International Journal of Business, Human and Social Sciences
ISSN: 2517-9411
Vol:3, No:6, 2009

# Consistent Modeling of Functional Dependencies along with World Knowledge

Sven Rebhan, Nils Einecke and Julian Eggert

*Abstract*—In this paper we propose a method for vision 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. Additionally, the system is able to estimate the overall computational 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 its vicinity. This knowledge could then be used to determine which actions the system has to perform to measure a certain property of an object. If, for example, the system wants to measure which color an object is, it first needs to know where the object is and what retinal size it approximately has. 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 represent the knowledge about dependencies between different routines and on how to use it in a system context.

S. Rebhan, N. Einecke and J. Eggert are with the Honda Research Institute Europe GmbH, Carl-Legien-Str. 30, 63073 Offenbach/Main, Germany, e-mail: sven.rebhan@honda-ri.de.

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 work, 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 the mentioned methods map a fixed and predefined algorithm to a graph structure. This structure is used for parallelizing and optimizing that fixed algorithm later. 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. The graph size for measuring a certain property 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 section we present 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 propose 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 in section V.

## 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 link patterns. Thus we can define classical link types like ”hasProperty”, ”isPartOf” or ”isContainedIn” as shown in Fig. 1. Additionally, we store a sensory representation 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.

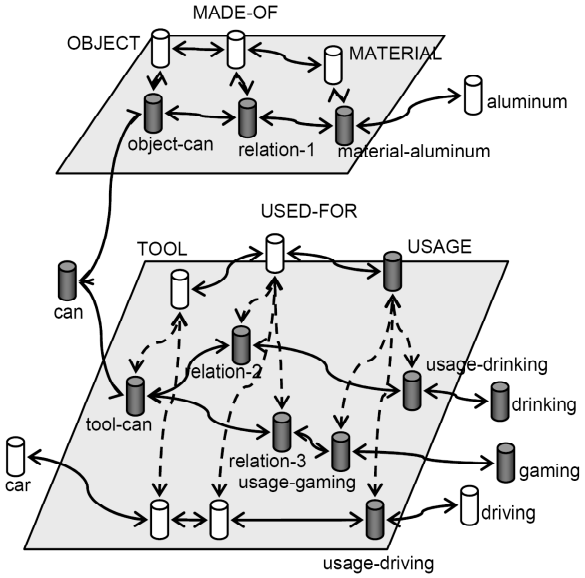


Fig. 1. The employed relational memory can represent an arbitrary number of link patterns. Some examples are shown here. See [7] for details.

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 measure *A* in a demand-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 and can thus be performed directly. Afterwards *A* can be measured. If the structures are getting more complex and the graph is getting deeper, a more sophisticated algorithm is needed to parse the graph. Details can be found in section III.

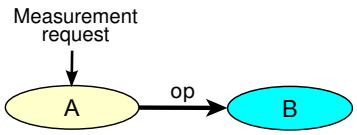


Fig. 2. The measurement of node *A* depends on the operation *op* of node *B*. We store this pattern in exactly the same memory as shown in Fig. 1.

### *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. For example a spatial modulation map could constrain the search space for an object, but is not necessary, as in the latter case the whole space has to be searched for the object (see Fig. 3 a).
- The system requires different operations for the target node to be executed 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 get a shape (see Fig. 3 c).

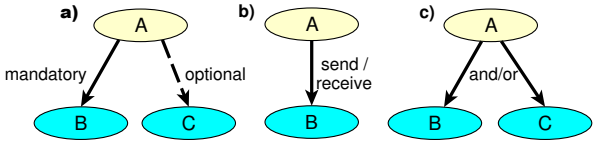


Fig. 3. We cover different cases using modifiers for a dependency link pattern: a) dependencies can be optional or mandatory, b) different operations (send and receive) are requested for the target node and c) we differentiate between the need for all dependencies or only one-of-many dependencies to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

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

- **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 depend 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*

In our case each node has a state marking the validity of the node data. We later use this to determine if the node information needs to be updated 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 when we encounter a node with valid data.

