Graph Grammars for Super Mario Bros * Levels

Santiago Londoño
University of Bonn
londono@cs.uni-bonn.de

Olana Missura
University of Bonn
olana.missura@uni-bonn.de

ABSTRACT

We assume that the structure of a level in a platformer game has a profound influence on the enjoyment of players. Another assumption is that, given levels created by expert designers, it is possible to extract and transfer their structural properties to new levels. To make it an automatic process, in this paper we first propose a graph-based representation of *Super Mario Bros* levels to encode their structure. Next, to abstract the structural elements we extend an algorithm for learning a graph grammar, SubdueGL, to produce a stochastic graph grammar. Then we describe our work in progress on generating new levels from graphs produced by the graph grammar.

Keywords

Stochastic graph grammars, graph data mining, game level generation, procedural content generation

1. INTRODUCTION

In side-scrolling games such as *Super Mario Bros*, the player interacts with a simplified, two-dimensional world made up of elements from a finite set. Conventionally, specialized designers build the levels of the game, applying their creativity and experience to produce content exhibiting structural correctness (the player must be able to traverse the level from start to end), interestingness for the player, and balanced difficulty among other quality features.

Professional designers are able to consistently construct high quality game levels. In other words, their expert knowledge gets embedded in the resulting levels. We aim to create a learning mechanism capable of extracting design concepts from a set of human-authored levels and to use the acquired knowledge to algorithmically generate new ones. To that purpose we propose to use stochastic graph grammars in their both qualities, descriptive and generative. That is, first, given a set of levels, we will learn a grammar that encodes and abstracts structural properties presented in these levels together with their respective probabilities of occurrence. Next, the inferred grammar will be used to generate new levels possessing the same properties.

*Specifically, our application is aimed at generating levels for the framework *Infinite Mario Bros*, a clone of the classic *Super Mario World* originally published by Nintendo.

2. RELATED WORK

PCG techniques have been used to automatically generate content for computer games since the early eighties [5]. Although the types of generated content, techniques and algorithms employed are quite diverse, most of these applications have the same general objective: to *autonomously* create elements, such as missions [4], spaces [6], textures [16] and levels. The last one in particular, has been amply explored through a variety of approaches.

Search-based techniques, such as evolutionary algorithms, genetic programming and other forms of stochastic optimization have been employed in multiple ways. For instance, Cardamone et al. [1] made use of a search-based, evolutionary approach to generate FPS[1] levels, which were represented as grids. Johnson [6] implemented a two-dimensional cellular automaton that gives form to cave levels, also in grid configuration, by randomly carving tunnels on an initial block.

The specific case of level generation for *Super Mario Bros* has also been explored. For instance, in [14] levels were abstracted as matrices and Markov Chains were employed to randomly generate new content. Although the matrix representation implicitly contains sound information about the elements of a level and their relations, it does so in a way that is too complex and expensive to analyze via high-order Markov chains. This is a significant hindrance, as these relations are major determinants of the structure and quality of the level.

The PCG community starts to recognise the usefulness of grammars in representing and generating levels. Two notable examples are presented by Dormans [4] and Shaker et al. [12]. Dormans [4] builds level layouts for adventure games over mission scripts, based on context-free and space grammars. Shaker et al. [12] generate *Super Mario Bros* levels. The authors use genetic programming to search the space of levels derivable from a predefined grammar in a combined approach called Grammatical Evolution [12].

[1]Acronym for the genre First-Person Shooter.

Smith et al. [13] developed *Tanagra*, a tool aimed at assisting level design for platformer games. They represent levels as sequences of predefined patterns, composed of a set of basic elements of the game. These patterns are assembled together through a rhythm-based approach, under the direction of the designer.

The work of Smith et al. is based on the abstraction proposed by Compton and Mateas [2], which models levels by means of context-free string grammars. The grammars define how a set of patterns of varied complexity, can be put together to give form to a complete level. These patterns represent optimal sequences of basic game components, automatically built through a hill-climbing algorithm. The PCG process is graph-grammar driven and in principle similar to our approach, as an initial graph is recursively expanded to form a new level, following the grammar rules. However, both of the aforementioned approaches assume that the grammars are provided and no mechanisms to automatically learn them are discussed.

In this work we go a couple of steps further by using graph grammars to encode the relationships between various elements of a level and by learning these grammars instead of requiring them to be predefined.

