Self-generating Programs – Cascade of the Blocks

Josef Kufner, Radek Mařík

Self-generating Programs – Cascade of the Blocks

Josef Kufner and Radek Mařík

Department of Cybernetics, Faculty of Electrical Engineering,
Czech Technical University in Prague, Czech Republic
kufnejos@fel.cvut.cz, marikr@k333.felk.cvut.cz

Abstract When building complex applications the only way not to get lost is to split the application into simpler components. Current programming languages, including object oriented ones, offer very good utilities to create such components. However, when the components are created, they need to be connected together. Unluckily, these languages are not a very suitable tool for that. To help with composition of the components we introduce *cascade* – a dynamic acyclic structure built from blocks, inspired by the Function Block approach. The cascade generates itself on-the-fly during its evaluation to match requirements specified by input data and automatically orders an execution of the individual blocks. Thus the structure of a given cascade does not need to be predefined entirely during its composing/implementation and fixed during its execution as it is usually assumed by the most approaches. It also provides a real-time and fully automatic visualization of all blocks and their connections to ease debugging and an inspection of the application.

1 Introduction

In last 30 years object oriented languages have developed to state, where they are a pretty good tool for creating components, but when it comes to composing these components together, the situation is far from perfect.

There are not many successful and widely used tools or design patterns to compose applications. Probably the most known approach, which is well established in the field of programmable logic controllers, is *Function Blocks* [1], and its simplified variant, used by unix shells, known as *Pipes and Filters*. Despite these approaches are decades old [2], they are used in only few specific areas, and there is very low development activity in this direction.

A basic idea of both Function Blocks and Pipes and Filters is to split a complex application to simpler blocks, and then connect them together using well defined and simple interfaces. This adds one level of abstraction into the application and simplifies significantly all involved components. Simpler components are easier to develop. Well defined interfaces improve reusability of the components. In total, it means faster and more effective development.

From other point of view the connections between blocks can be easily visualized in a very intuitive way and conversely these connections can be specified

using a graphical editor. This significantly lowers programming skill requirements and allows non-programmers to build or modify applications if a suitable GUI tool is provided.

A main limitation of Function Blocks is that blocks are often connected together in advance by a programmer and this structure remains static for the rest of its life time. Therefore, an application cannot easily adapt itself to changing requirements and environment, it can usually change only few parameters. The next few sections will present how to introduce dynamics into these static structures and what possibilities it brings.

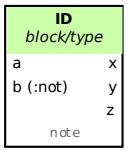
In this paper we introduce *cascade*, a dynamic acyclic structure built of blocks. The next two sections (2, 3) describe the blocks and how they are composed into a cascade, including a description of their important properties. Then, we explain the most interesting features of the cascade in Section 4. Finally, in the last two sections (6, 5) a practical use of cascade is described and it is also compared to existing tools and approaches.

2 Creating the blocks

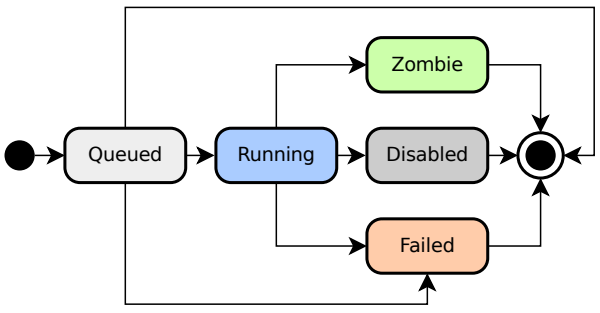
As mentioned above, object oriented languages are very good tools to create components. So it is convenient to use them to create blocks.

In cascade a block is atomic entity of a given type (class), which has named inputs and outputs, each block instance is identified by unique ID, and can be executed. During execution the block typically reads its inputs, performs an action on received data, and puts results to its outputs.

The symbol used in this paper to represent a block is in Figure 1a. There are an ID and a block type in the header, named inputs on the left side and outputs on the right. A color of the header represents the current state of the block. At the bottom a short note may be added, for example a reason of failure.



(a) The block symbol



(b) Lifetime of the block

Figure 1: The block

The block is implemented as a class, which inherits from the abstract block class. This abstract class implements a required infrastructure to manage inputs, outputs and execution of the block. Execution itself consists of calling `main` method of the block class.