### *System Memory Layout*

After discussing the different link types, operations, modifiers and node states, we now present the actual designed prototypical memory patterns we use in our 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 only shows two different views on the memory content. Both representations coexist 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, about both the world and 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 system's vicinity. If we look back at the bottom of Fig. 4, 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 resolved dependency graph for the world location in Fig. 5 illustrates 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 a depth estimation algorithm. 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 follow the graph further, we see that the sending of the spatial modulation map itself depends on the acquisition (receiving) of the spatial modulation map (3). This makes sense, as we first must have the modulatory information before using it. Looking at the steps we have done up to here, we already can see 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 at the end of this section.

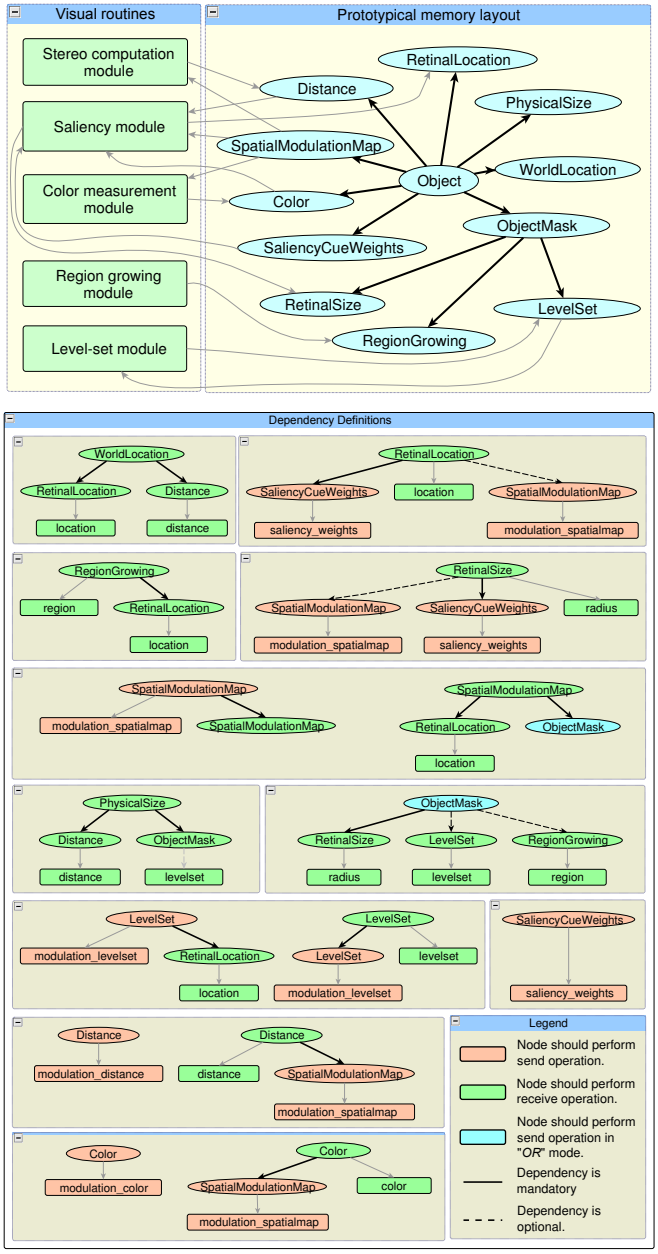


Fig. 4. On top the prototype of an object structure is displayed as used by our system. You can see the binding (gray lines) to the visual routines (rectangles), both feed-forward and feed-back. At the bottom, the dependency structure can be seen. The color indicates the operation the node should perform. The rectangles mark the bindings (gray lines) to visual routine variables.

