Honda Research Institute EU
Honda Research Institute Europe GmbH
https://www.honda-ri.de/

Consistent Modeling of Functional Dependencies along with World Knowledge

Sven Rebhan, Julian Eggert

2009

Preprint:

This is an accepted article published in Proceedings of the International Conference on Cognitive Information Systems Engineering (ICCISE). The final authenticated version is available online at: https://doi.org/[DOI not available]

Consistent Modeling of Functional Dependencies along with World Knowledge

Sven Rebhan and Julian Eggert

Abstract—In this paper we propose a method for visual systems to consistently represent functional dependencies between different visual routines along with relational short- and long-term knowledge about the world. Here the visual routines are bound to visual properties of objects stored in the memory of the system. Furthermore the functional dependencies between the visual routines are seen as a graph also belonging to the object's structure. This graph is parsed in the course of acquiring a visual property of an object to automatically resolve the dependencies of the bound visual routines. Using this representation, the system is able to dynamically rearrange the processing order while keeping its functionality. Furthermore, the system is able to estimate the overall costs of a certain action. We will also show that the system can efficiently use that structure to incorporate already acquired knowledge and thus reduce the computational demand.

Keywords—Adaptive systems, Knowledge representation, Machine vision, Systems engineering

I. Introduction

COGNITIVE vision systems, both technical and biological, with at least a minimal claim on generality have to carefully select the information they acquire from their environment. This is necessary to fulfill constraints on computing and memory resources. Therefore, those systems implement algorithms to focus on certain aspects of the surrounding scene, depending on their need, their task and their knowledge about the world they have accumulated. This flexible control architecture like proposed in [1] must be able to dynamically rearrange the processing pathways of the system, use the already acquired knowledge and estimate the cost and benefits of the system's actions. To achieve this in a reasonable manner the system not only needs knowledge about relations between objects, but also needs knowledge about the relations of internal routines it can use to acquire information about the vicinity. This knowledge could then be used to determine which actions the system has to perform to measure e.g. a certain property of an object. If, for example, the system wants to measure which color an object has, it first need to know where the object is and what retinal size it has approximately. Furthermore, determining the position of an object might involve further processing which is again a dependency of the localization module and so on. The structure we have chosen makes it possible to model those dependencies along with the world knowledge the system has in a relational memory. In this paper we concentrate on how we can efficiently implement the knowledge about dependencies between different routines and on how to use it in a system context.

In computer science problems similar to the representation of such dependencies exist. Those problems on representing the data flow of a computer program date back to the work of Dennis [2], [3]. In this and later works, graph structures are used to analyze the data and control flow of a computer program to parallelize and optimize the program by a compiler [4], [5]. There, the program dependence graph "[introduces] a partial ordering on the statements and predicates in the program that must be followed to preserve the semantics of the original program" [4, p. 322]. In the domain of computer vision, data flow graphs are also used to ease the design of vision systems and keep their complexity manageable [6].

However, all of the mentioned methods try to map a fixed and predefined algorithm to a graph structure to later use it for parallelization and optimization of that fixed algorithm. Contrary to that, we propose a method to implement an *on-demand* vision system that parses its internal representation of the dependencies and *dynamically creates* a program for acquiring the requested property of an object. As the vast majority of the literature in the field of computer science shows, graph structures are well suited for that purpose. In this paper we will show that:

- Using graph structures we are able to consistently model functional dependencies between object properties along with the property structure of objects and world knowledge, both short- and long-term.
- Using graph structures the system is provided with the means to estimate the costs of a certain measurement. In an example we show that the size of the graph required for performing the measurement can be used as a cost function.
- Using our proposed parsing algorithm, knowledge already acquired by the system can be reused in a simple and efficient way. This leads to a reduction of the computational demand and speeds-up operations of the system.
- Using our proposed parsing algorithm, the complexity of designing the vision system is considerably reduced by only modeling direct dependencies.