Each block instance is executed only once. During its lifetime block goes through a few states as presented in Fig. 1b. It starts in the state *queued*, where waits for execution. Then it enters state *running*, and when execution is completed, one of the final states *zombie* (success), *disabled*, or *failed* is entered. In the final state the block exists only to maintain its outputs for other blocks[1]. This means that the block can process only one set of input data. To process another data set, a new block instance must be created. But because the lifetime of the block ends before another input data arrive, it causes no trouble (this will be explained later).

3 Connecting the blocks

To create something interesting out of the blocks, we need to connect them together. These connections are established by attaching an input of the second block to the output of the first block – the connections are always specified at inputs, never at outputs. So the input knows where its value came from, but the output does not know whether it is connected somewhere at all. Transfers of values over the earlier established connections are also initiated by the inputs. That means the outputs are passive publishers only. By connecting blocks together a *cascade* is being built.

The cascade is a directed acyclic graph composed of blocks, where edges are connections between outputs and inputs of the blocks.

When a block is inserted into cascade, its inputs are already entirely defined. It means, that a connection to another block or a constant is assigned to each of its inputs. At this moment, connections are specified using block ID and output name. Later, when block is being executed, actual block instances assigned to these block IDs are resolved, so the specified connections can be established and values transferred from outputs to inputs. Thanks to that, it does not matter in which order blocks are inserted into cascade, as long as there are all required blocks present before they need to be executed.

3.1 Evaluation

Evaluation of the cascade is a process, in which the blocks are executed in a correct order, and data from the outputs of executed blocks are passed to the inputs of the blocks waiting for execution.

By creating a connection between two blocks, a dependency (precedence constraint) is defined, and these dependencies define the partial order, in which

[1] Therefore the successful state is called *zombie*, like terminated unix process whose return value has not been picked up by parent process.

blocks need to be executed. For single threaded evaluation, a simple depth-first-search algorithm with cycle detection can be used to calculate topological order compatible with given partial order [3].

Since DFS algorithm requires a starting point to be defined, selected blocks (typically output generators[2]) are enqueued to a *queue*, when inserted into cascade. Then the DFS is executed for each block in the queue. If a block is not enqueued, it will not be executed, unless some other block is connected to its outputs. This allows preparing set of often, but not always, used blocks and let them execute only when required. Evaluation of the cascade ends, when execution of the last block is finished and the queue is empty.

These features relieve the programmer from an explicit specification of execution order, which is required in traditional procedural languages.

3.2 Visualization

It is very easy to automatically visualize connections between blocks. Once cascade evaluation is finished, its content can be exported as a code for Graphviz[3], which will automatically arrange given graph into a nice image, and this generated image can be displayed next to results of the program with no effort. It is a very useful debugging tool.

Note that there is no need for a step-by-step tracing of cascade evaluation, since the generated image represents the entire process, including errors and presence of values on connections.

For example, Figure 2 shows a cascade used for editing an article on a simple web site. An article is loaded by the block `load`, then is passed to the block `form`, which is displayed to user by the block `show_form`. Because form has not been submitted yet, the block `update`, which will store changes in the article, is disabled (a grey arrow represents `false` or `null` value). This figure was rendered by Graphviz and similar figures are generated automatically when creating web sites with a framework based on the cascade (see section 6).

3.3 Basic definitions

It is necessary to define few basic concepts and a used notation, before a behavior of cascade can be described in detail.

A block name in the text is written using monospaced font, for example `A`. An execution of the block `A` starts with an event A (i.e. begin of the execution) and ends with an event $\bar{A}$ (i.e. end of the execution).

During the event A the `main` method of the block is called. Within the `main` method block reads its inputs, performs an operation on received data, and sets its outputs.

---

[2] Output generator is a block which prepares data for a future HTTP response as a side-effect. The prepared data are passed to template engine when cascade evaluation is finished.