### *Circular Dependencies Detection and Handling*

We now continue the example and pursue the dependencies one step further (see Fig. 6). We find that the measurement of the spatial modulation map depends on the measurement of an object mask and on the measurement of the retinal location (4) of the object. These two information are necessary to create the spatial modulation map at the correct location with the correct shape. However, we have already visited the retinal location node before. What we see here is a loop or *circular*

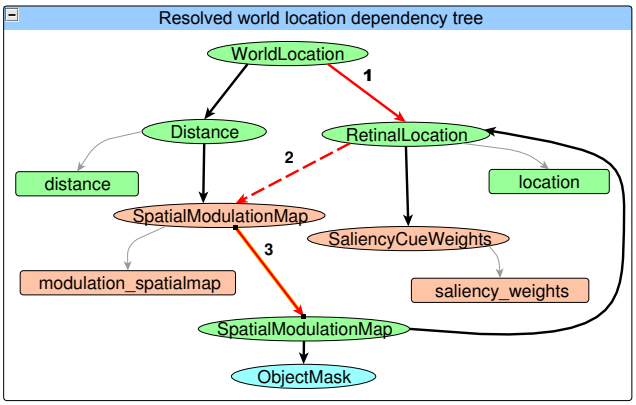


Fig. 5. The resolved dependency tree for the world location property of the object looks much more complex than the definitions in Fig. 4. We did not completely resolve the graph here for simplicity (cut at the object mask node). The path described in the text is marked red.

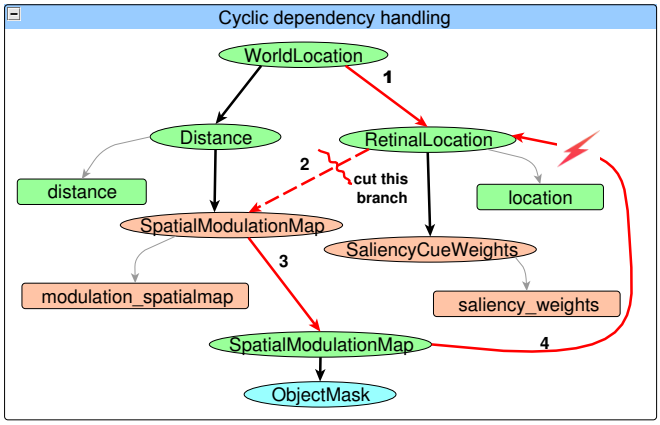


Fig. 6. The detection of a cyclic dependency at the retinal location node leads to a trace back of the path (red). Finally the circular dependency can be resolved by cutting the branch between the retinal location and the spatial modulation map nodes.

*dependency*, which would lead to a dead-lock situation if the system did not 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 a circular dependency, we go back (4) to the parent node (spatial modulation map) 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, the system can not resolve the dependencies of the current node. The latter case is true for our example, because the spatial modulation map requires the retinal location to be known. Thus the system needs to go back another step (3) 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 to trace back the dependency path one step (2). 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 not 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.

*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 parents 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 us assume that we have 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 reduced graph you can see in Fig. 7. If you

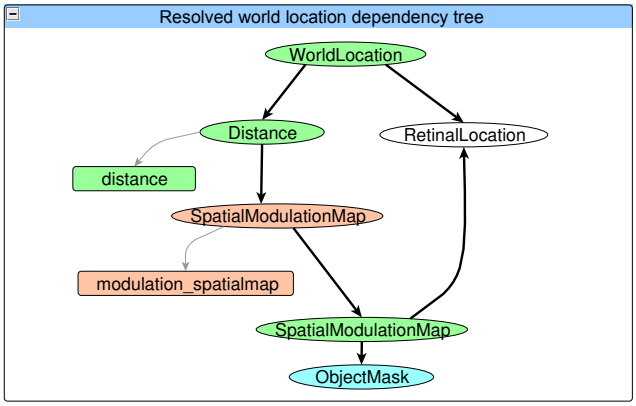


Fig. 7. By incorporating knowledge already acquired by the system, the structure of the effective dependency graph changes and its size shrinks.

compare that graph to the original one in Fig. 5, you see that the resulting graph is smaller. This means that the *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], only working 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 also need to add a way to deal with such alternative pathways. In our example shown in Fig. 8 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

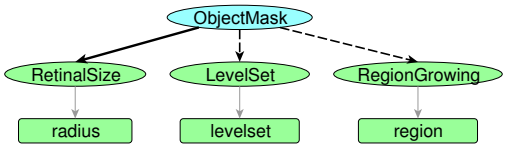


Fig. 8. The mask of an object can be calculated by any of the three algorithms (retinal size estimation using the saliency, region growing and a level-set method). However, only one of these *alternatives* has to be run to get the mask.