To the best of our knowledge, graph grammars have not been applied as yet to algorithmically generate *Super Mario Bros* levels. Nevertheless, they have shown promising results in diverse fields. Zhao et al. [17] proposed a graph grammar based approach to discover patterns in the execution of programs. Cook and Holder [3] have also experimented with the application of graph grammar learning to the discovery of interesting structures in chemical compounds.

Müller et al. [8] developed *CGA Shape*, a computer graphics model of building architectures. It consists of context-sensitive shape grammars inspired by Lindenmayer Systems, whose rules define the spatial structure of the shapes that compose a building. The rules of the grammars describe sequences of geometric transformations that applied to shapes and volumes, can be used to procedurally generate the structure and external details of a building.

CGA Shape was successfully tested by modeling buildings, neighborhoods and even ancient cities and demonstrated to be of great value for computer games. This intent of this approach is very similar to ours and shape grammars are closely related to graph grammars. However, the work of Müller et al. does not aim at autonomously learning the shape grammars, a task for which we deem graph grammars more appropriate, as they allow graph data mining techniques to be applied.

3. REPRESENTATION OF LEVELS

From an implementation perspective, *Super Mario Bros* levels are represented as a matrix of bytes, with dimensions $width \times height$. Each entry corresponds to a *sprite*, defined here as an atomic game element (e.g. a brick block, power-up, coin, etc.). In this paper, we will refer to the matrix of a level as its *grid* and denote it by M.

In contrast, from an analytical point of view, *Super Mario Bros* levels are composed of elements that are not isolated, but relate to each other in different ways. Certainly, the matrix representation of a level implicitly contains all the information about its elements and their interrelations, but this information is encoded in a convoluted manner, making access to it a daunting task. An explicit and succinct representation of the levels is fundamental for the development of effective algorithms for learning and procedurally generating content. For instance, the ability to model the relationships among level components is crucial to generate correct results. Hence, we suggest to use graphs as a representation of levels, where nodes are various elements and edges encode their relationships.

3.1 Levels as graphs

We represent *Super Mario Bros* levels as *directed graphs* $G = \langle V, E \rangle$, with V a set of *nodes* and $E \subseteq V \times V$ a set of *edges*. Both nodes and edges have labels in the sets Δ and Δ_E respectively. The set Δ is made up of the names of all the game elements of interest, such as coins, power-up boxes, rocks, pipes, brick blocks, etc. The set Δ_E comprises the relationships between these elements, as explained below.

3.1.1 Relationships among elements

One of the most significant advantages of using graphs, is that they enable us to establish semantic relationships between the elements of the levels. Specifically, edges allow to connect several elements and to give a meaning to those connections. We use this capability to classify all level elements of *Super Mario Bros* in various types, according to their interaction with the player. In the current stage of the project we have defined two types, which we consider major determinants of the structure of levels:

Platforms are groups of solid sprites (e.g. brick blocks, rocks, pipes) that are consecutive and lie on the same row of the level matrix. They form solid positions upon which the character can stand and whence he can reach other elements.

In a level graph, platforms are represented by edges, labeled as *platform* and connecting two nodes, respectively the first and last sprite of the actual platform. Both nodes must have labels corresponding to solid sprites. In addition, platform edges have a *length* attribute, stating the number of sprites it spans.

Along these lines, we denote platforms by $p = (p_s, p_t) \in V \times V$, where p_s and p_t are the start and end nodes. It can be the case that p_s and p_t are the same node, which entails that the platform consists of a single sprite.

Item clusters are subgraphs, denoted by $C = \{c_1, ..., c_n\} \subseteq V$. They represent conglomerates of sprites that are not solid, but still interact with the player (e.g. coins, power-ups, enemies). If the cluster contains more than one sprite, each one of them has at least one neighbor. That is, another sprite in the same cluster located at a distance lower than an established threshold.

Item clusters are represented within level graphs, by sets of nodes labeled as non-solid sprites. Each of these nodes is connected with its closest neighbor, through an edge labeled as *cluster*. Note that the definition above allows for single-node clusters.