[3] Graphviz: `http://www.graphviz.org/`

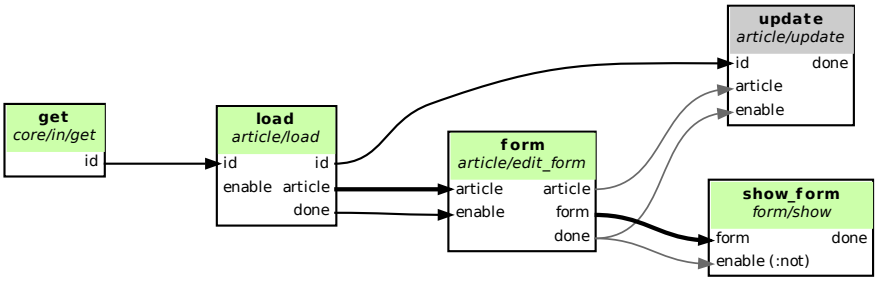


Figure 2: Cascade example

During the event $\bar{A}$ cascade performs all requested output forwarding (see section 4.4) and the block execution is finished. The block itself performs nothing at this point.

Because any execution of a block must begin before it ends, a trivial precedence constraint is required for each block:

$$A \prec \bar{A} \tag{1}$$

3.4 Automatic parallelization

One of the first questions usually asked after a short look at the cascade is whether the blocks can be executed in parallel. A short answer is "yes, of course" and in this section we try to explain how straightforward it is.

Let there are blocks `A`, `B` and `C`, where `C` is connected to some outputs of blocks `A` and `B`, as displayed in Figure 3.

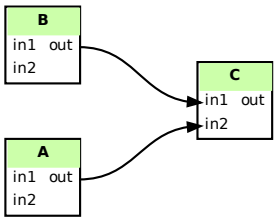


Figure 3: Automatic parallelization example

Because the begin of block execution precedes its end, trivial precedence constraints are defined:

$$A \prec \bar{A},\ B \prec \bar{B},\ C \prec \bar{C} \tag{2}$$

Due to the connections, execution of the blocks `A` and `B` must be finished before execution of `C` begins:

$$\bar{A} \prec C,\ \bar{B} \prec C \tag{3}$$

When this cascade is evaluated in single thread, blocks have to be executed in a topological order, which is compatible with partial order defined by precedence constraints (2) and (3). Assuming non-preemptive execution of the blocks, there are two compatible orders:

$$A \prec \bar{A} \prec B \prec \bar{B} \prec C \prec \bar{C} \tag{4}$$

$$B \prec \bar{B} \prec A \prec \bar{A} \prec C \prec \bar{C} \tag{5}$$

Both (4) and (5) will give exactly the same results, because blocks `A` and `B` are completely independent. Therefore these two blocks can be executed in parallel with no trouble:

$$\Big(\big(A \prec \bar{A}\big) \parallel \big(B \prec \bar{B}\big)\Big) \prec C \prec \bar{C} \tag{6}$$

A naive implementation of a parallel execution can be done by spawning a new thread for each block, and using a simple locking mechanism to postpone execution of blocks with unsolved dependencies. More efficient implementations may involve a thread pool and per-thread block queues for solving dependencies. Since cascade is being used to generate web pages, the block-level parallelization was not investigated any further, because all web servers implement parallelization on per-request basics.

4 Growing cascade

So far, everything mentioned here is more or less in practical use by various tools, especially in data mining, image processing and similar areas. What makes cascade unique, is the ability to grow during the evaluation.

When block is being executed, it can also insert new blocks into cascade. Inputs of these blocks can be connected to the outputs of any other blocks (as long as circular dependencies are not created) and it can be enqueued to the queue for execution. The algorithm described in section 3 will handle these new blocks exactly the same way as previous blocks, because it iterates over the queue and the new blocks will be enqueued there before the enqueuing block is finished.

4.1 Namespaces

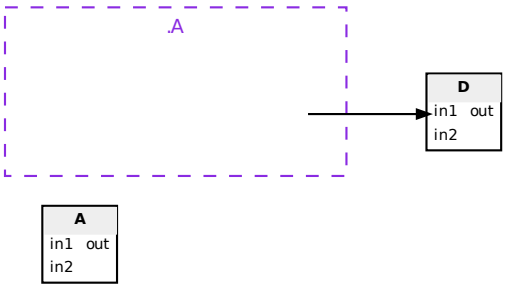
To avoid collisions between block IDs, each block inserts new blocks into its own namespace only. These namespaces are visualized using a dashed rectangle with an owner block ID under the top edge of the namespace rectangle.

