*Generating process diagrams for control software in the Reflex language*

Sofia Evgenievna Belenkaia   
*Novosibirsk state university*  
*Institute of Automation and Electrometry of the Siberian Branch of the Russian Academy of Sciences (IA&E SB RAS)*Novosibirsk, Russia  
s.belenkaia8@g.nsu.ru

Vladimir Evgenievich Zyubin   
*Institute of Automation and Electrometry of the Siberian Branch of the Russian Academy of Sciences (IA&E SB RAS)*Novosibirsk, Russia  
zyubin@iae.nsk.su

Andrei Sergeevich Rozov  
*Institute of Automation and Electrometry of the Siberian Branch of the Russian Academy of Sciences (IA&E SB RAS)*Novosibirsk, Russia  
rozov@iae.nsk.su

*Abstract*— Many domain-specific languages (DSL) are used in the sphere of industrial automation and embedded systems, e.g. process-oriented programming languages. Usually, there are tasks of reverse engineering and code refactoring in a process of developing and supporting projects, therefore, developers need documentation to the source code. Automatic documentation tool may decrease the time for creating diagrams and protect them from a human error.

In this work, we analyze available graph description languages and present an approach to automatic generation of graphical documentation for specialized programming languages. The presented approach involves generating diagrams as graph forms of the source code and using already available external tools for laying out, visualization and editing purposes. A software module was created for the Reflex programming language to generate a process data communication diagram, control diagram and states diagram. Also, parts of graphs that don’t share nodes are automatically written to different files. The module automates the creation of the graphical documentation and simplifies the support of Reflex projects.

Practical testing has shown that using the developed module helps to decrease the time expenses for creating process diagrams (from a few hours to a couple of seconds) and guarantees that there are no errors caused by human factor.

Keywords— control software, process diagrams, graphical documentation, process-oriented programming, code refactoring, reverse-engineering, code analysis

# Introduction

Software development projects can often benefit from reverse engineering techniques. Such methods may be utilized in code refactoring and documentation both in development and maintenance phases of a project. For commonly used object-oriented languages such as Java, C++, C#, etc. various software tools are available for automatic generation graphical documentation. In many cases, these tools are integrated into the development environment for the language.

Developing code analysis tools in industrial automation, writing control software and programming embedded systems is of especially high interest for problem-oriented languages (DSL). In particular, this problem is relevant for Reflex process-oriented programming (POP) language [3], [4], [5], which is actively developed in the Institute of Automation and Electrometry of the Siberian Branch of the Russian Academy of Sciences. POP [1], [2], [3] is intended to write a control program as a set of intercommunicated processes. While process-oriented programming increases the quality of the control software, code analysis of programs provides an additional safety level. In current projects, code analysis is performed manually, which takes a lot of time, and the resulting documentation is likely to contain errors because of the human factor.

# Purpose of work and tasks

In this work, we develop an approach to create the automatic documentation, and for testing our method we develop a software module for building process diagrams according to the specification in the Reflex language. We analyze: specific features of process-oriented programming language Reflex, existing tools for generating diagrams from the source code and types of notations for diagrams that are used for code analysis. We determine requirements for the software module, and diagram notation for showing communications between processes is developed. We choose the file format for saving diagrams. Also, an architecture of the software module is developed, the software module is implemented and tested.

# Specific Reflex language features.

Control algorithm is represented by a hyperprocess. Program name in the Reflex language is written after the reserved word ‘program’. Then, the body of the program is put inside the braces and consists of attributes of the hyperprocess specification and of processes definitions. Attributes may be input/output ports, a period of process starting cycle and constants.

Processes are defined sequentially. The process, which was written first, is in the active state. All other processes in the first moment of running the program are in passive state. State functions are defined sequentially inside the process body. State name is specified after the reserved word ‘state’. State body is defined inside braces and consists of events and reactions to events, which are defined by the standard C operators and some special Reflex operators.

Reflex language has operators start/stop <process name>, set state <state name> for changing the state of the current process, and timeout <time specification>. Process can import variables from another processes using operator ‘from proc <process name> <names of variables>’.

A Reflex program consists of process definitions, which are defined by state machines. Processes run in the cooperative model of multithreading and communicate with each other via shared variables and special control statements (i.e. process can start and stop each other and themselves, and import variables from other processes) [3], [4], [5].

# Analysis of existing tools for generating diagrams from code

We have found out that not all tools for generating diagrams from the source code for common programming languages have the ability for code generation, dynamic diagram building and hiding components for diagram, but all of them have the ability to modify diagrams. The results of our analysis of existing tools for diagram generation from the source code for common programming languages are shown in Table 1.