In the next we show the memory structure of the system together with the way we are modeling the functional dependencies. We will also elucidate modifiers required for covering the whole functionality of a vision system in the dependency structure. In section III we then present a parsing algorithm that exploits the previously described graph structure. We discuss some special situations we came across when working with that structures. Using the parsing algorithm presented in section III we perform some experiments in a proof-of-concept system based on the architecture proposed in [1] and finally discuss our results.

II. Relational Memory

Memory Structure

In our vision system we use the relational semantic memory proposed in [7] for representing information in the short- and long-term memory. This relational memory is, contrary to many other semantic memories, able to represent an arbitrary number of relation patterns. Thus we can define classical link types like ”hasProperty”, ”isPartOf” or ”isContainedIn” as shown in Fig. 1. Additionally we store a sensory representation

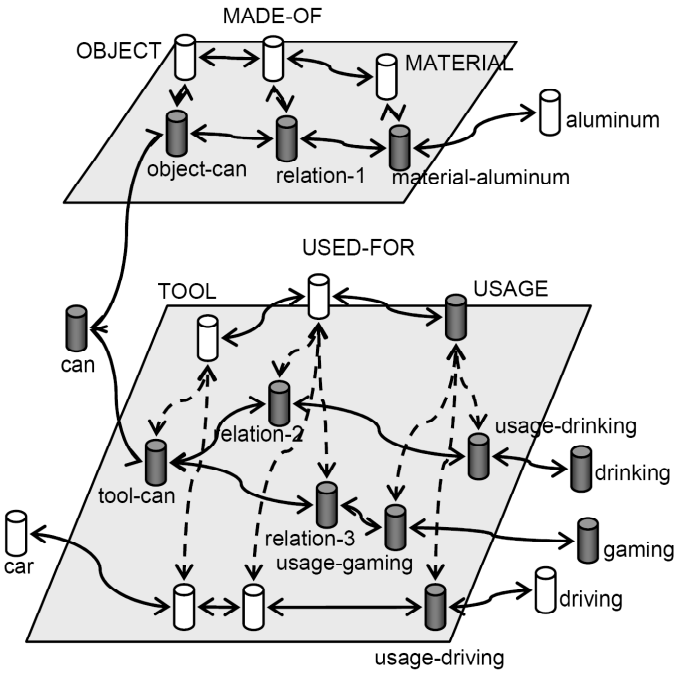

Fig. 1. In the relational memory we use, an arbitrary number of link patterns can be constructed. Some examples are shown here.

in the property nodes to be able to later feed back that information into the system. Along with the sensory representation a direct link to the visual routine used for acquiring a certain property is stored in the property nodes. Thus we can *demand* the attached visual routine to deliver information. The objects in the memory are composed of several visual properties.

Beside the classical link patterns, we can also construct dependency patterns. The dependency pattern that can be seen in Fig. 2 reads as ”the measurement of *A* depends on operation *op* of *B*. Given this link, we can now measure *A* in a demand-

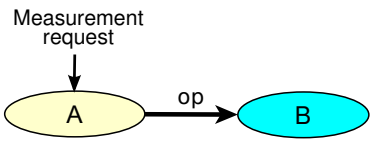

Fig. 2. The measurement of node *A* depends on the operation *op* of node *B*. For representing this we use exactly the same memory structure as shown in Fig. 1.

driven way: if the system needs to measure node *A* it knows it has to perform *op* of node *B* before being able to process *A*. The operation *op* of *B* has no further dependency in this example and can thus be performed directly. Afterwards *A* can be measured. If the structures getting more complex and the graph is getting deeper, a more sophisticated algorithm is needed to perform the parsing of the graph. Details on this can be found in section III.