In traditional languages like C namespaces are used to manage visibility (scope) of local variables, where code located outside of a namespace cannot access a variable defined inside this namespace, but the inner code can reach global variables. However the global variables can be hidden by local variables.

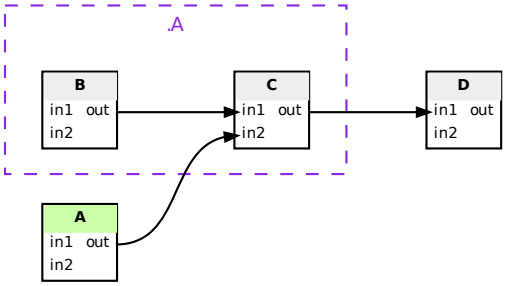
The same approach is used in the cascade with a small difference – it is possible to *explicitly* access content of a namespace from outside, so connections across the namespaces can be made with no limitation.

To identify block in other namespace, a dot notation is used. For example block `B` in the namespace of block `A` in the root namespace is referred as `.A.B` (see Figure 4b). If there is no leading dot, the first block is searched in the namespace of current block and all its parent namespaces up to the root. By specifying additional blocks it is possible to enter namespaces of other and completely unrelated blocks.

Since the primary purpose of namespaces is to avoid collisions in IDs, there is no reason to deny connections from the blocks outside of the namespace (see block `D` in Figure 4b). This allows to extend existing applications by attaching additional blocks without need to change the application.



(a) Cascade *before* execution of the block `A`



(b) Cascade *after* execution of the block `A`

Figure 4: Growing cascade

4.2 Dependencies during the growth

The secondary purpose of the namespaces is to help handling dependencies on blocks that are not yet present in the cascade.

Take a look at Figure 4a. The block `D` is connected to a so far nonexistent block `C` inside the namespace of the block `A`, and block `A` has not been executed yet. In this moment, the only known precedence constraint is $\bar{C} \prec D$ and because there is no connection between `A` and `D`, block `D` could be executed before `A`. But that would end up with a "block `C` not found" error.

Since the only block, which can insert blocks into namespace of the block `A`, is the block `A`, additional precedence constraint can be introduced: Each block depends on its parent (creator). Therefore cascade in Figure 4a contains following precedence constraints:

$$\bar{A} \prec C,\ \bar{C} \prec D \tag{7}$$

And because block `C` is not present yet, the only compatible topological order is:

$$A \prec \bar{A} \prec D \prec \bar{D} \tag{8}$$

Block `A` inserts blocks `B` and `C` into the cascade during its execution and additional precedence constraints are created:

$$\bar{B} \prec C,\ \bar{A} \prec C,\ \bar{A} \prec B \tag{9}$$

Note that $\bar{A} \prec C$ is there for the second time, because of the connection between blocks `A` and `C`. And constraint $\bar{A} \prec B$ is already fulfilled, because block `A` has been executed already.

Now, after the block `A` is finished, the cascade contains new blocks and new precedence constraints, so new topological order must be calculated before a next block is executed. The only topological order compatible with (7) and (9) is:

$$\underbrace{A \prec \bar{A}}_{\text{executed}} \prec \underbrace{B \prec \bar{B} \prec C \prec \bar{C} \prec D \prec \bar{D}}_{\text{queued}} \tag{10}$$

4.3 Safety of the growth

It is not possible to break an already evaluated part of the cascade by adding new blocks. Reason is fairly simple – the new blocks are always appended after the evaluated blocks. That means the new precedence constraints are never in conflict with already existing constraints, and the new topological order is always compatible with the old one.

Each block is executed only after execution of all blocks it depends on. And because connections are specified only when a block is inserted into the cascade, the already executed part of the cascade cannot be modified. It also means that all precedence constraints in the executed part of the cascade have been fulfilled.

When inserting a new block, there are two kinds of connections which can be created: a connection to an already executed block, and connection to enqueued or missing block. When connected to the already executed block, any new precedence constraint is already fulfilled. When connected to the enqueued or missing block, the order of these blocks can be easily arranged to match the new constraints – the situation is same as before execution of the first block.