1. tools for generating diagrams from code for common languages

| Tool name | Tool features | | | |
| --- | --- | --- | --- | --- |
| Code generation | ***Diagrams modification*** | ***Hiding diagram components*** | Dynamic building |
| Class Designer for Visual Studio | - | + | + | + |
| Astah UML | + | + | Not found | Not found |
| MagicDraw | + | + | Not found | + |
| Software Ideas Modeler | + | + | + | + |
| BOUML | + | + | + | - |
| Visual paradigm | + | + | Not found | Not found |
| Rational Rose | + | + | + | + |
| Enterprise Architect | + | + | Not found | Not found |
| IntelliJ Idea | - | + | + | + |
| Sybase PowerDesigner | + | + | + | - |
| NetBeans | - | + | - | - |
| Lab view | + | + | - | Not found |
| Altova UModel 2008 | + | + | Not found | Not found |

By analyzing software development tools for control software and embedded systems programming, we have found, that even among commercial tools, a very small percentage of them provide reverse engineering functionality. In General, reverse engineering is supported by object-oriented SCADA packages, which have little to no relation to the sphere of the control and embedded software.

# requirements for the software module

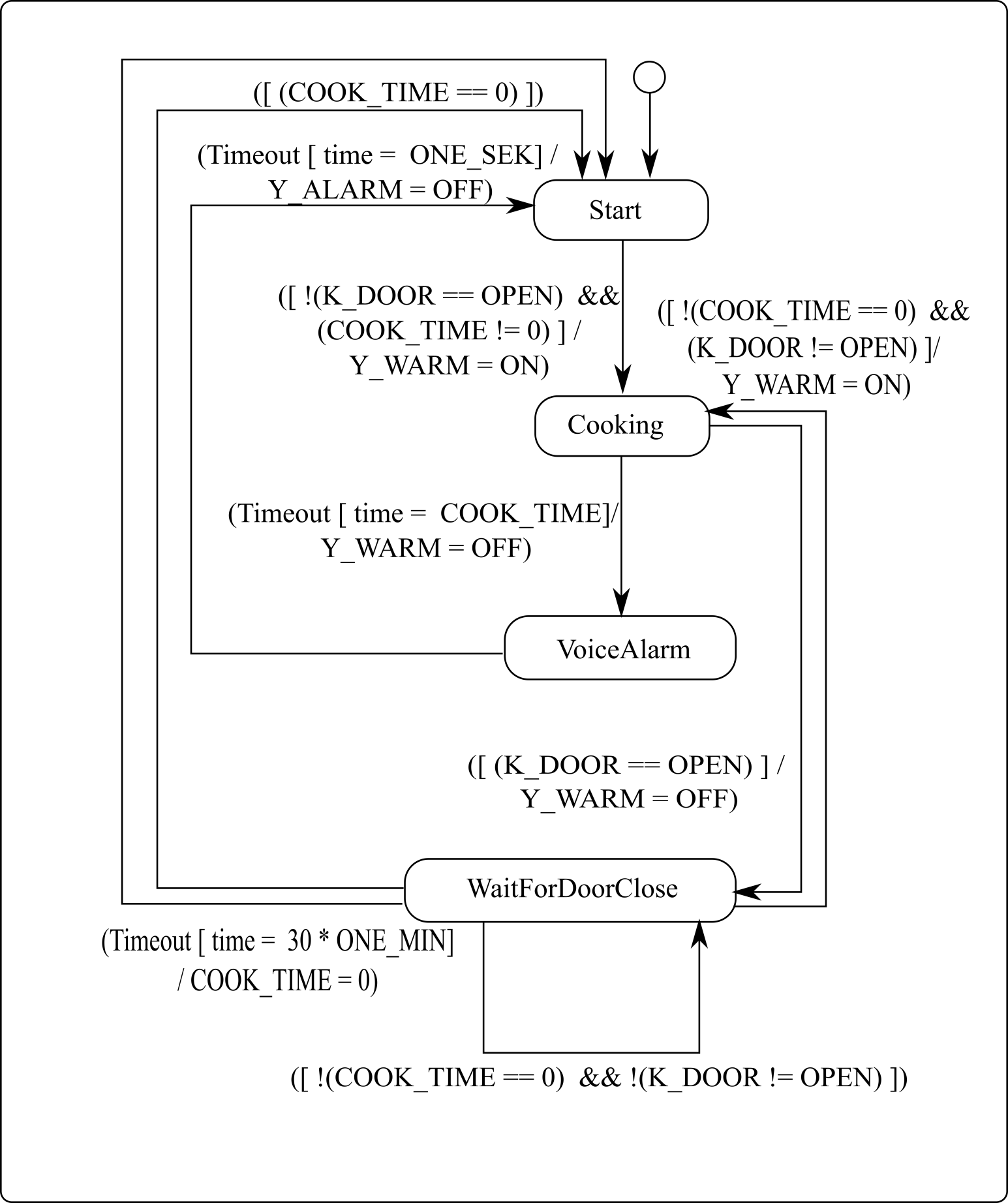
Requirements to the software module have been formulated based on the analysis results. The module needs to implement the following capabilities:

* Diagram modification;
* Automatic layout;
* Generation of the following diagram types:  
  Data communication diagram;  
  Process control diagram;  
  Process states diagram;
* Saving diagrams to the file;
* Dividing diagrams into parts and saving them to different files if parts do not have any shared nodes.

