chance. Specifically, they occur with a high frequency compared with the expected from an ensemble of randomized graphs with identical degree structure [5]. The random networks are generated by means of the switching rule. For every pair of links \( i \rightarrow j \) and \( u \rightarrow v \) in the original software network, we add the pair \( i \rightarrow v \) and \( j \rightarrow u \) in the randomized network. This rule keeps intact the in/out-degree sequences but destroys degree-degree correlations. The statistical significance of a given subgraph \( \Omega_i \) is described by its \( Z \) score [5], defined as:

\[
Z(\Omega_i) = \frac{N_{\text{real}}(\Omega_i) - \langle N_{\text{rand}}(\Omega_i) \rangle}{\sigma(N_{\text{rand}}(\Omega_i))} \tag{1}
\]

Here \( N_{\text{real}}(\Omega_i) \) is the number of times the subgraph appears in the network, whereas \( \langle N_{\text{rand}}(\Omega_i) \rangle \) and \( \sigma(N_{\text{rand}}(\Omega_i)) \) refer to the mean and standard deviation of its appearances in the randomized ensemble, respectively. In order to be significant, it is required that \( |Z(\Omega_i)| > 2 \). When \( Z(\Omega_i) > 2 \) (\( Z(\Omega_i) < -2 \)) the motif (antimotif) is considered to be more (less) common than expected from random. In table 1 the results from our analysis are shown for some typical software networks. A handful of these subgraphs appear to be present in all software systems analysed and also in both electronic circuits and biological networks involving computation. This is the case of Bi-parallel (S904), Bi-fan (S204), the feed-forward loop (S38) and its close variants (such as S2186 and S408). Such a common point might be easily interpreted in functional terms: similar subgraphs are abundant because they are selected or chosen to perform a given function or task. As shown below, no evidence from statistical patterns supports such view.

Assuming sparse graphs (\( \langle K \rangle \ll N \)), the probability of a given subgraph \( \Omega_i \) can be estimated. Following Itzkovitz et al., we can see how this is calculated using the first subgraph in figure 1. Here we have: \( \{K_j\} = \{2,1,1,0\} \) and \( \{R_j\} = \{0,1,1,2\} \). The idea is to compute the different probabilities associated to each directed edge linking all pairs of nodes. For example, the probability of having a directed link from node 1 to node 2 (for \( K_1 R \ll N \langle K \rangle \)) is approximately:

\[
P(1 \rightarrow 2) = \frac{K_1 R_2}{N \langle K \rangle} \tag{2}
\]

which can be interpreted as follows [23]: we perform \( K_1 \) attempts for the first node to connect to the target node with a probability \( R_2 / N \langle K \rangle \). Similarly, we would have:

\[
P(1 \rightarrow 3) = \frac{(K_1 - 1) R_3}{N \langle K \rangle} \tag{3}
\]

being the approach used for all edges. The average number of appearances of \( \Omega_i \) is finally computed by averaging Itzkovitz et al. [23] have shown that the average number of appearances \( \langle G \rangle \) of a given subgraph is given by a product of moments of different orders of the indegree, outdegree and mutual degree distributions:

\[
\langle G \rangle \sim N^{n-g_a-g_m} \langle K \rangle^{g_m} \langle Y \rangle^{g_n} \prod_j \left( \frac{K_j}{k_j} \right)^{g_a} \left( \frac{R_j}{r_j} \right)^{g_m} \left( \frac{M_j}{m_j} \right)^{g_n} \tag{4}
\]

where \( g_a \) and \( g_m \) are the number of single and mutual edges. The approximation assumes uncorrelated, sparse networks (\( \langle K \rangle \ll N \)). Both conditions are met by software maps [25]. These mean-field quantities can be used as a null model estimate of the number of motifs, and eventually to detect stray, significant deviations from random. Since different motifs are found in different systems [5] they can actually allow us to identify the basic functional blocks for a given class of networks.