For example, the only difference between topological orders (8) and (10) is in added blocks `B` and `C`, thus the relative order of all actions occurring in (8) is same as in (10).

4.4 Nesting of the blocks and output forwarding

The basic idea of solving complex problems is to hierarchically decompose them into simpler problems. Function call and return statement are the primary tool for this decomposition in λ-calculus and all derived languages (syntactical details are not important at this point). The function call is used to breakdown a problem and the return statement to collect partial results, so they can be put together to the final result.

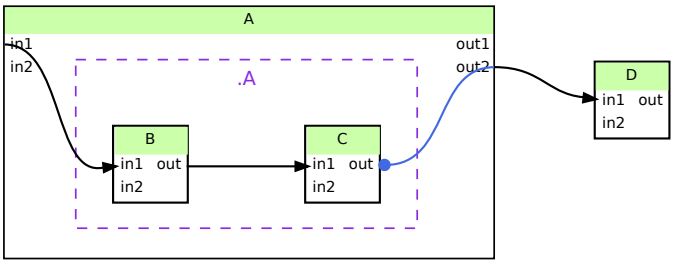


Figure 5: Idea of nested blocks

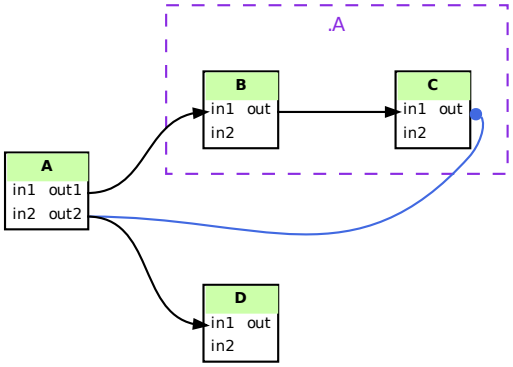


Figure 6: "Nested" blocks in real cascade

But the cascade evaluation is one-way process. The entire concept of return value makes no sense here. Everything in the cascade goes forward and never looks back. Also there is no stack in the cascade where return address could be stored, so even if something would want to return a result, it has no chance of knowing where to.

To allow a hierarchical problem decomposition in the cascade, a slightly different tool was created – *output forwarding*. When block solves some problem, it presents results on its outputs. When block delegates solving to some other blocks, the results are on their outputs. To achieve the same final state as before, the results must be transferred to the original outputs of the delegating block. This schema exhibits similar behavior as the return statement.

From other point of view, there is no need to transfer result values back, if all connections are redirected to the forwarded outputs. Both approaches are equivalent and they both preserve all properties of the cascade mentioned earlier.

For example, let block `A` perform a complex task and block `D` display result of this task – see Figure 5. Block `A` inserts blocks `B` and `C` into the cascade, and passes a parameter from its input to the input of the block `B` and then collects a result from block `C`, which received a partial result from the block `B`. The final result is then published on an output of the block `A`. Figure 5 presents this solution as it is usual in λ-calculus based languages.

The cascade is a flat structure and does not allow nesting of the blocks. It emulates this hierarchy using namespaces, but all blocks are in the flat space. Therefore, blocks `B` and `C` are inserted next to the block `A`, as presented in Figure 6.

Note that output forwarding can be chained over multiple outputs. For example some other blocks could request forwarding of the output of the block `A` in Figure 6. In such cases output forwarding is solved recursively with no trouble.

Output forwarding adds exactly the same precedence constraint as any other connection between the blocks. The situation here is almost the same as on Figure 4b. When the value copying approach is used, the output forwarding adds the following constraint:

$$\bar{C} \prec \bar{A} \tag{11}$$

This is because all connected blocks are tied to the $\bar{A}$ event, so it is easier to delay this event until dependencies are solved. If the second (redirecting) ap-

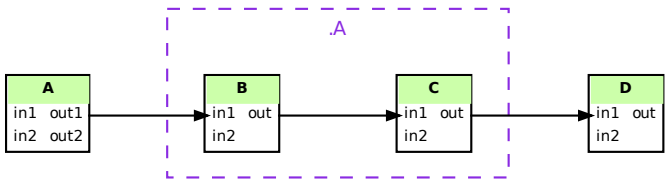


Figure 7: Equivalent cascade without output forwarding

proach is used, the situation would be exactly the same as presented in Figure 7, but an implementation of this transformation may be too complicated.