# Process diagrams

## Process states diagram

For the process states diagram, we use UML statecharts diagram [6], [7]. An example of such diagram is shown in Fig. 1.

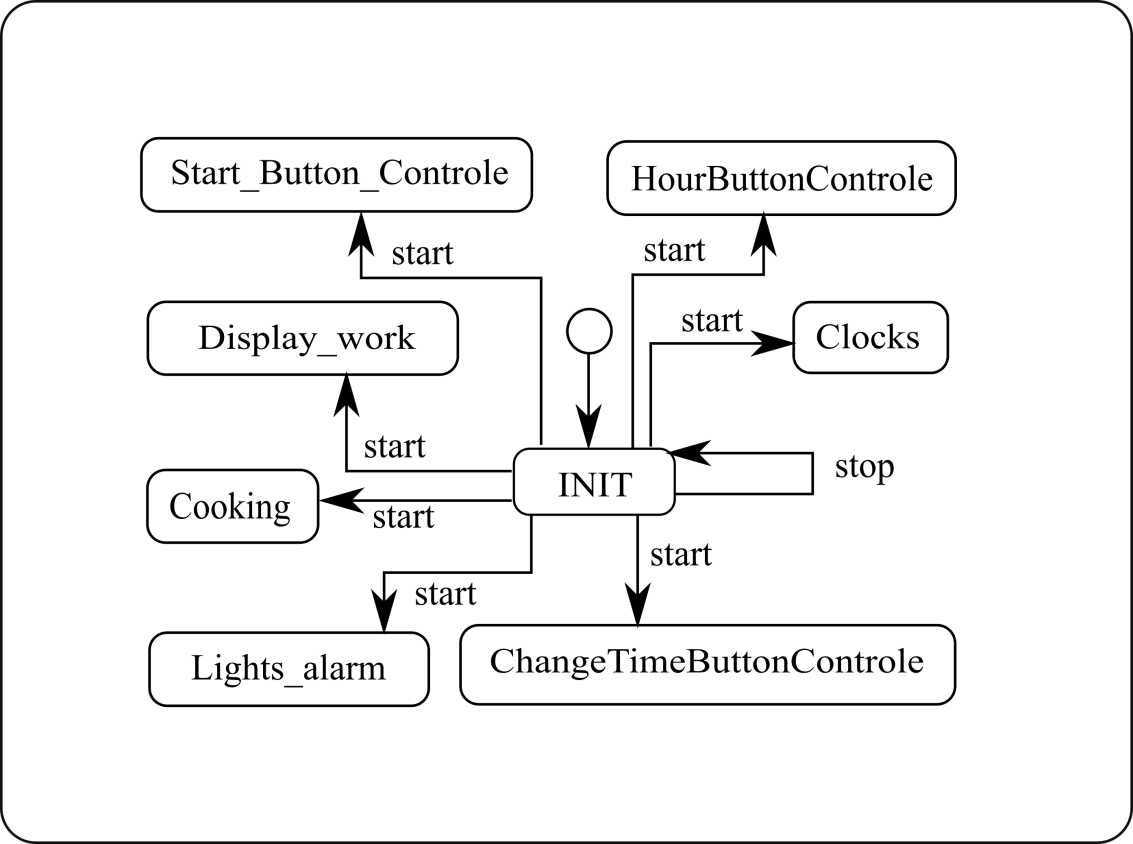


1. Example of a process states diagram

Names of process states are inside the nodes. They are connected by arrows, with conditional expressions above them in brackets. Actions executed upon transition are marked after the slash.

## Process control diagram

We have analyzed graphic notations for diagrams to develop a process control diagram. Our diagram is based on the statecharts diagram [6], [7], from where we have adopted a start node labeling, and the activity UML diagram, from where we adopt the shape for nodes and marking. An example of such diagram is shown in Fig. 2.



1. Example of a process control diagram

## Data communication diagram

A graphical notation has been developed for the data communication diagram. The notation is based on the statecharts diagram [6], [7] (where we have adopted a shape for process nodes) and on the activity UML diagram (where we have adopted a shape for variable nodes and arrows labeling). An example of such diagram is shown in Fig. 3.