Link Modifiers

Even though Ballance et. al state in their paper that ”neither switches nor control dependence are required for a demand driven interpretation” [8, p. 261], we need some modifiers for the dependency link patterns to cover interesting cases of a vision system. Those interesting cases are:

- The operation of node *B* is optional and not absolutely required for measuring node *A*, but would e.g. improve the result of the measurement. A modulatory input for a visual routine would be an example for this case, where the presence of such an input is helpful but not necessary (see Fig. 3 a).
- The system requires different operations for the target node to be fulfilled before it is able to process the current node (see Fig. 3 b).
- There might be alternative ways to measure a certain property and the system only needs to fulfill one of several dependencies. Think of different segmentation algorithms for estimating the shape of an object, where only one of those algorithms is required to complete to get a shape (see Fig. 3 c).

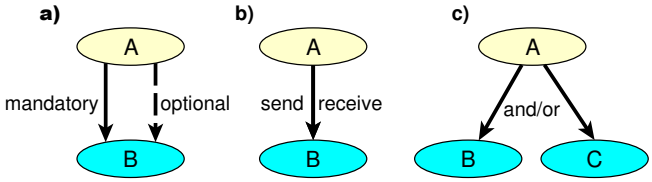

Fig. 3. We cover different cases using modifiers for the dependency link patterns: a) a dependency can be optional or mandatory, b) different operations(send and receive) are requested for the target node and c) we differentiate between the need for all dependencies or only one-of-many dependencies to be fulfilled.

The generic pattern we implement reads as ”*A* depends *dependency type* on *operation* of *B logical mode C* ...” The modifiers of this generic pattern are in our case:

- **Dependency type:** The link between the node can be **mandatory** or **optional** as shown in Fig. 3 a.
- **Operations:** We realize **send** and **receive** operations that push or pull information of the target node, respectively as shown in Fig. 3 b.
- **Logical mode:** The node ”*A* can depends on *B* **and** *C*” or node ”*A* can depend on *B* **or** *C*”. That way we can mark alternative pathways by using the logical **or** mode, else node *A* depends on all target nodes (see Fig. 3 c).

Node States

Beside the link patterns, the graph also contains nodes. In our case each node has a state marking the validity of the nodes

data. We use this later to determine if the node information needs to be update i.e. the visual routine bound to this node needs to be executed or not. Basically there are two states, as the data is either *valid* or *invalid*. In the beginning, all nodes contain invalid data. After updating, i.e. *receiving* information from a visual routine, the datum of the node is *valid*. The transition of the node's state back to *invalid* can be determined by time or any other criteria. In section III you will see how we use the state of the nodes to dynamically reduce the number of operations if we encounter a node with valid data.

System Memory Layout

After discussing the different link types, operation, modifiers and node states, we now present the actual designed prototypical memory patterns we use in out system. The upper part of Fig. 4 shows the view on the designed object structure. You can see that the object properties are bound to the different visual routines (shown in the upper left). In the lower part of Fig. 4 the designed dependency patterns are shown. Please note that the illustration shows only two different views on the memory content. Both representations coexists in the very same memory using the same nodes! As you can see we only define the *direct dependencies* of the node and not the whole tree. This eases the design process, as it keeps the system structure manageable. The complete dependency tree will later be generated on the fly using the parsing algorithm described in the next section.

III. Dependency Parsing

THE last section illustrated the way the system represents its knowledge, both about the world and about its internal functional dependencies. In this section we will show how we use that knowledge for implementing a demand-driven acquisition of sensory information about the vicinity. If we look back at the bottom of Fig. 4 in section II we see that we define only direct dependencies. To update a property of an object like its 3D-position (world location), we need to *resolve* the dependencies of that node. The manually resolved dependency graph for the world location is shown in Fig. 5 to illustrate the necessary steps.