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 ”only 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 algorithms, 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* (but 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 can never 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 consecutively 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 the dependencies an extended version of the parsing algorithm could take into account the costs and accuracy of the different pathways.

[1]A node can deny its execution if its initial conditions are not fulfilled.

*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 into account the cyclic dependency detection and handling. The update procedure reads as follows:

**Procedure *UpdateNodeValue*:**

**(a) Check the ability of the node to run**

(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 our 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 is fulfilled.

**(c) Execute current node's operation**

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

## IV. Experiments

By using the algorithm above and the structural definitions shown in Fig. 4 we have implemented a proof-of-concept system. The memory content of the system can be seen in Fig. 9. In the ”prototypes” section of Fig. 9 only the structure of an object is defined i.e. it is defined which properties constitute an object and how they relate. In our proof-of-concept system we only use ”hasProperty” links (black). In the ”SensoryInterface” section we *inherit* the object structure and add the dependency definitions and the bindings to the visual routines. The dependency structure originates from the direct dependency definition shown in Fig. 4. The long-term memory inherits the object structure from the ”SensoryInterface”. Here only the ”hasProperty” links are shown to maintain readability. For our proof-of-concept system we handcrafted the memory content, but we are planning to investigate ways to learn the content. As you can see, not all properties are instantiated per object (here only color and size are chosen). However, other nodes like retinal location or distance can be instantiated on demand. Property nodes in the long-term memory store sensory representations which are stable for the objects linked to them. Other more volatile object information is not stored there, but rather measured in a concrete scene and stored in the short-term memory. The content of the short-term memory inherits its structure from the long-term memory. To summarize, we propagate the object structure by inheritance

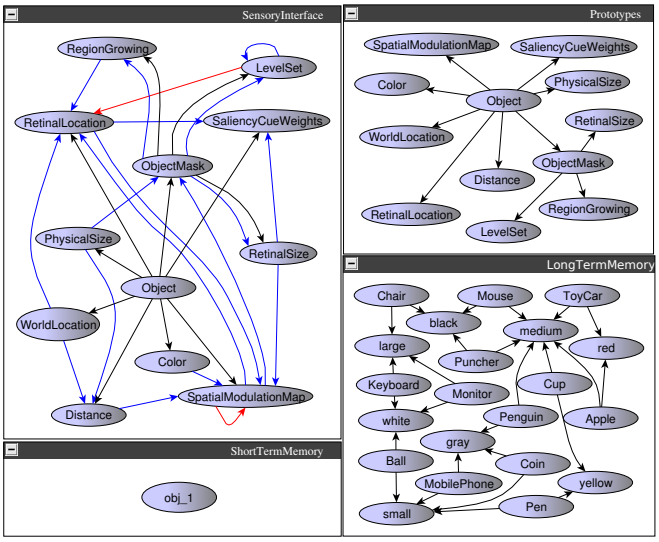


Fig. 9. On the left hand side you see both the ”hasProperty” and the ”dependsOn” connectivity. On the right hand side you see the pure structural definition (top) and the content of the long-term memory (bottom).

through the different memory instances starting from the prototypical definitions down to the short-term memory. Beside this structural information the representation is enriched by the bindings to the sensors and the graph linking nodes depending on each other. The binding and dependency graph is introduced in the ”SensoryInterface” layer. Up to this point no real values are filled into the property nodes. This happens in the long-term memory for longtime stable sensor data that can be linked to one or more objects.