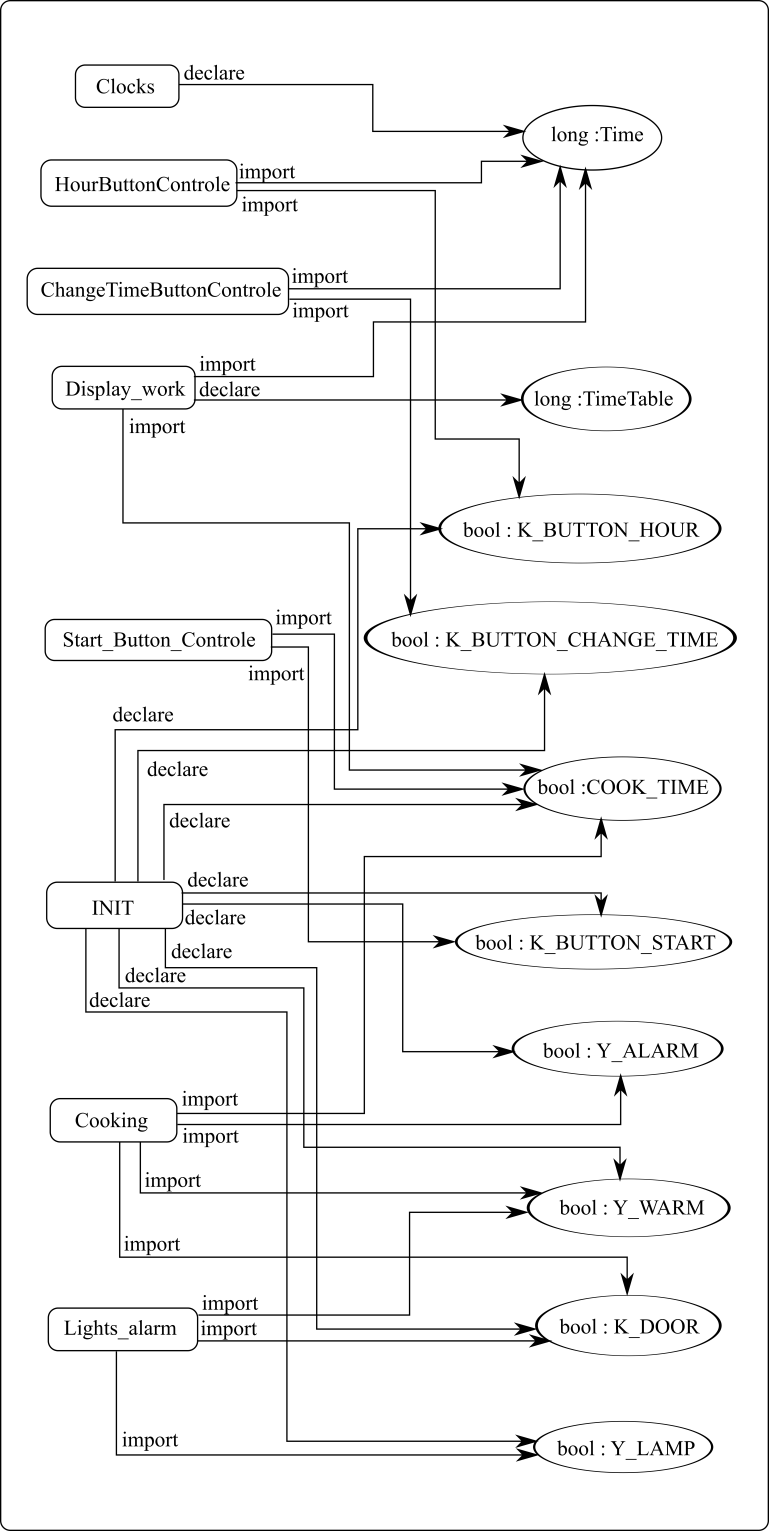
Processes names are marked in round-corner nodes, names of variables are written in elliptical diagram nodes. Process nodes are connected with variable nodes by arrows, and connection type is labeled above them (e.g. imported variables are marked as ‘import’, declared variables are marked as ‘declare’).

# Graph storage formats

To choose the graph storage format for output diagram files for the software module, we have performed analysis of graph representation formats. The result of analysis is shown in the Table 2. The following formats were considered: GraphML [9], [11], GV [15], XGML [12], GML [16], Node list, Edge list, PAJEC [13], LEDA [14], [19], [20], TLP [17], [18], GW [14], [19], [20] and GEXF [21] – [23].

An evaluation criterion like an ability to save coordinates is important because it makes possible to save graph layout (by a third-party layout tool) and modifications of the diagram after closing the file. We also analyzed notation criteria to understand which formats may storage graphs in our notations. The last criterion was a list of tools that are able to use the format [8], [10].

An ability to convert one file format to another is shown in Fig. 4. We found that here are two families of graph file formats, inside which conversion is possible.