All constraints in the example cascade in Figure 6 are:

$$\bar{A} \prec B,\ \bar{A} \prec C,\ \bar{B} \prec C,\ \bar{A} \prec D,\ \bar{C} \prec \bar{A}$$

And the compatible topological order is:

$$A \prec B \prec \bar{B} \prec C \prec \bar{C} \prec \bar{A} \prec D \prec \bar{D} \tag{12}$$

4.5 Semantics of the output forwarding

Using the output forwarding a block says to the cascade: "When I'm done, set my output to a value which is on that output of that block." From a block's point of view the output forwarding is exactly the same process as setting a value to an output. Only difference is in specifying what value will be set – output forwarding uses a reference instead of a value.

The namespaces and the output forwarding were both designed to allow a hierarchical decomposition of a problem, but they are completely independent tools in contrast to a function call and a return statement in λ-calculus based languages. And since the output forwarding is not limited to any particular namespace, it allows creation of a very unusual use cases, where the "return values" can be picked up anywhere in the cascade.

5 Comparison with other approaches

Probably the most similar approach to the cascade is Function Block programming [1], which has a very long history in industrial automation to program PLCs[4]. The main difference from the cascade is that blocks and connections between them are static. The connections are left unchanged as long as the programmer does not upload a new version of software into PLC. The second difference is in data transmitted via inputs and outputs. In the cascade once output is set, it stays constant forever. But in Function Blocks it is a stream of values.

Various data processing tools like Orange [4], Khoros Cantata [5], and Rapid Miner [6] adopted the function blocks approach with some modifications. But the basic difference still stands – all these tools use a static structure built from blocks by the programmer.

λ-calculus differs from the cascade in the semantics of a return statement and in a heavy use of a stack. Because there is no stack in the cascade, the return statement is thrown away and replaced by the output forwarding mechanism, which has a slightly different semantics, but it can be used to achieve the same goals.

[4] PLC: Programmable Logic Controller

Cascade exhibits a number of shared features with Hierarchical Task Networks (HTN) planning [7]. HTN planning provides a much more sophisticated approach including backtracking and different constraint mechanisms, such as constraint refinement, constraint satisfaction, constraint propagation, etc. Cascade trades these advanced methods with an execution speed. The fast execution is achieved by the elimination of decomposition alternatives.

The preference calculus based on properties of partial order relations forms a foundation of dependency trees heavily used in scheduling theory [8].

A dependency injection is a software design pattern used to ease dependencies in the code [9]. A basic idea is in a separation of tool users (code) from creators (factory) of the tool (object), so the creators can be easily replaced. In the cascade, this approach is very natural to use, since it is easy to wrap the creator into a common block and let users connect to this block and retrieve the tool. Thanks to the embedded visualization of the cascade it is very easy to track these dependencies in an application.

6 Real world application

6.1 Web framework

The cascade serves as a core of a push-style web framework written in PHP, where cascade takes a place of a controller (as in MVP pattern). Since a cascade does not have any own inputs and outputs, specialized blocks are used as a connectors to receive input data from a HTTP request and to pass response data into a template engine (view). Processing of any HTTP request is split into two stages. During the first stage the cascade is evaluated and response data are prepared as a result. In the second stage the template engine generates a final web page using the prepared data.

When a development mode is enabled, each web page contains automatically generated visualization of a cascade used to create that page (see Section 3.2), so the debugging tool is always at hand.

Also a simple profiler is embedded into the framework. An average time needed to evaluate a cascade in real applications (ca. 20–40 blocks per page) is approximately 30 ms and an overhead of the cascade is less than 3 ms of that time, which is as good as widely used web frameworks.

6.2 Modular applications

Extensible applications typically declare places, where they can be extended, which is usually done using hooks or virtual methods. The cascade uses different approach. It offers user a set of blocks and it is up to him what he will build. And when there is some special need, which cannot be satisfied using available blocks, an additional set of blocks shall be created.