Based on this memory structure we now want to show what an update process for a node looks like. The update process for the color property of object 1 is illustrated in Fig. 10. Please note that the dependency resolving process and the subsequent information propagation process are implemented asynchronously. The first step to update a property is to instantiate it (see Fig. 10 a). In doing so, the properties up to the prototypical definitions are inherited. One of those properties are the dependencies of the node. To measure the color of an object we need a spatial modulation map, which in turn requires the retinal location of an object, which again needs weight factors for the saliency. You can see the result of that propagation process in Fig. 10 b. Because the weights for the saliency have no further dependency, they can be sent right away. After doing so, all dependencies of the retinal location node are fulfilled and it requests its visual routine for data (see Fig. 10 c). After triggering the visual routine of the retinal location node, the process continues at the spatial modulation map. As defined previously (see Fig. 4), the modulation map requires an object mask to be processed. The object mask is an *OR node*, because three alternative measurement processes exist (retinal size estimation, region growing and a level-set method). Only one of these visual routines needs to run. As shown in Fig. 10 d, the object mask node first tried to trigger the level-set measurement. However, as described in section III, this algorithm needs

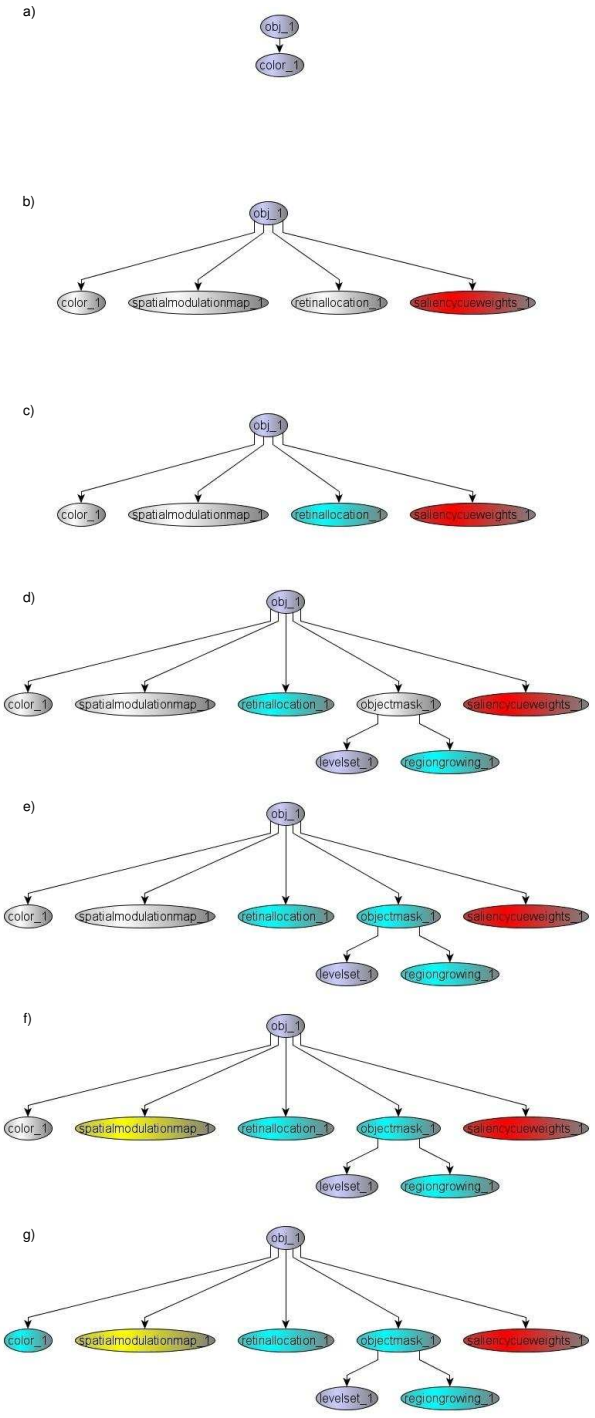


Fig. 10. This shows the update process for the color property of object 1 zoomed into the short-term memory. Color codes: Nodes currently resolving dependencies (gray), nodes waiting on data to receive (cyan), nodes waiting on data to send (yellow) and nodes finished sending data (red).

an initial mask to run. We do not have such a mask yet, so we tried the region growing method next, which was successful. Because the region growing node was the last leaf node, the dependencies could be resolved by tracing back the dependency path while executing the node operations. This

can be seen in Fig. 10 e to g. Finally, all cyan nodes triggered their visual routine to deliver data. The spatial modulation map (yellow) waits on data to send them.