1. Example of a data communication diagram
2. Graph representation formats

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Format name | Format features | | | | |
| Saving coordinates | Right node shape | Oriented graph | Edge labels | Tools that use format |
| GraphML | + | + | + | + | Gephi, Igraph, OGDF, Yed, NetworkX |
| GV | - | + | + | + | GraphViz, Gephi, Igraph, OGDF, ZGRViewer, NetworkX |
| XGML | + | + | + | + | Yed |
| GML | + | + | + | + | Gephi, Igraph, OGDF, Yed, NetworkX, Tulip, LEDA |
| Node list | - | - | + | - | Yed, NetDraw |
| Edge list | - | - | + | - | OGDF, Yed, NetworkX, NetDraw |
| PAJEC | + | + | + | + | Gephi, Igraph, NetworkX, NetDraw |
| LEDA | + | + | + | + | Igraph, OGDF, NetworkX, LEDA |
| TLP | + | + | + | + | OGDF, Gephi, Tulip |
| GW | + | + | + | + | LEDA |
| GEXF | + | + | + | - | Gephi, OGDF, NetworkX, Tulip |

GraphML [9], [11] and GML [16] file formats look most suitable for use in this work. We have made an analysis of these file formats, the result of which is shown on the Table 3.

1. Comparison of GraphML and GML file formats

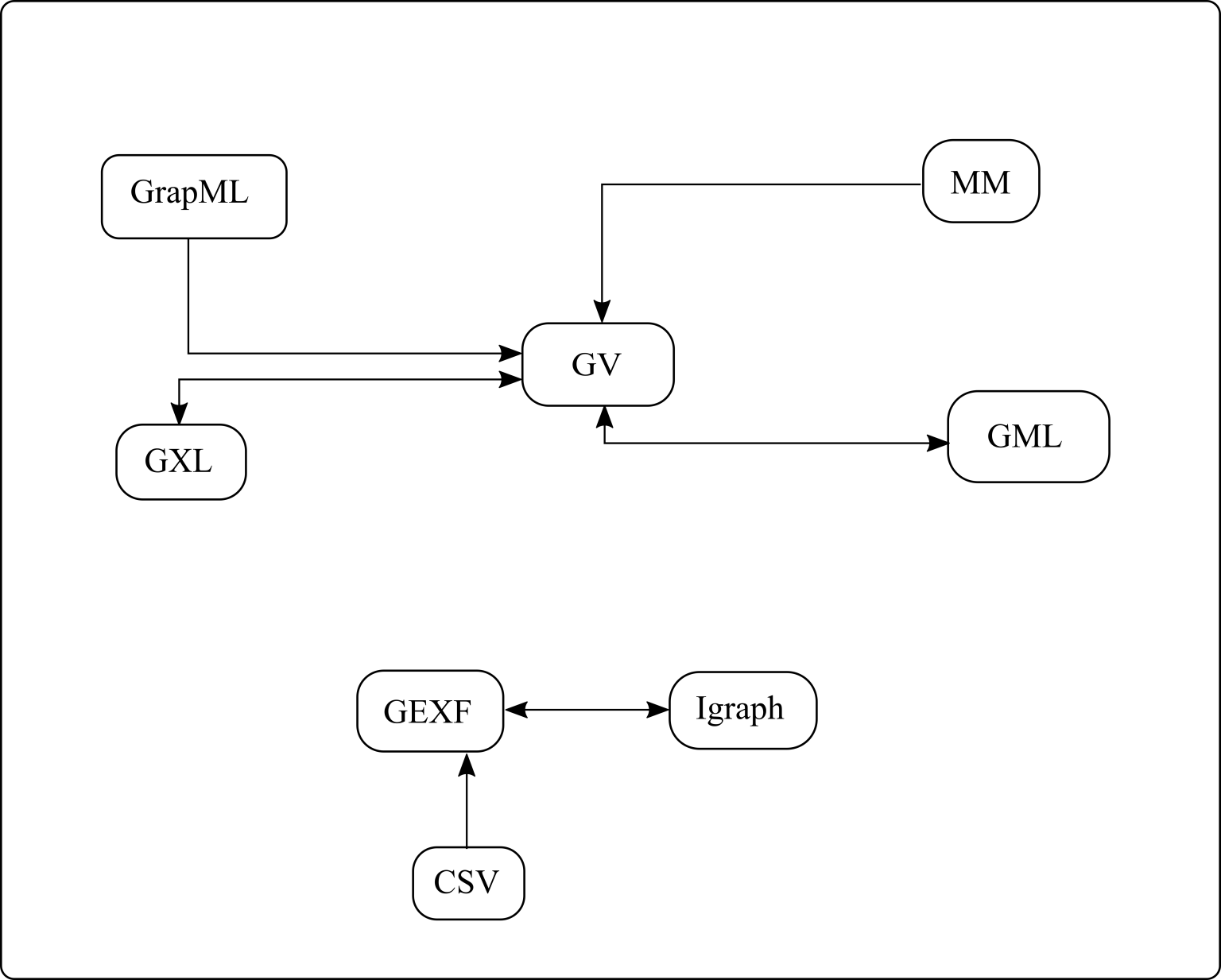
| Comparison criterion | Name of the file format | |
| --- | --- | --- |
| GML | GraphML |
| File size | x | 2 \* x |
| Easy to read by human | + | - |
| Can store links | - | + |