The cascade introduces an unified mechanism allowing various blocks from foreign sets to be connected into one structure, but it does not specify, how this

structure should look like. It is a job for the framework built above the cascade to specify common fragments of these structures and define configuration style, so the solid base for application is created.

6.3 Rebuild rather than modify

The cascade tries to make maximal use of block reusability. When a new part of an application is to be created, a many of required blocks are already available for use, so the new part of the application can be built in a little time. Thanks to this, a need for adjusting existing structures to suit current needs is significantly reduced, because they can be thrown away and easily replaced.

6.4 Generating the application

To avoid repetition while composing blocks, it is possible to algorithmically generate these structures. A block which inserts additional blocks into the cascade is not limited on how it should get a list of these blocks. It may be from a static configuration file or a database, but the block can interpret these data and use them as templates or as parameters of a predefined template.

The cascade is designed to support this behavior. The dynamic insertion of blocks into the cascade is its key feature. The way, how the cascade is built (see Section 3), makes very easy to attach new blocks to existing sources of data. Because the order of blocks is not significant while inserting them into the cascade, the generating blocks can be much simpler than traditional code generators. Also, it is easy to use multiple blocks to generate a single structure, since connections can be made across namespaces (see Section 4.1).

When this approach is combined with sufficient metadata about entities in the application, the results may be very interesting.

7 Conclusion

In practice the cascade made development more effective, because it supports a better code reusability while creating a new application, and it helps the developer to analyse an old code when modifying or extending an existing application. The graphical representation of the code is very helpful when tracking from where broken data arrived and what happen to them on their way. It reduces significantly time required to locate a source of problems.

A dynamic nature of the cascade allows the programmer to cover a large number of options while using a fairly simple structure to describe them. And since the execution order is driven by the cascade, the programmer does not have to care about it. That means less code and less space for errors.

A main contribution of the cascade is extending the time-proven function blocks approach with new features making it suitable for many new use cases and preparing base ground for further research.

References

1. K. John and M. Tiegelkamp, *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making AIDS*. Springer-Verlag, 2001. [Online]. Available: http://books.google.cz/books?id=XzlYGLulBdIC
2. D. M. Ritchie, "The evolution of the unix time-sharing system," *Communications of the ACM*, vol. 17, pp. 365–375, 1984.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.
4. T. Curk, J. Demsar, Q. Xu, G. Leban, U. Petrovic, I. Bratko, G. Shaulsky, and B. Zupan, "Microarray data mining with visual programming," *Bioinformatics*, vol. 21, pp. 396–398, Feb. 2005. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/21/3/396.full.pdf
5. K. Konstantinides and J. Rasure, "The Khoros software development environment for image and signal processing," *Image Processing, IEEE Transactions on*, vol. 3, no. 3, pp. 243–252, 1994.
6. I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler, "Yale: Rapid prototyping for complex data mining tasks," in *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, L. Ungar, M. Craven, D. Gunopulos, and T. Eliassi-Rad, Eds. New York, NY, USA: ACM, August 2006, pp. 935–940. [Online]. Available: http://rapid-i.com/component/option,com_docman/task,doc_download/gid,25/Itemid,62/
7. S. Sohrabi, J. A. Baier, and S. A. McIlraith, "Htn planning with preferences," in *Proceedings of the 21st international jont conference on Artifical intelligence*, ser. IJCAI'09. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009, pp. 1790–1797. [Online]. Available: http://dl.acm.org/citation.cfm?id=1661445.1661733
8. R. Rasconi, N. Policella, and A. Cesta, "*SEaM:* analyzing schedule executability through simulation," in *IEA/AIE*, ser. Lecture Notes in Computer Science, M. Ali and R. Dapoigny, Eds., vol. 4031. Springer, 2006, pp. 410–420.
9. N. Schwarz, M. Lungu, and O. Nierstrasz, "Seuss: Decoupling responsibilities from static methods for fine-grained configurability," *Journal of Object Technology*, vol. 11, no. 1, pp. 3:1–23, Apr. 2012. [Online]. Available: http://www.jot.fm/contents/issue_2012_04/article3.html