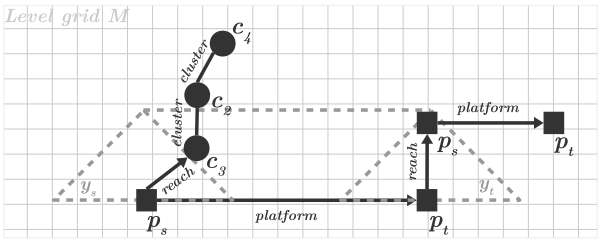


Figure 1: Reachability area of a platform, delimited by the dashed lines. The triangles y_s and y_t depict the reachability pyramids of the start and end nodes.

Additional types of elements may be devised in future iterations of the project, after we get our first experimental results. These new types could be used to represent groups of enemies at their starting positions. Regarding dynamic elements (i. e. sprites with changing positions) such as enemies or moving platforms, we consider that they could be modeled as nodes (or groups of nodes connected by edges) representing their starting positions. The parameters governing the movement, could be set as attributes of these nodes.

The concept of *reachability*, fundamental to our approach, is described in terms of a relationship between nodes.

Let $G = \langle V, E \rangle$ be a graph representing a level grid M and $p_x \in V$ a node that is part of a platform. We define the ***reachability pyramid*** of p_x, as the isosceles triangle having p_x at the center of its base and spanning the area of M directly above it.

Now, let $v \in V$ be a node and $p = (p_s, p_t) \in V \times V$ a platform. The node v is said to be ***reachable*** from p, if and only if the position of v in M is contained in the reachability pyramid of p_s or p_t, or the rectangle delimited by the top of these pyramids and by p_s and p_t. Figure 1 illustrates these ideas.

Finally, let $p = (p_s, p_t), q = (q_s, q_t) \in V \times V$ be two platforms. We say that q is reachable from p, if and only if either (or both), q_s and q_t are reachable from p.

Regarding the graph abstraction, reachability relationships between a platform p and other platforms or item clusters, are represented as edges with label *reach*, connecting the node of p and of the reached element that are closest to each other.

The importance of reachability edges arises from the fact that they are a key tool to guarantee the playability of the levels. In the particular case of *Super Mario Bros* (more precisely, on the version of the game our work is based upon), a level is playable if and only if the player is able reach the goal starting at his initial position. In turn, the latter condition can be guaranteed by properly defining the productions of the graph grammar and the initial graph to be used on the level generation process.

3.2 Transformation of level grids into graphs

In order to obtain the graph representation of the example levels we will learn from, we implemented an algorithm to transform a level in matrix form M (i. e. grid), into a graph G. The structure of the resulting graph is a combination of interconnected platforms and item clusters.

The algorithm consists of two major stages:

1. **Detection of elements:** platforms and item clusters are detected by scanning the level grid M row by row.

   Platforms are built by grouping adjacent sprites, found at the same row and that are suitable to be part of a platform.

   Item clusters are constructed through a simplified version of the *GDBSCAN* algorithm [11]. They are built up starting at a non-solid, interactive sprite, then expanding the cluster to its closest neighbor and repeating the process therefrom, until no further expansion is possible.

2. **Assembly of reachability edges:** once all platforms and item clusters in the map have been constructed, the algorithm evaluates what elements are reachable from what platforms and inserts *reach* labeled edges accordingly.

   This is performed by iterating over all the platforms previously identified. For each platform, the algorithm computes the reachability pyramid of its start and end nodes, defined as in section 3.1.1.

   It then finds all other nodes whose positions on the grid are contained in either, the reachability pyramid of the start or end node, or the area comprehended between them (as shown in figure 1) and regards them as reachable from the platform.

   Finally, for each one of them, the algorithm computes their distance to the start and end nodes of the platform and connects them to the closest one via a reach-labeled edge.

As the result, the transformation returns a graph representation of the level, as required by the learning algorithm.

Level grids are built from graphs in two stages: First, the main structure of the level is constructed, by rendering the platforms and item clusters on the grid. Second, adornments such as background sprites are randomly added around the main structure.

4. LEARNING FROM LEVEL GRAPHS

We hypothesize that human-authored, high-quality levels contain design paradigms reflecting the knowledge of the designers who created them. Furthermore, we assume that this knowledge is encoded in a way representable by graphs.