By exploring our collection of software graphs, we determined \( < G > \) for real nets (indicated as \( N_{\text{real}} \) in fig. 2) and compared them to \( N_{\text{rand}} \). Here software maps with a size \( N > 10 \) have been analysed. Two groups have been chosen, involving medium-sized graphs (\( N < 300 \)) and large graphs (\( N > 300 \)). The previous set is compared with results from other networks involving computational
FIG. 3: Network motifs with \( n = 3 \), 4 elements found in software graphs. The numbers of node and edges for each network are shown. The most frequent motifs were classified in distinct rows for each type of system: medium software systems, large software systems, gene regulatory nets, neural networks and digital electronic circuits. For each motif, we display the number of occurrences in the network (\( N_{\text{real}} \)), the number of occurrences (\( N_{\text{rand}} \pm SD \)) in a set of 100 randomized network versions and a qualitative measure of its statistical significance as given by the Z score (see text). Medium and large software networks share a large amount of motifs but we found larger variability in the medium data set. A remarkable difference is motif 2190 (the last motif in the second row), which appeared only in the context of large software systems.

In order to explore the question of how relevant is the overall network structure in conditioning the frequency of given subgraphs, we should consider the global structure of the network. The first approximation is to consider the degree of heterogeneity as provided by the distribution of links. Software systems have a well-defined scale-free indegree distribution

\[
P_i(k) = \frac{i}{k_0^{i-1}} \left( k + k_0 \right)^{-\gamma_i}
\]

with \( \gamma_i \sim 2 \). A mean value \( \langle \gamma_i \rangle = 2.09 \pm 0.05 \) has been obtained by averaging over all the systems studied here. The distribution of scaling exponent is strongly peaked around \( \gamma_i = 2 \). The outdegree distribution \( P_o(k) \) is much

<table>
<thead>
<tr>
<th>Network</th>
<th>Nodes</th>
<th>Edges</th>
<th>( N_{\text{real}} )</th>
<th>( N_{\text{rand}} \pm SD )</th>
<th>( N_{\text{real}} )</th>
<th>( N_{\text{rand}} \pm SD )</th>
<th>( N_{\text{real}} )</th>
<th>( N_{\text{rand}} \pm SD )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Software Networks</strong> (medium)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fairuse</td>
<td>106</td>
<td>180</td>
<td>41</td>
<td>11.6±3.3</td>
<td>8.94</td>
<td>18</td>
<td>7.3±1.5</td>
<td>3.01</td>
</tr>
<tr>
<td>Aime</td>
<td>143</td>
<td>319</td>
<td>n/a</td>
<td></td>
<td></td>
<td></td>
<td>33</td>
<td>9.5±5.5</td>
</tr>
<tr>
<td>FileiD221a</td>
<td>186</td>
<td>331</td>
<td>77</td>
<td>29.4±6.5</td>
<td>7.86</td>
<td>230</td>
<td>30±19.2</td>
<td>10.46</td>
</tr>
<tr>
<td>Aztec</td>
<td>255</td>
<td>391</td>
<td>68</td>
<td>26.6±4.5</td>
<td>7.82</td>
<td>30</td>
<td>10±2.4</td>
<td>4.38</td>
</tr>
<tr>
<td>Exult</td>
<td>261</td>
<td>504</td>
<td>107</td>
<td>56.3±8.4</td>
<td>6.01</td>
<td>86</td>
<td>30±2.8</td>
<td>6.75</td>
</tr>
</tbody>
</table>

| **Software Networks** (large) |       |       |                       |                          |                       |                          |                       |                          |
| blender26         | 495   | 934   | 486                   | 138±30.3                 | 11.4                  | 33                       | 16±5.2               | 3.2                      |
| glk221            | 748   | 147   | 644                   | 180±33.8                 | 13.4                  | 119                      | 25±7.5               | 12.5                     |
| vdk               | 771   | 1362  | 512                   | 262±39.9                 | 6.27                  | 295                      | 69±14.5             | 15.8                     |
| java2             | 1364  | 947   | 816                   | 180±33.5                 | 17.7                  | 173                      | 48±10.8              | 11.5                     |
| porally           | 1993  | 4987  | 22750                 | 180±171                 | 122                  | 3848                     | 322±34.2             | 103.1                   |

| **Gene Regulation** (transcription) |       |       |                       |                          |                       |                          |                       |                          |
| E. coli           | 424   | 519   | 40                    | 7±3                     | 10                    | 203                      | 47±12               | 13                      |