Recursive Parsing

In the example of receiving the world location (see the steps in Fig. 5), this would require the measurement (receiving) of the retinal location (1) and the distance of the object, as we can calculate the 3D-position of the object by means of the camera parameter. However, the measurement of e.g. the retinal location itself depends on sending a spatial modulation map (2). If you take a closer look, you will see that the dependency is *optional*, as we can also measure the retinal location without having a modulatory input. If we again follow the graph, we see that the sending of the spatial modulation map itself depends on the acquisition (receiving) of the spatial modulation map (3). This makes perfect sense, as we first need to have the modulatory information before using it. Looking at the steps we have done up to here we already can see

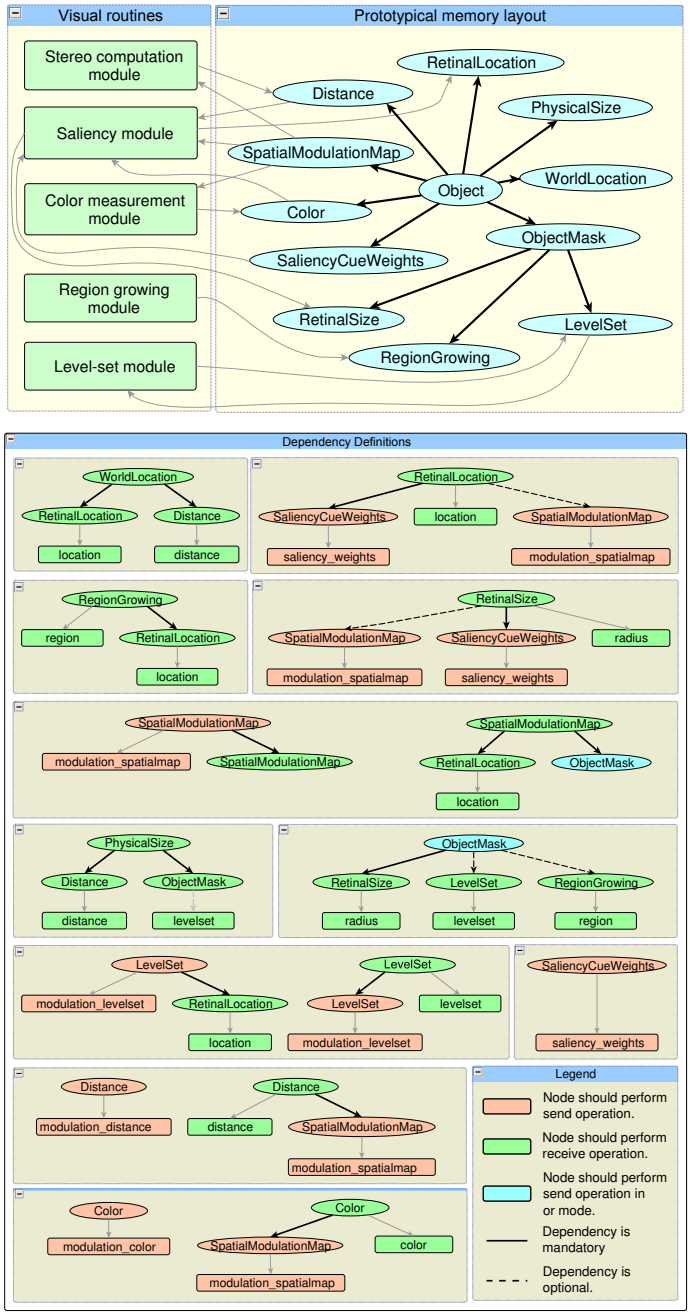

Fig. 4. On top the prototype of a object structure is displayed as used by our system. You can clearly see the binding to the visual routines, both feed-forward and feed-back. On the bottom, the dependency structure can be seen. The color indicates the operation the node should perform.

that parsing the dependency graph can be formulated as a recursive problem. So we implement our parsing algorithm as a recursive function as can be seen in the pseudo-code below.