Under these assumptions, graph data mining techniques can be applied to extract the knowledge embedded in a set of human-authored levels. Specifically, we implemented the *SubdueGL* algorithm [7]. SubdueGL is based on the *Subdue* algorithm originally proposed by Cook and Holder [3] for the discovery of frequent substructures in graphs. A *substructure* is a subgraph that can occur one or more times

in the input graph. Given a set of graphs (human-authored levels in our case), SubdueGL learns a context-free graph grammar, striving to achieve an optimal balance between the size of the substructures and their frequency of occurrence. This criterion is called the *MDL principle* [3]. The induced graph grammars are expected to define patterns frequently observed in high-quality levels.

4.1 Node Label Controlled graph grammars

SubdueGL learns Node Label Controlled (NLC) graph grammars [10]. An NLC graph grammar R is context-free and has the form:

$$R = (\Sigma, \Delta, P, C, S) \ ,$$

where Σ is the alphabet of terminal and non-terminal node labels, Δ is the alphabet of terminal node labels, P is the set of productions, C is the set of connection relations, S is the initial graph. A single production from P has the form $z \rightarrow \alpha$, with $z \in \Sigma \setminus \Delta$ a non-terminal node label and α a graph, i. e. it defines how a node with a non-terminal label can be replaced by a subgraph α. A connection relation is an ordered pair $(s, t) \in \Delta \times \Delta$.

The *node label controlled* aspect of R, implies that α would be connected to the neighborhood of the replaced node by means of a connecting mechanism [10]. For a production rule $z \rightarrow \alpha$ it operates by adding an edge from a node labeled s in the neighborhood of z, to a node labeled t in α, for each *connection instruction* $(s, t) \in C$.

As aforesaid, SubdueGL discovers the set of best substructures in the input graph, with respect to the MDL principle. For each substructure α_i in this set, a production $P = z_i \rightarrow \alpha_i$ is added to the resulting NLC graph grammar.

4.2 Stochastic NLC graph grammars

To accurately represent the occurrences of found substructures, we extend SubdueGL to learn stochastic NLC graph grammars (SNLCGGs). Figure 2 illustrates the structure of these graph grammars as learned by our algorithm.

An SNLCGGs G, has probabilities
$p_{z,1}, ..., p_{z,n_z}$ associated to its productions $z \rightarrow \alpha_1, ..., z \rightarrow \alpha_{n_z}$, in such a way that for a particular left-hand side z, $\sum_{j=1}^{n_z} p_{z,j} = 1$. Hence, during the learning a "probability of being applied" has to be estimated for each discovered substructure and associated to the derived productions.

As proposed by Oates et al. in [9], a maximum-likelihood estimate for the probability of application of a production $z \rightarrow \alpha$, provided a set of example graphs $\mathbf{G_X} = \{G_1, ..., G_m\}$, can be computed as:

$$\hat{p}\,(z \rightarrow \alpha) = \frac{c\,(z \rightarrow \alpha | \mathbf{G_X})}{\sum_{\delta} c\,(z \rightarrow \delta | \mathbf{G_X})}$$

We employ SubdueGL to infer an NLC graph grammar from the set of example graphs $\mathbf{G_X}$. On each iteration, SubdueGL discovers frequent substructures and synthesizes them as right-hand sides of productions $z \rightarrow \alpha$. Thus, the total number of times a production was observed when building the graph grammar, namely $c(z \rightarrow \alpha | \mathbf{G_X})$, can be computed as a byproduct of SubdueGL. Suppose that SubdueGL is run

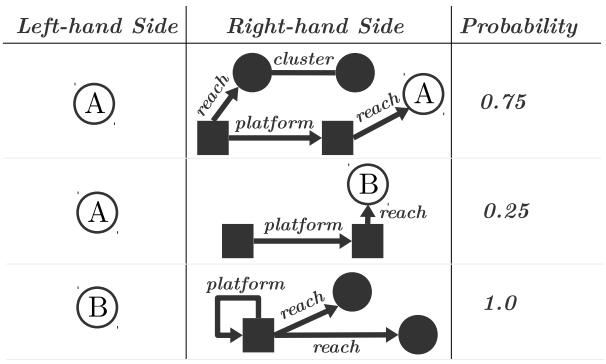


Figure 2: A stochastic graph grammar.

on a source graph $G_m \in \mathbf{G_X}$ and as the result, a set of substructures $\mathbf{B} = \{\alpha_i : i \in [1, k] \wedge \alpha_i \subseteq G_m\}$ are chosen to become productions. Each substructure appears $c\,(\alpha_i)$ times in the input graph. The resulting productions are:

$$P = \{z \rightarrow \alpha_i : i \in [1, k] \wedge \alpha_i \subseteq G_m\}, \text{ with } z \text{ non-terminal}$$

Their associated probabilities are computed as:

$$\hat{p}\,(z \rightarrow \alpha_i) = \frac{c\,(\alpha_i)}{\sum\limits_{\delta \in \mathbf{B}} c\,(\delta)}$$

Note that, it also holds $\sum_{\delta \in \mathbf{B}} \hat{p}\,(x \rightarrow \delta) = 1$.