| **Neurons**       |       |       |                       |                          |                       |                          |                       |                          |
| C. elegans        | 252   | 509   | 125                   | 90±10                   | 3.7                   | 127                      | 55±13               | 5.3                      |

| **Electronic Circuits** (forward logic chips) |       |       |                       |                          |                       |                          |                       |                          |
| s15850            | 1038  | 14240 | 424                   | 2±2                     | 285                   | 1060                     | 1±1                  | 1200                     |

| **Multi-X** (incomp.) |       |       |                       |                          |                       |                          |                       |                          |
| **Multi-Z**        |       |       |                       |                          |                       |                          |                       |                          |
| **Multi-Y**        |       |       |                       |                          |                       |                          |                       |                          |

<table>
<thead>
<tr>
<th><strong>Network</strong></th>
<th>Nodes</th>
<th>Edges</th>
<th>( N_{\text{real}} )</th>
<th>( N_{\text{rand}} \pm SD )</th>
<th>( N_{\text{real}} )</th>
<th>( N_{\text{rand}} \pm SD )</th>
<th>( N_{\text{real}} )</th>
<th>( N_{\text{rand}} \pm SD )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Software Networks</strong> (large)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>blender26</td>
<td>495</td>
<td>934</td>
<td>486</td>
<td>138±30.3</td>
<td>11.4</td>
<td>33</td>
<td>16±5.2</td>
<td>3.2</td>
</tr>
<tr>
<td>glk221</td>
<td>748</td>
<td>147</td>
<td>644</td>
<td>180±33.8</td>
<td>13.4</td>
<td>119</td>
<td>25±7.5</td>
<td>12.5</td>
</tr>
<tr>
<td>vdk</td>
<td>771</td>
<td>1362</td>
<td>512</td>
<td>262±39.9</td>
<td>6.27</td>
<td>295</td>
<td>69±14.5</td>
<td>15.8</td>
</tr>
<tr>
<td>java2</td>
<td>1364</td>
<td>947</td>
<td>816</td>
<td>180±33.5</td>
<td>17.7</td>
<td>173</td>
<td>48±10.8</td>
<td>11.5</td>
</tr>
<tr>
<td>porally</td>
<td>1993</td>
<td>4987</td>
<td>22750</td>
<td>180±171</td>
<td>122</td>
<td>3848</td>
<td>322±34.2</td>
<td>103.1</td>
</tr>
</tbody>
</table>

| **Gene Regulation** (transcription) |       |       |                       |                          |                       |                          |                       |                          |
| E. coli      | 424   | 519   | 40                    | 7±3                     | 10                    | 203                      | 47±12               | 13                      |

| **Neurons** |       |       |                       |                          |                       |                          |                       |                          |
| C. elegans  | 252   | 509   | 125                   | 90±10                   | 3.7                   | 127                      | 55±13               | 5.3                      |

| **Electronic Circuits** (forward logic chips) |       |       |                       |                          |                       |                          |                       |                          |
| s15850      | 1038  | 14240 | 424                   | 2±2                     | 285                   | 1060                     | 1±1                  | 1200                     |

The reason of using biological networks as a reference system is twofold. First, the chosen systems are known to perform computational tasks (or can be described by means of an equivalent computational circuit). The second is that it has been conjectured that both natural and artificial networks might share some commonalities relating the mechanisms that shape their evolution [12]. Common features might reflect common functional traits, but also (as shown below) common rules of graph evolution with no special functional meaning.
steeper, and seems better described by a broad scale distribution, not far from the exponential limit. This is actually the opposite situation considered in [23] but is not difficult to show that it is essentially symmetric in the theoretical treatment.

For the regime considered here, it was shown that \( \langle G \rangle \) follows a scaling law

\[
\langle G \rangle \sim N^\alpha
\]  \hspace{1cm} (6)

Specifically, for a given pair \( \{n, g\} \) and a given scaling exponent, we have:

\[
\langle G \rangle \sim N^{n-g+s-\gamma_1+1}
\]  \hspace{1cm} (7)

where \( s \) is the maximum in-degree for our case. This scaling is actually valid for \( 2 < \gamma_1 < s + 1 \).