Circular Dependencies Detection and Handling

We now continue the example and pursue the dependencies one step further. We find that the measurement of the spatial modulation map depends on the measurement of a object mask and on the measurement of the retinal location (4) of the object. This two information are necessary to create the spatial

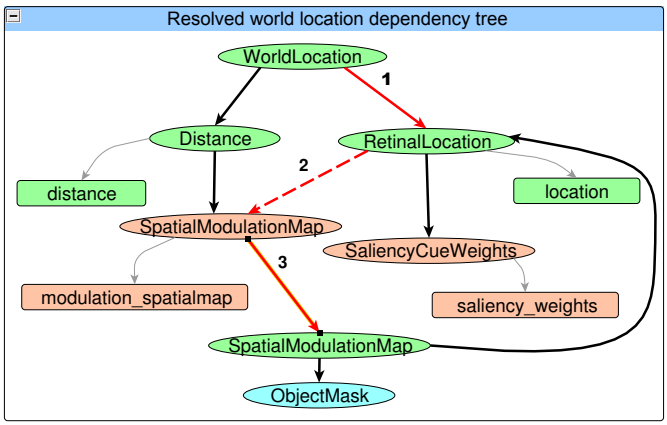

Fig. 5. The resolved dependency tree for the world location property of the object looks much more complex then the definitions in Fig. 4. We the didn't complete resolve the graph here for simplicity (cut at the object mask node). The path described in the text is marked red.

modulation map at the correct location with the correct shape. However, if we further trace the dependencies of that branch in Fig. 6, we see that the retinal location depends on the spatial modulation map, which in the end depends on the retinal location. What we see here is a loop or *circular dependency*,

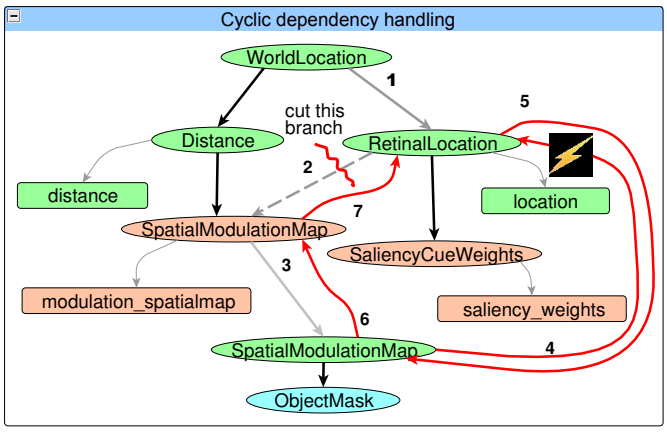

Fig. 6. The detection of a cyclic dependency at the retinal location node, leads to a trace back of the path. Finally the circular dependency can be resolved by cutting the branch between the retinal location and the spatial modulation map nodes.

which would lead to a dead-lock situation if the system doesn't have means to deal with it. So the first important point is to detect such circular dependencies which can be easily done by marking visited nodes in the graph and check if the node is already marked before entering it. The second important question is what to do once a circular dependency is detected. Here the dependency types described in section II come into play. After detecting the circular dependency, we go back to the parent node (5) and check if the dependency is mandatory or optional. If the dependency is optional we are done, as we can simply cut the loop at this point without breaking the algorithm. This is because the information that is missing, is not essential for the algorithm to run. However, if the dependency is mandatory (as in our example) the system can't resolve the dependencies of the current node. In this case, the system can't receive the spatial modulation map, because it requires the retinal location to be known. Thus the system needs to go back one step (6) and check if the operation of the dependency graph's parent can be executed (in our case the sending of the spatial modulation map). As you can see in Fig. 6 this is not the case, as sending the spatial modulation map strictly depends on receiving it first. Again we have have to trace back the dependency path one step (7). This brings us back to the receiving of the retinal location, which only optionally depends on sending the spatial modulation map. At this point we can "solve" the circular dependency by *cutting the complete branch* leading to the loop. The procedure for handling circular dependencies can be summarized as:

1) Detect a circular dependency.
2) If the current link leading to a dependency loop is optional, cut it and thus remove the whole branch containing the circular dependency.
3) Otherwise check if we are already at the root node. In this case, the dependencies can't be resolved and an error should be returned. If we are not yet at the root node, trace back the dependency path one step and continue with step 2.