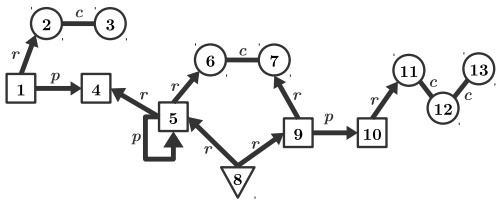
As an illustrative example, figure 3 depicts the SNLCGG inference process as performed by our implementation of SubdueGL. The input consists of an example graph, representing a simple level composed of platforms of two types of tiles (square and triangle nodes) and of some clusters of coins (circle nodes).

As aforementioned, SubdueGL strives to find substructures that are frequent and as large as possible within the example graph. It starts by taking each different node label appearing in the example graph as candidate substructure. This is shown at step *b)* on figure 3. Note that a substructure has a schema, which is simply a subgraph where the nodes have no particular identifiers, only labels and the set of occurrences of that very schema in the input graph, called instances.

Then, the instances of each substructure are extended one edge at a time, in order to discover a new set of larger substructures (step *c)*). The compressing power of each substructure is computed as a measure of how much would the size of the input graph be reduced, should all instances of the substructure be replaced for a single node (a non-terminal). In this example, such measure was computed as the number of edges and nodes in the schema, times its number of instances and was called *compression*. SubdueGL aims at compressing the input graph as much as possible (i. e. follows the MDL principle), thus it selects the substructures with the top three compression values to be extended at the next step.

The extension process is repeated on the best three substructures, until no instances can be extended further or until the maximum number of extensions (provided as parameter) has

a) Example graph



b) Substructures after initialization

Schema	Instances	Compression
○	2 3 6 7 11 12 13	7
□	1 4 5 9 10	5
▽	8	1

c) Top 3 substructures after first expansion

Schema	Instances	Compression
r	1 r 2 5 r 6 9 r 7 10 r 11	12
c	2 c 3 6 c 7 11 c 12 12 c 13	12
p	1 p 4 9 p 10 5 p 5	9

d) Top 3 substructures after second expansion

Schema	Instances	Compression
r c	1 r 2 c 3, 5 r 6 c 7, 9 r 7 c 6, 10 r 11 c 12	20
r p	1 r 2 p 4, 5 r 6 p 5, 9 r 7 p 10	15
c	2 c 3 6 c 7 11 c 12 12 c 13	12

e) Resultant Stochastic NLC Graph Grammar

Left-hand Side	Right-hand Side	Probability
L	r c	4/11 = **0.364**
L	r p	3/11 = **0.273**
L	c	4/11 = **0.364**

Figure 3: Simplified run of SubdueGL and the resultant SNLCGG.

been reached.

At the end, the best three substructures will become the productions resultant of this iteration of the inference process. Each schema is set as right-hand side of the productions and a new non-terminal is generated as left-hand side. The probabilities of applying each production are computed as specified in section 4.2.

5. LEVEL GENERATION

The level generation algorithm will first create a new graph using the learned grammar: It will expand an initial graph (e.g. a non-terminal node representing the start of the level) by iteratively applying the productions of the stochastic NLC graph grammar. On each iteration, the algorithm expands a non-terminal node with label z. To do so, it selects a production $z \rightarrow \alpha$, in accordance with the probabilities having z at the left-hand side. We expect the resultant generated graphs, to be structurally similar to the examples from which they were derived. In the next step, the so generated graph is transformed into a level grid and additional details such as background sprites will be rendered onto it.