Two examples of the observed scaling laws are shown in figure 4 for two very frequent software motifs. The two motifs shown are very different, with \( n = g = 4 \) and (a) \( s = 2 \) (subgraph S904) and (b) \( s = 3 \) (subgraph S2190). Using (3) the expected number of times a given motif appears would scale as \( \langle G \rangle \sim N^{s+1-\gamma_1} \) and using the scaling exponent \( \gamma_1 = 2.09 \pm 0.06 \) a scaling law \( \langle G \rangle \sim N^{0.9} \) should be predicted for uncorrelated, sparse graphs. The predicted scaling is recovered from real data (see figure 4), with \( \alpha(S904) = 0.91 \pm 0.13 \) and \( \alpha(S2190) = 0.85 \pm 0.15 \), thus indicating that the average trends are consistent with the expectation from random scale-free networks. It confirms the validity of Itzkovitz et. al [23] prediction and its agreement with a set of real networks. This agreement is an interesting result, particularly if we remember that this is a designed system to perform given functions. The fact that we obtain the scaling law expected for the random, scale-free graph reveals that the observed scaling in motif abundances are a consequence of top-down constraints derived from graph evolution.

IV. Duplication-Based Evolution

The topology of software architecture emerges from designed evolution. On top of the process, there must be a basic building plan towards a final function or set of functions. The engineer foresees the outcome of its work. But there are a number of strong constraints no less important and operating through the software building process. On the one hand, modular structures are shaped through parallel paths of evolution. Different blocks will be involved in more specific subfunctions. On the other hand, increased complexity leads to conflicts between different subparts. This is reflected for example at the topological level: small software maps tend to display tree structure, whereas larger systems typically display much more complex patterns [14]. The common overall structure detected in software graphs in terms of the degree distribution (and other average properties) suggests that the final topological patterns might be strongly constrained.

We conjecture that the abundance of subgraphs in software networks relates to universal mechanisms of network growth underlying their evolution. Real software maps tend to display motif generalizations or subgraphs having an structure comprising many replicas of the 4-motifs observed here. These structures are highly redundant. This suggests a very simple duplication-based mechanism of subgraph generation. New modules depend upon other modules in order to provide useful functionalities. And it seems reasonable to assume that similar modules will share a large number of module dependencies. In a related software engineering study [27], structural similarities in C++ software at the module (class) level have been analyzed. They have found quantitative evidence of structural duplications. However, they did not provide any model explaining the origin of duplications.

Figure 5 shows a detailed example illustrating how top-down duplication works in software development. Imagine we want to add a new software module representing the queen, in the previous chess-playing program (see fig. 2a). First, we will add a new module declaration, which is conveniently named Queen (see fig. 5a). Because a queen is a type of chessmen, it seems reasonable to make this module depend upon the same modules referenced by similar modules, which in this case is the Pawn. By using the Pawn module and its neighborhood as a template, we add an inheritance relationship from the Queen to the Chessmen (see fig. 5b). Duplication is completed with the addition of a collaboration relationship from Queen to Move (see fig. 5c). Comparison between final network (see fig. 5c) with the initial network (see fig. 2c) reveals a new bi-parallel subgraph and twice the number of bi-fan subgraphs.

An example of this process taking place in real software development is shown in figure 6. Here a given subsystem
FIG. 5: From (a) to (c), an illustration of duplication mechanism in software map evolution. Time flows from top to bottom. Here, a new module Queen is introduced by cloning the links of the similar module Pawn. Every stage displays the evolving C++ program (right) and its corresponding software network (left) reconstructed by the method described in section 2. New text is enclosed in a box. Note how duplication of links in the software map is parallel to duplication of code in the C++ program.

inside a growing software graph is displayed at different development stages. Duplication of nodes seems to be at work, as well as further removal of many links associated to a given hub. From c to d a duplication of the hub involving many incoming links has taken place (together with some further node addition). From d through i, it is evident how a large number of new classes were added by copying the pattern of single nodes connected to two central hubs. There is extensive rewiring in some stages, such as in f, where the lower hub loses a large fraction of in-links. Moreover there is also addition of new connections between existing nodes (see h → i). The whole sequence spans one year of development. The main observation from this example (which is a typical one) is that node duplication plus rewiring, particularly link removal, are widespread. This is also the case in the evolution of cellular networks [6].