This steps are also fold into the pseudo-code at the end of this section.

Reusing Already Acquired Knowledge

One of the biggest advantages of our approach to flexibly model functional dependencies is the fact that we can reuse the knowledge the system has. For doing so we introduced the *node state* in section II. This node state tells the graph parsing algorithm if a node requires updating, i.e. performing the operation required by its parent in the dependency graph, or if it already holds valid data. If the node already has valid data, the system does not need to execute the whole dependency sub-tree below the node. Let's assume that we already measured the retinal location (the data is still valid) and we now want to update the world location of that object. This will lead to the effective graph you can see in Fig. 7. If you compare that effective to the original one in Fig. 5, you see that the *effective structure of the dependency graph is determined by the knowledge of the system*! This is a main difference to previously proposed methods like [4], [8], [6], working only on fixed graphs. Eventually, the graph shrinks by incorporating the knowledge of the system, leading to a more efficient and less demanding system.

Alternative Pathways

Alternative ways of measuring a certain object property are desirable in a cognitive vision system, as the redundancy often increases the robustness of the system. This is because different algorithms for determining a property might differ in the assumptions they make on the data, the way they compute the result, the speed, the accuracy and the weakness they might have. Therefore we need to also add a way to deal with such alternative pathways. In our example shown in Fig. 8

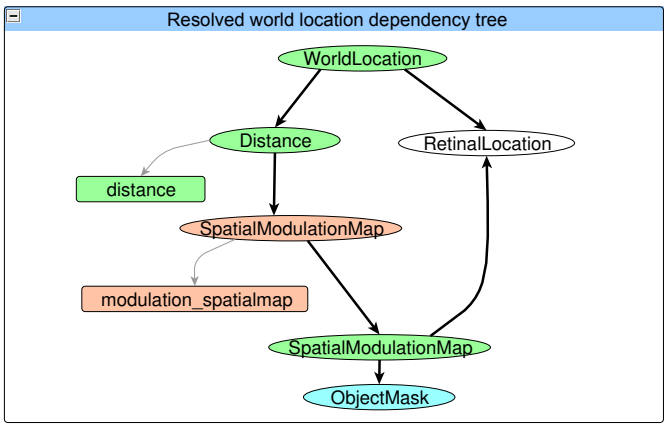

Fig. 7. By incorporating knowledge already acquired by the system, the structure of the effective dependency graph changes and its size shrinks.

we have three different segmentation algorithms: a simple size estimation using the saliency map (see [9] for details), a region-growing method (see [10]) and a level-set method (see [11]). For modeling alternative pathways we introduced

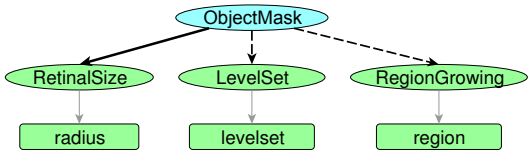

Fig. 8. The mask of an object can be calculated by any of the three algorithms (retinal size estimation using the saliency, region growing and a level-set method). However, only one of these *alternatives* has to run to get the mask.

the *logical or mode* in section II. As you can see in Fig. 8 the ”object mask” node is marked by a bluish color which indicates ”or nodes”. The *or mode* is interpreted by the graph parser as ”one of these dependencies is required”. To calculate the object mask we only need to start one of these routines. However, if you look at the different algorithm you will see that they differ in speed, initial requirements and accuracy. The retinal size estimation is *very fast* and *only needs the object's location* as an initial value, but is *not very accurate*. The region-growing is *fast* (even though it is slower than the retinal size estimation), *only needs the object's location* as an initial value and is *more accurate* at least for homogeneously structured objects. The level-set method on the other hand is *relatively slow* compared to the other two algorithms, *needs an initial segmentation* to start but is *very accurate* even for structured objects. One consequence of the mentioned properties is, that the level-set method never can be used for initially estimating the object mask, because it needs an initial mask to run. Furthermore, the system should be able to select the algorithm that is as accurate as required, but as fast as possible. What we need here is a decision dependent on the current system's state (e.g. required accuracy and available time) and the system's knowledge (e.g. initial object mask). In the easiest implementation the parser now tries to resolve the dependencies in order until one of them can be resolved [1]. If none of the dependencies can be resolved a trace back as described in the circular dependency case can be performed. Besides the resolvability of dependency an extended version of the parsing algorithm could take the costs and accuracy of the different pathways into account.