GML has a human-readable syntax; GraphML has a XML-like syntax. GraphML diagrams have twice the size of the same GML diagrams. Also GraphML allows storing links, and that is important, because in our diagrams in the each process node we want to have a link to the state diagram for that process.

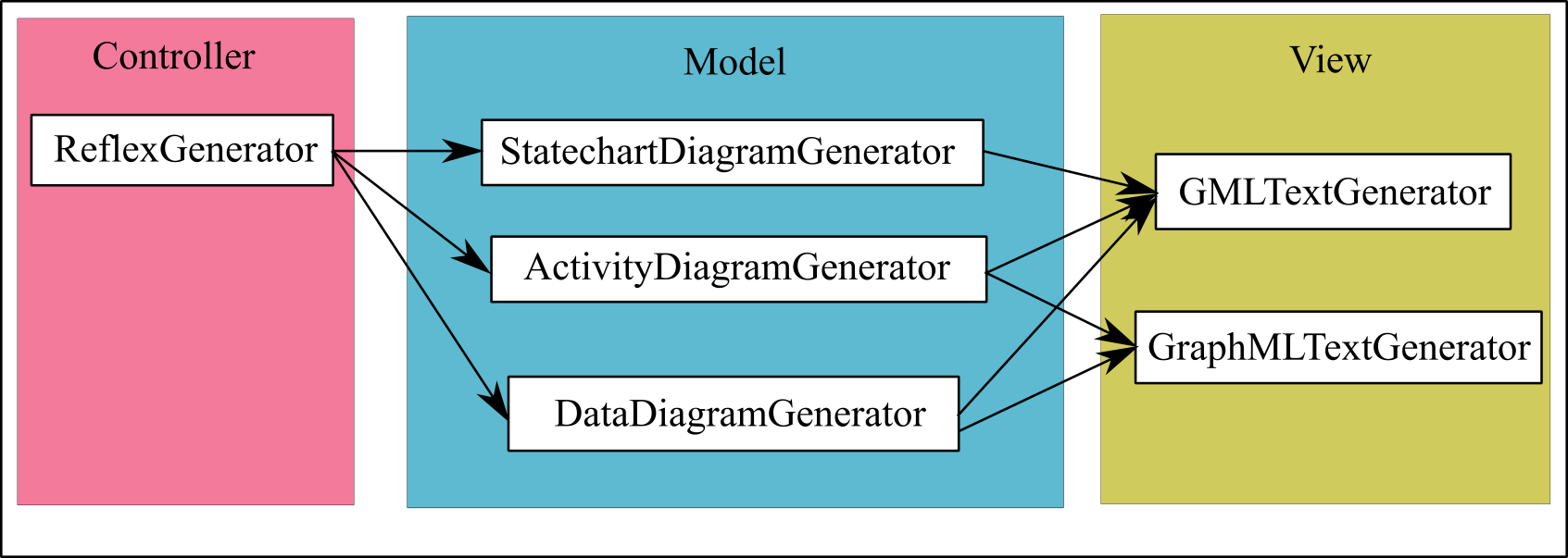
As a result, GML has been chosen for the process state diagram because of small sizes of files in that format (the module generates such diagram for the every process in the program). GraphML has been chosen for process communication and control diagrams, because it allows storing the links in graph nodes.

# Architecture of the software module

The developed module uses the Model–view–controller (MVC) software design pattern (with the active Model), which enables quick modification of the architecture. Interconnection diagram of the software module parts is shown in Fig. 5. Firstly, the controller starts the model generator for process state diagrams, then, the view part creates GML files of those diagrams. After that, the controller starts the model generators for process communication and control diagrams. Then it uses a graph separator module to divide models of the diagram in case they have independent parts. After that the diagram generators create GraphML files of diagrams with links in each process node to the state diagram for that process (which was generated before).



1. An ability to convert one file formats



1. Interconnection diagram of the software module parts

# Implementation of the sofware module

The module has been implemented as an Eclipse plugin (to simplify integration in the Reflex IDE) using Xtext and Xtend technologies. After opening the Reflex IDE and writing the source code, user can click the save file button, and then process data communication, control and states diagrams will be automatically generated. We recommend opening them with the yEd [10] tool that supports automatic graph layout. The sequence of user actions is shown in Fig. 6.