When the data arrives from the visual routines, they get propagated upwards along the dependency tree. In Fig. 11 the retinal location and the mask of the region growing algorithm arrive (almost) simultaneously. The information about the

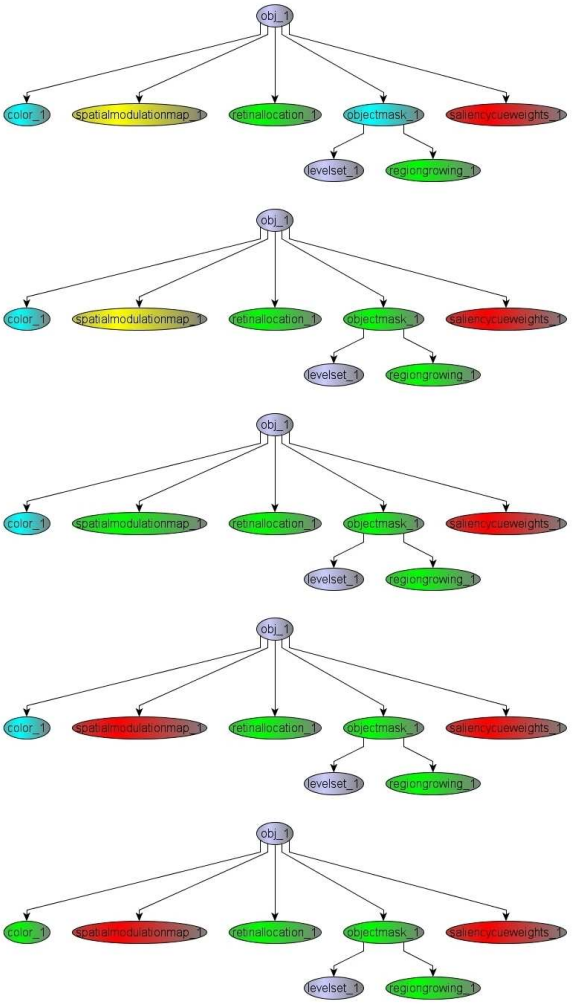


Fig. 11. The arriving information (retinal size and region growing mask) travels along the dependency tree and triggers computations in their parent nodes. Color code as in Fig. 10 and nodes have finished receiving (green).

region growing mask then travels upwards to the object mask where the processing can be finished. With a valid object mask and the retinal location of the object, a spatial modulation map can be computed and sent subsequently. After sending the spatial modulation map, the visual routine of the color node runs and eventually returns the color (see bottom of Fig. 11).

Now let us assume that after some time, the data of some nodes gets invalid again. Such a case is shown in the top row of Fig. 12 where the data of the nodes retinal location, region growing and object mask got invalid. Furthermore, let us assume that the system needs to know the distance of object 1. Receiving the distance requires sending the spatial