Examples like the previous one suggest that duplication-divergence growth is the cause of observed subgraphs abundances in software maps. This hypothesis can be tested by comparing the distribution of subgraphs in real networks with those obtained with a stochastic model of network growth based in asymmetric duplication-divergence rules, previously described in [6]. First, an initial random (or backbone) network of \( n_0 < N \) nodes is created. This random graph is generated by the addition of nodes with degree \( k_0 = 2 \), every link pointing to a random target node [28]. This backbone posse a tree-like structure (as it occurs with software maps at the beginning of their evolution). Starting from this backbone, we apply the following rules at each iteration of the model:

1. **Duplication.** A randomly chosen target node \( v \) is cloned, and the new node \( w \) attaches to all the neighbors of the target node.

2. **Divergence.** For each pair of original/redundant links remove one of them with probability \( \delta \).

3. **Cross-linking.** In addition, the target and new node are linked \((w \rightarrow v)\) with probability \( \beta \). This rule is important in order to generate triads or 3-subgraphs.

In spite of the simple set of rules implicit in the duplication model, the frequency of subgraphs obtained from our *in silico* system are remarkably close to those seen in their
thetic networks are obtained, we perform a 4-subgraph parameter space is sampled uniformly. Once the syn-
census by counting the number of appearances of each 4-
subgraph.

In order to compare the two systems, the raw number of subgraphs of size four are computed and their concentration in the model and in the synthetic network.

Notice that we do not test for statistical signiﬁcance (as in the motif analysis). Instead, our comparison test is based solely in raw subgraph counts. In order to compare the two systems, the raw number of subgraphs of size four are computed and their concentration C evaluated. Here, the concentration is simply the number of appearances of the 4-subgraph over the total number of 4-subgraphs found.

In Figure 7, each point represents the pair \((C_{\text{predicted}}, C_{\text{observed}})\) of observed and predicted concentrations for given 4-subgraph \(\Omega_i\). Speciﬁcally, we display the set of pairs and the power law ﬁt \(C_{\text{pred}} \sim C_{\text{obs}}^\xi\). Despite ﬂuctuations, the simple duplication model presented here predicts reasonably well the concentration of common software network motifs: the value of the exponent \(\xi\) is reasonably close to one in all cases. This is remarkable, given the oversimpliﬁcation considered here and given the limited constraints imposed to the selected model graphs to be compared with the real ones. The error bars grow as less common subgraphs are used.

If we restrict to \(C > 10^{-3}\), the exponent \(\xi\) becomes much closer to one. Speciﬁcally, we obtain now (a) \(\xi = 0.96 \pm 0.11\), (b) \(\xi = 0.97 \pm 0.10\), (c) \(\xi = 0.98 \pm 0.12\) and (d) \(\xi = 1.06 \pm 0.18\), respectively. Consistently with previous work [8], less common subgraphs are typically more dense (have more links). In Figure 8 an example of this correlation is shown for Exult. Using a frequency-rank plot of 4-subgraphs, we can see that subgraphs with high frequencies have few links whereas higher ranks (small frequencies) are associated to dense subgraphs.

V. DISCUSSION

In this paper we have analysed the statistical patterns of network motifs in a large set of software diagrams. Software maps have been previously shown to be scale free and display small world behavior [14, 20, 22] but no previous analyses focused on the small-scale architecture. The main goal of our study is to explore the relevance of graph evolution in relation with true functionality. Our study actually suggests that dynamical rules, with little relation to underlying functional constraints, largely determine the frequency of motifs in software graphs.

By using recent theoretical and numerical methods to measure and characterize network motifs, we have found that:

1. A number of network motifs are obtained, being the most common shared with other (natural) systems involving computational traits, such as genetic and neural networks.

2. The number of appearances of a given network motif scales as \(\langle G\rangle \sim N^{a-g^{t+\gamma^s+1}}\), in agreement with previous calculations for random graphs with scale-free degree distributions. This result is supported

FIG. 7: Comparison of observed and predicted (from a duplication model) 4-motif concentrations for (a) Blender, (b) Filezilla, (c) gtk and (d) exult (here concentrations are rescaled by \(\times 10^{-3}\)). The exponents for the least squares ﬁt are: (a) \(\xi = 0.94 \pm 0.12\), (b) \(\xi = 0.92 \pm 0.13\), (c) \(\xi = 0.96 \pm 0.11\) and (d) \(\xi = 1.14 \pm 0.12\), respectively.