For the graph separator module we used such algorithm: firstly, all vertices (that have not been visited) are added to a queue. We sequentially get one node from the queue. After that:

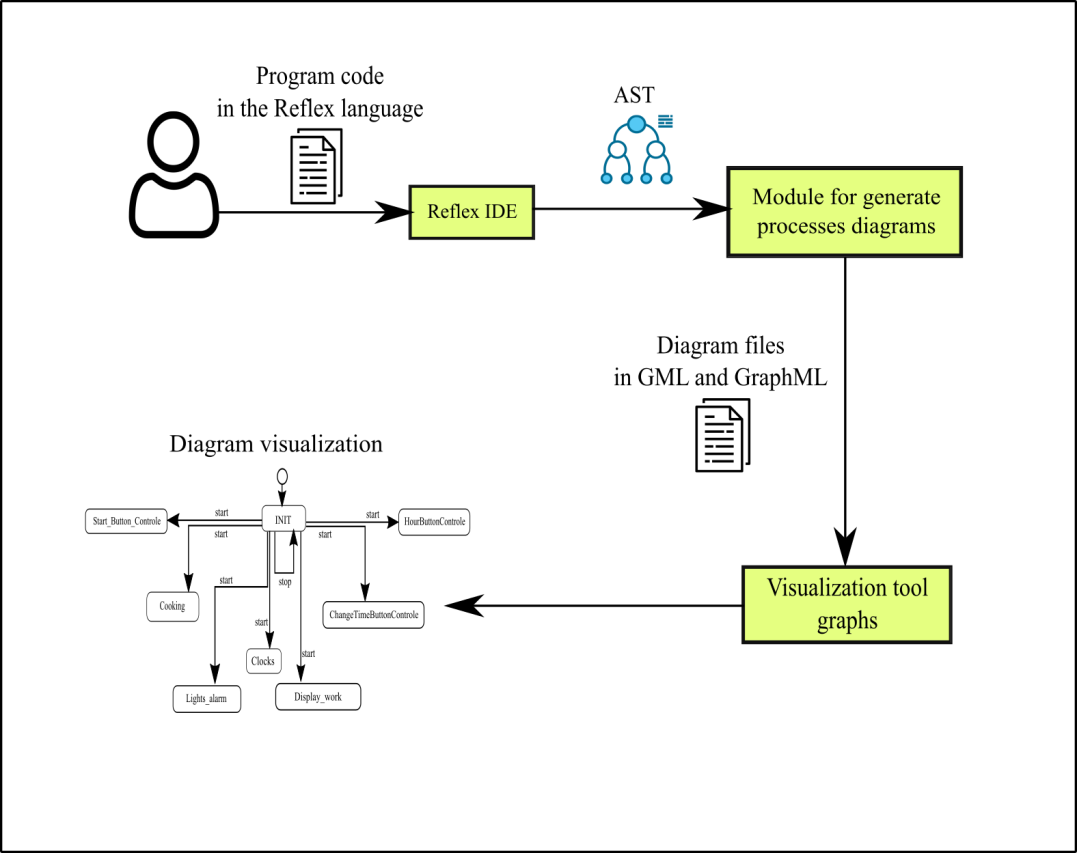
* For vertex that has not been visited: we mark vertex as visited and add it to the current component. Get all children for node and add them to the queue.
* For vertex that has been visited we check the number of component that included him. If that is not a current component, then:
  + If component number is less than current component number, then number of the current component will be replaced with the number of the component that was found and components will be merged.
  + If current component number is less than found component number, then components will be merged.

# Conclusion

We have developed an approach to creating graphical automatic documentation for domain-specific languages. The main idea of our approach is in the using available external graph visualization tools for rendering diagrams which were created by the developed software module.

We have analyzed available tools for creating diagrams from the source code for common programming languages and graph storage formats for choosing the most suitable file format for that work. Also process-oriented programming language Reflex was analyzed from the side of the main Reflex features for visualization.

As a result, a software module for building process diagrams according to the specification in the Reflex language was created. The module creates processes data communication, control and states diagrams. It automates the creation of the graphical documentation and simplifies the support of projects in the Reflex language. Practical testing has shown that using the module decreases the time for creating processes diagrams (from a few hours to a couple of seconds) and guarantees that there are no errors caused by human factor.



1. Model usage scheme

Therefore, our approach was successfully tested on the Reflex language and may be used for other programming languages.