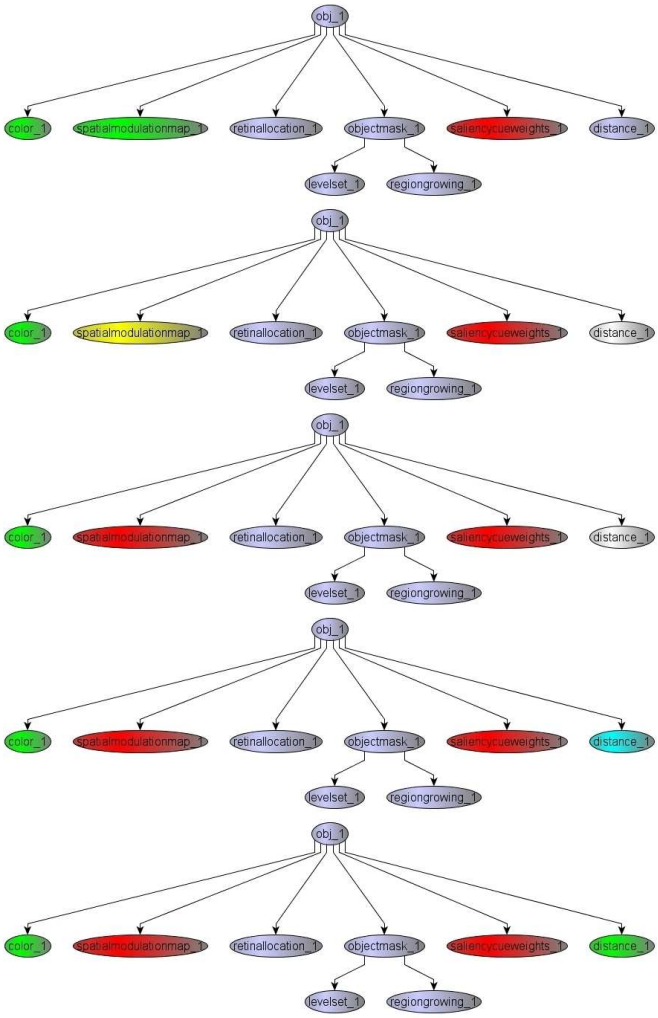


Fig. 12. The system requests the distance of the object. During this process, the existing information about the spatial modulation map gets reused. This reduces the computational demand dramatically.

modulation map (see Fig. 4). Typically, sending this map requires the retrieval of the map, but in this case, the data of the spatial modulation map is still valid. Thus there is no need for updating and we can continue with just sending out the information. If you compare this to the procedure in Fig. 10, where sending the spatial modulation map triggered the whole object mask and retinal location branch, the computational effort is reduced dramatically. Finally, the distance is requested and received by the system as shown at the bottom of Fig. 12.

## V. Conclusion and Future Work

In this paper we have presented a system that uses graph structures to represent both knowledge about functional dependencies and knowledge about the world in a consistent way. We exploit the fact that the relational semantic memory in our system can represent an arbitrary number of link patterns between nodes and is furthermore able to bind visual routines to its nodes. In the paper we concentrated on the modeling of dependency links and introduced some modifiers

for that link pattern, which allow to cover important use cases for vision systems. We have discussed the cases of optional and mandatory information, different node operations modeling the direction of information flow and alternative pathways. We proposed a parsing algorithm based on the dependency link structure, which is able to detect and under certain circumstances ”resolve” circular dependencies. Beyond this, the parsing algorithm is also able to efficiently reuse previously acquired sensory information and thus reduces the computational demand while keeping the full function of the system. Our experiments show that with this framework we are able to build systems that acquire data on demand and are able to flexibly adapt their processing chain.

Even though this paper provides a basis for such flexible control structures, many interesting questions that arose during the experiments are not answered yet. Some of those questions and ideas we would like to mention here. One thing we did not cover in this paper is the possibility for the system to estimate the costs of a certain action. In section II we briefly mentioned that the number of dependency nodes can be used as a cost function. However, one could also think of measuring the time a certain action takes and use this as a cost function. With this information, the system could learn which actions it can take if a time constraint applies. In the same direction, the system could furthermore learn how accurate and reliable a certain pathway is and use fast but coarse functions in cases where precise information is not necessary. To push this a step further, the system could also try to find the dependencies itself and learn ”optimal” processing queues.

Beside estimating the dependency structure we could easily exploit the consolidated findings from computer science like [4] and later work to optimize and parallelize processing pathways. The algorithms found there can be easily applied, as the underlying structure is comparable.

## 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.
[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.