Pseudo-code

The pseudo-code of our graph parsing algorithm has a recursive nature as the problem is recursive. The algorithm dynamically generates the dependency graph starting from the requested property. It also needs to take care about the cyclic dependency detection and handling. The update procedure reads as follows:

Procedure *UpdateNodeValue*:

(a) Check run-ability of the node

(1) Check the current node for valid data. If it already has valid data we skip any operation and return success.
(2) Check for a cyclic dependency indicated by an already set visited flag. If we *detect a cyclic dependency*, hand the corresponding error to the node's parent.
(3) Set the visited flag for the current node.

(b) Updating dependencies

(1) Get the list with all dependencies for the current node.
(2) For each dependency (child node) do:
  (2.1) Call ***UpdateNodeValue*** on the child node.
  (2.2) Check the return code of the call for a cyclic dependency error. If we get such an error and we have a *mandatory dependency* for that child node, propagate the error further up to the node's parent. For an error on an *optional dependency* we continue with processing the next dependency in the list.
  (2.3) If we are on a *logical or* node we can leave the loop and continue with (c), because at least one dependency was fulfilled.

(c) Execute current node's operation

(1) Perform the *send or receive operation* of the current node as requested by the parent node and optionally store the sensor data locally.
(2) Set the data validity flag.
(3) Remove the visited flag.

REFERENCES

[1] Julian Eggert, Sven Rebhan, and Edgar Körner. First steps towards an intentional vision system. In *Proceedings of the 5th International Conference on Computer Vision Systems (ICVS)*, 2007.
[2] Jack B. Dennis. First version of a data flow procedure language. In *Proceedings of the Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 362–376, London, UK, 1974. Springer-Verlag.
[3] Jack B. Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, November 1980.
[4] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Language and Systems*, 9(3):319–349, July 1987.
[5] Robert Cartwright and Matthias Felleisen. The semantics of program dependence. In *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation*, pages 13–27, 1989.
[6] Per Andersson. Modelling and implementation of a vision system for embedded systems, 2003.
[7] Florian Röhrbein, Julian Eggert, and Edgar Köerner. Prototypical relations for cortex-inspired semantic representations. In *Proceedings of the 8th International Conference on Cognitive Modeling (ICCM)*, pages 307–312. Psychology Press, Taylor & Francis Group, 2007.

[1]Note that a node can deny its execution if its initial conditions are not fulfilled.

[8] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN 90 Conference on Programming Language Design and Implementation*, volume 25, pages 257–271, New York, NY, USA, 1990. ACM.

[9] Sven Rebhan, Florian Röhrbein, Julian Eggert, and Edgar Körner. Attention modulation using short- and long-term knowledge. In A. Gasteratos, M. Vincze, and J.K. Tsotsos, editors, *Proceeding of the 6th International Conference on Computer Vision Systems (ICVS)*, LNCS 5008, pages 151–160. Springer Verlag, 2008.

[10] Milan Sonka, Vaclav Hlavac, and Roger Boyle. *Image Processing, Analysis, and Machine Vision*. Thomson-Engineering, 2 edition, 1998.

[11] Daniel Weiler and Julian Eggert. Multi-dimensional histogram-based image segmentation. In *Proceedings of the 14th International Conference on Neural Information Processing (ICONIP)*, pages 963–972, 2007.