##### References

1. A. Rozov, V. Zyubin and D. Nefedov, "Hyperprocess-Based Approach for Embedded Microcontroller Programming", Vestnik NSU. Series: Information Technologies, vol. 15, no. 4, pp. 64-73, 2017. Available: 10.25205/1818-7900-2017-15-4-64-73.
2. A. Rozov, T. Lyakh, D. Krasnov and E. Sanzhiev, "A Practical Application of IndustrialC Language in Vacuum Deposition Plant Automation", Vestnik NSU. Series: Information Technologies, vol. 15, no. 3, pp. 90-99, 2017. Available: 10.25205/1818-7900-2017-15-3-90-99.
3. A. Rozov and V. Zyubin, "Process-oriented programming language for MCU-based automation", in International Siberian Conference on Control and Communications (SIBCON), 2013, pp. 1-4.
4. T. Lyakh and V. Zyubin, "The Reflex language usage to automate the large solar vacuum telescope", 17th International Conference of Young Specialists on Micro/Nanotechnologies and Electron Devices (EDM), pp. 137-139, 2016.
5. V. Zyubin, T. Lyakh and A. Rozov, "Reflex language: a practical notation for cyber-physical systems", System Informatics, no. 12, 2018. Available: 10.31144/si.2307-6410.2018.n12.p85-104.
6. D. Harel, "Statecharts: a visual formalism for complex systems", Science of Computer Programming, vol. 8, no. 3, pp. 231-274, 1987. Available: 10.1016/0167-6423(87)90035-9.
7. G. Lüttgen, "Modeling and verification using UML Statecharts. By Doron Drusinsky. Published by Newnes Publishers, 2006. ISBN: 0-7506-7617-5, 306 pages. Price £39.99. Hard Cover.", Software Testing, Verification and Reliability, vol. 18, no. 3, pp. 189-190, 2008. Available: 10.1002/stvr.377.
8. R. Tamassia, Handbook of graph drawing and visualization. CRC Press.
9. V. Kasyanov, Grapg representation language GraphML: basic tools 1. 2012.
10. t. yWorks, "yEd Graph Editor", yWorks, the diagramming experts, 2020. [Online]. Available: https://www.yworks.com/products/yed. [Accessed: 27- May- 2020].
11. "The GraphML File Format", Graphml.graphdrawing.org, 2020. [Online]. Available: http://graphml.graphdrawing.org. [Accessed: 27- May- 2020].
12. "XGML", Docs.yworks.com, 2020. [Online]. Available: https://docs.yworks.com/yfiles/doc/developers-guide/xgml.html. [Accessed: 27- May- 2020].
13. V. Batagelj and A. Mrvar, "Program Package: Pajek - Spider / Pajek to EPS", Vlado.fmf.uni-lj.si, 2020. [Online]. Available: http://vlado.fmf.uni-lj.si/pub/networks/pajek/doc/draweps.htm. [Accessed: 27- May- 2020].
14. "LEDA Guide: Native File Format for Graphs", Algorithmic-solutions.info, 2020. [Online]. Available: http://www.algorithmic-solutions.info/leda\_guide/graphs/leda\_native\_graph\_fileformat.html. [Accessed: 27- May- 2020].
15. "The DOT Language", Graphviz - Graph Visualization Software, 2020. [Online]. Available: https://graphviz.gitlab.io/\_pages/doc/info/lang.html. [Accessed: 27- May- 2020].
16. "GML", Docs.yworks.com, 2020. [Online]. Available: https://docs.yworks.com/yfiles/doc/developers-guide/gml.html. [Accessed: 27- May- 2020].
17. D. Auber and P. Mary, "Tulip software graph format (TLP)", Tulip.labri.fr, 2020. [Online]. Available: https://tulip.labri.fr/TulipDrupal/?q=tlp-file-format. [Accessed: 27- May- 2020].
18. D. Auber and P. Mary, "LGPL", Tulip.labri.fr, 2020. [Online]. Available: https://tulip.labri.fr/TulipDrupal/?q=licence. [Accessed: 27- May- 2020].
19. "AlgoSol - LEDA Graphs for Java", Algorithmic-solutions.com, 2020. [Online]. Available: https://algorithmic-solutions.com/index.php/products/leda-graphs-for-java. [Accessed: 27- May- 2020].
20. "AlgoSol - LEDA for C++", Algorithmic-solutions.com, 2020. [Online]. Available: https://algorithmic-solutions.com/index.php/products/leda-for-c. [Accessed: 27- May- 2020].
21. S. Heymann, "GEXF File Format", Gephi.org, 2020. [Online]. Available: https://gephi.org/gexf/format/schema.html. [Accessed: 27- May- 2020].
22. G. Woodhull, J. Ellson, E. Gansner, E. Koutsofios and S. North, "Graphviz and dynagraph – static and dynamic graph drawing tools", Citeseerx.ist.psu.edu, 2020. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.96.3776. [Accessed: 27- May- 2020].
23. S. Heymann et al., "GEXF 1.2 draft Primer", Gephi.org, 2020. [Online]. Available: https://gephi.org/gexf/1.2draft/gexf-12draft-primer.pdf. [Accessed: 27- May- 2020].