FIG. 8: Frequency-rank distribution of network subgraphs in a software network ( ). Here the most frequent subgraph has rank one, the second has rank two, etc. The frequency \(P(r)\) of a subgraph with rank \(r\) decays rapidly with subgraph rank. An interesting feature is that most common subgraphs are sparser than less common ones, which are more dense.
by previous observations of the uncorrelated character of software maps.

3. Evidence from software evolution suggests that duplication and rewiring, as it occurs with some cellular networks, might play a key role in shaping software maps. Using a previous model of network growth by duplication and diversification, it has been shown that it fits rather well the frequencies of appearance of network motifs.

Previous studies have proposed the idea that network motifs seem to define the minimal, meaningful building blocks of network complexity. Perhaps not surprisingly we often find them as the basic structures associated to specific functional traits, from computation to pattern formation. The former is exemplified by feed-forward loops, a three-element motif found in genetic regulatory systems [29, 30]. The latter is actually a particularly relevant example. However, since the statistical distribution of network motifs involves dealing with large numbers of different subgraphs, the question of how motifs in general might reflect functional traits requires the formation of appropriate null models of graph evolution. Such models must ignore any functional trait in order to test the possibility that global properties of network structure (such as graph heterogeneity) might strongly influence what we should expect.

The model chosen here has been a duplication-rewiring one [6]. These models have been shown to generate heterogeneous graphs with many properties close to relevant biological systems such as protein-protein interaction maps. Network heterogeneity is largely due to effective preferential attachment. Additionally, the rules of duplication strongly bias the types of motifs to be formed towards some special subsets. The final consequence is that the patterns of network motifs generated by the duplication model might be able to explain (in statistical terms) the observed abundances of motifs, with no further requirement of functional constraints. The fact that biological systems, also involved in performing computations, have common motifs might support this view. Although sharing common motifs seems to call for common functionalities, it is important to remember that biological structures are largely generated through tinkering [10, 11]. Protein interaction networks grow by gene duplication and neural networks also experience increases of cell numbers together with wide synaptic changes. Perhaps the common traits are a byproduct of the common tinkered evolution based on extensive reuse and copy of available structures.

One final comment concerns with the common subgraphs also shared by digital circuits. They are not obtained, strictly speaking, through a process of duplication and rewiring. Although the way complex circuits are built does include some amount of reuse [31] considerations involving low cost in links are of fundamental importance. In spite of such constraints, it has been shown that electronic circuits have small world structure and are also highly heterogeneous [32]. Previous work seems to indicate that optimal design towards efficient communication at low cost can generate scale-free, heterogeneous architectures [33, 34]. Such result suggests again that network heterogeneity might pervade motif abundances.

Acknowledgments

The authors thank Shalev Itzkovitz for a careful reading and comments on an earlier version of the manuscript. The analysis of network motifs has been done using available free software from Uri Alon’s Lab (see http://www.weizmann.ac.il/mcb/UriAlon/index.html). This work has been supported by grants BFM2001-2154 and by the EU within the 6th Framework Program under contract 001907 (DELIS) and by the Santa Fe Institute.

All software maps analysed here (and others studied by other authors) have been shown to be sparse. Correlations have been also analysed in Solé, R. V. and Valverde, S., in: Complex Networks, E. Ben-Naim, H. Frauenfelder, and Z. Toroczkai (eds.). Lecture Notes in Physics, pp. 169-190. Springer, Berlin (2004). Using statistical measures derived from information theory, it was shown that software maps are considerably uncorrelated.

The source code is available at the following web sites: http://www.blender.org (Blender), http://filezilla.sourceforge.net (Filezilla), http://www.gtk.org (GTK) and http://exult.sourceforge.net (Exult).


[31] As circuit complexity increases (both in terms of number of components and computational tasks) it becomes more difficult to design from scratch choosing sets of small gates and building optimal, low-cost circuits. Pre-defined gates involving well-known (and sometimes complex) input-output functions are widely used and assembled together. In that sense, some amount of re-use is at work.