6. CURRENT STATUS OF THE PROJECT

We have implemented the algorithm to transform level grids (matrices of bytes) into graphs as described in section 3.2. The stochastic version of SubdueGL, as specified in section 4, is also implemented and SNLCGGs are now being generated. The level generation process is yet to be implemented. This includes the algorithm we will use to expand random graphs from graph grammars, the transformation of a level graph into an equivalent grid and the rendering of details.

We plan to perform an evaluation to verify whether the generated levels preserve the quality features of the examples and at the same time, are novel enough to amuse human players. To do so, we intend to use metrics proposed by Shaker et al. in [12] to compare the levels used as training instances with the newly generated levels. Furthermore, we would design an experiment involving human players to qualitatively evaluate how enjoyable are our results and to compare our system against existing level generators for *Infinite Mario Bros*, namely (*i*) the *default generator* included as part of the framework, as developed by Markus Persson; (*ii*) the *GE generator* by Shaker et al. [12], based on grammar evolution; (*iii*) the *winner of the Mario AI Championship 2012* [15], an approach driven by the player's score, as presented by Chen et al. A set of levels will be generated through each system and split into random groups. Each participant will get assigned a different group to be played and at the end of the trial, they will be asked to rank the levels as to their preference.

7. REFERENCES

[1] L. Cardamone, G. N. Yannakakis, J. Togelius, and P. L. Lanzi. Evolving interesting maps for a first person shooter. In *Applications of Evolutionary Computation*, pages 63–72. Springer, 2011.

[2] K. Compton and M. Mateas. Procedural level design for platform games. In *AIIDE*, pages 109–111, 2006.

[3] D. J. Cook and L. B. Holder. Substructure discovery using minimum description length and background

knowledge. *Journal of Artificial Intelligence Research*, pages 231–255, 1994.

[4] J. Dormans. Adventures in level design: generating missions and spaces for action adventure games. In *Proceedings of the 2010 workshop on procedural content generation in games*, page 1. ACM, 2010.

[5] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup. Procedural content generation for games: A survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)*, 9(1):1, 2013.

[6] L. Johnson, G. N. Yannakakis, and J. Togelius. Cellular automata for real-time generation of infinite cave levels. *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 1–4, 2010.

[7] I. Jonyer, L. B. Holder, and D. J. Cook. MDL-based context-free graph grammar induction and applications. *International Journal on Artificial Intelligence Tools*, 13(01):65–79, 2004.

[8] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. Van Gool. *Procedural modeling of buildings*, volume 25. ACM, 2006.

[9] T. Oates, S. Doshi, and F. Huang. Estimating maximum likelihood parameters for stochastic context-free graph grammars. In *Inductive Logic Programming*, pages 281–298. Springer, 2003.

[10] G. Rozenberg and H. Ehrig. *Handbook of graph grammars and computing by graph transformation*, volume 1. World scientific Singapore, 1999.

[11] J. Sander, M. Ester, H.-P. Kriegel, and X. Xu. Density-based clustering in spatial databases: The algorithm GDBSCAN and its applications. *Data mining and knowledge discovery*, 2(2):169–194, 1998.

[12] N. Shaker, M. Nicolau, G. N. Yannakakis, J. Togelius, and M. O'Neill. Evolving levels for Super Mario Bros using grammatical evolution. In *Proceedings of the IEEE Conference on Computational Intelligence and Games*, pages 304–311. IEEE, 2012.

[13] G. Smith, J. Whitehead, and M. Mateas. Tanagra: A mixed-initiative level design tool. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, pages 209–216. ACM, 2010.

[14] S. Snodgrass and S. Ontañón. Experiments in map generation using Markov chains. In *Proceedings of the 9th International Conference on Foundations of Digital Games*, volume 14, 2014.

[15] J. Togelius, N. Shaker, S. Karakovskiy, and G. N. Yannakakis. The Mario AI championship 2009-2012. *AI Magazine*, 34(3):89–92, 2013.

[16] D. Yoon and K.-J. Kim. 3D game model and texture generation using interactive genetic algorithm. In *Proceedings of the Workshop at SIGGRAPH Asia*, WASA '12, pages 53–58, New York, NY, USA, 2012. ACM.

[17] C. Zhao, K. Ates, J. Kong, and K. Zhang. Discovering program's behavioral patterns by inferring graph-grammars from execution traces. In *Proceedings of the 20th IEEE International Conference on Tools with Artificial Intelligence*, volume 2, pages 395–402. IEEE, 2008.