*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, so, we need documentation to the source code. Auto documentation tool may decrease the time for creating diagrams and protect them from human’s mistakes.

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 of 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 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 can 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 algorithms 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, which is actively developed in the Institute of Automation and Electrometry of the Siberian Branch of the Russian Academy of Sciences [2]. 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 human factor.

# Purpose of work and tasks

The purpose of work was to develop a software module for building process diagrams according to the specification in the Reflex language. To achieve that goal, we have set several tasks. Firstly, to analyze: specific features of process-oriented programming language Reflex, existing tools for generating diagrams from source code, and types of diagram’s notations which is used for code analysis. To develop the module (to define requirements for the software module, to develop diagram notation for showing communications between processes, to choose the file format for saving diagrams). Then, to develop the architecture of the software module, implement it and test.

# Specific Reflex language features.

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

# Analysis of existing tools for generating diagrams from code

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

1. tools for generate diagrams by code for common languages

| Tool’s name | Tool’s abilities | | | |
| --- | --- | --- | --- | --- |
| Code generation | ***Modify diagrams*** | ***Hiding diagram’s components*** | Dynamic building |
| Class Designer for Visual Studio | - | + | + | + |
| Astah UML | + | + | ? | ? |
| MagicDraw | + | + | ? | + |
| Software Ideas Modeler | + | + | + | + |
| BOUML | + | + | + | - |
| Visual paradigm | + | + | ? | ? |
| Rational Rose | + | + | + | + |
| Enterprise Architect | + | + | ? | ? |
| IntelliJ Idea | - | + | + | + |
| Sybase PowerDesigner | + | + | + | - |
| NetBeans | - | + | - | - |
| Lab view | + | + | - | ? |
| Altova UModel 2008 | + | + | ? | ? |

By analyzing software development tools for control algorithms 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 control and embedded software.

# requirements for software module

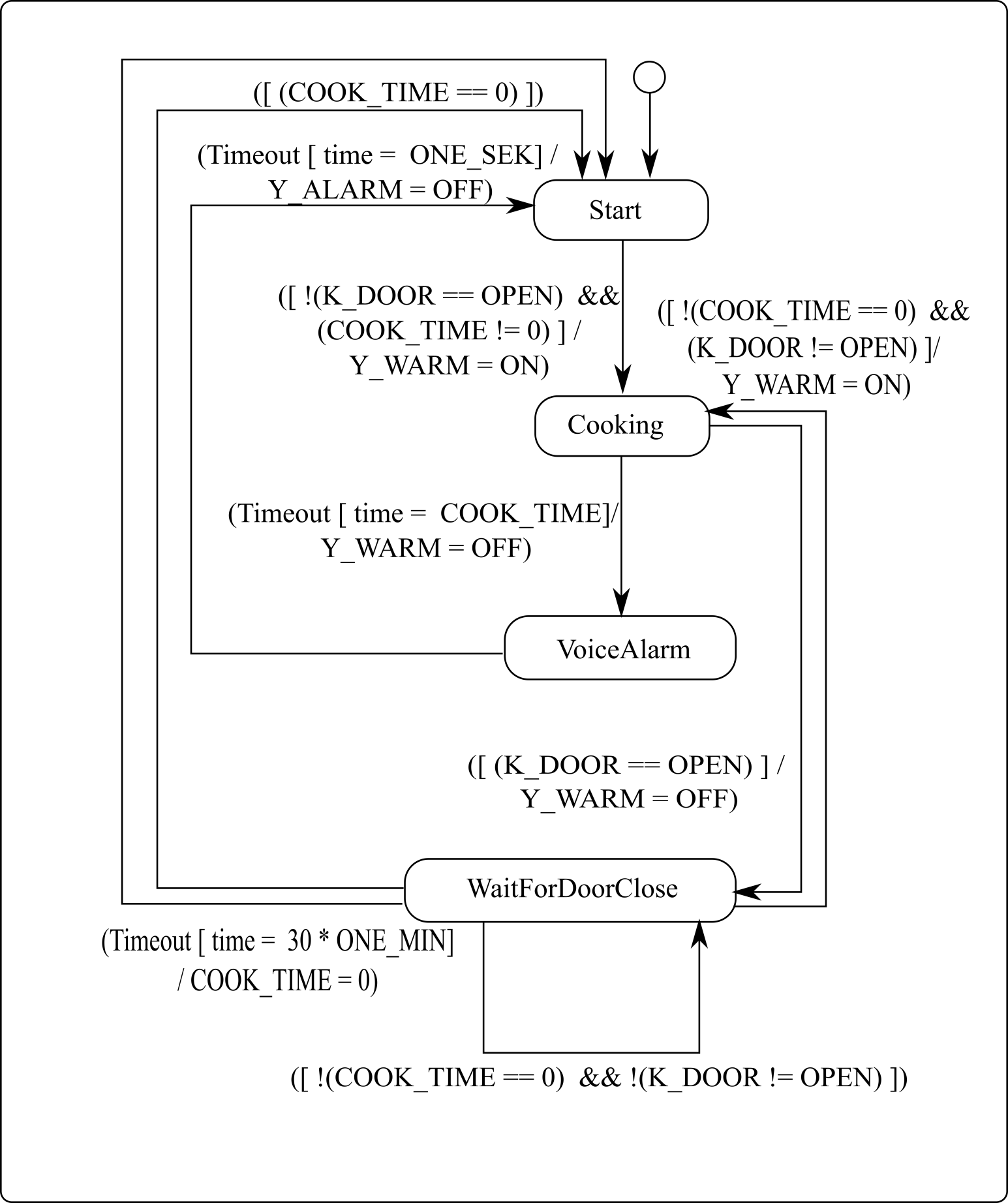
Requirements for software module was formulated based on the analysis results. The module has to implement the following capabilities:

* Diagram modification;
* Automatic layout;
* Generation of the following diagram types:
  + Data communication diagram;
  + Process control diagram;
  + Processes states diagram;
* Saving diagrams to the file;
* Dividing diagrams into parts and saving them to different files, if parts haven’t got any shared nodes.

# Process diagrams

## Process states diagram

For process states diagram we use UML statechart diagram [1][3]. An example of such diagram is shown on the Fig. 1.



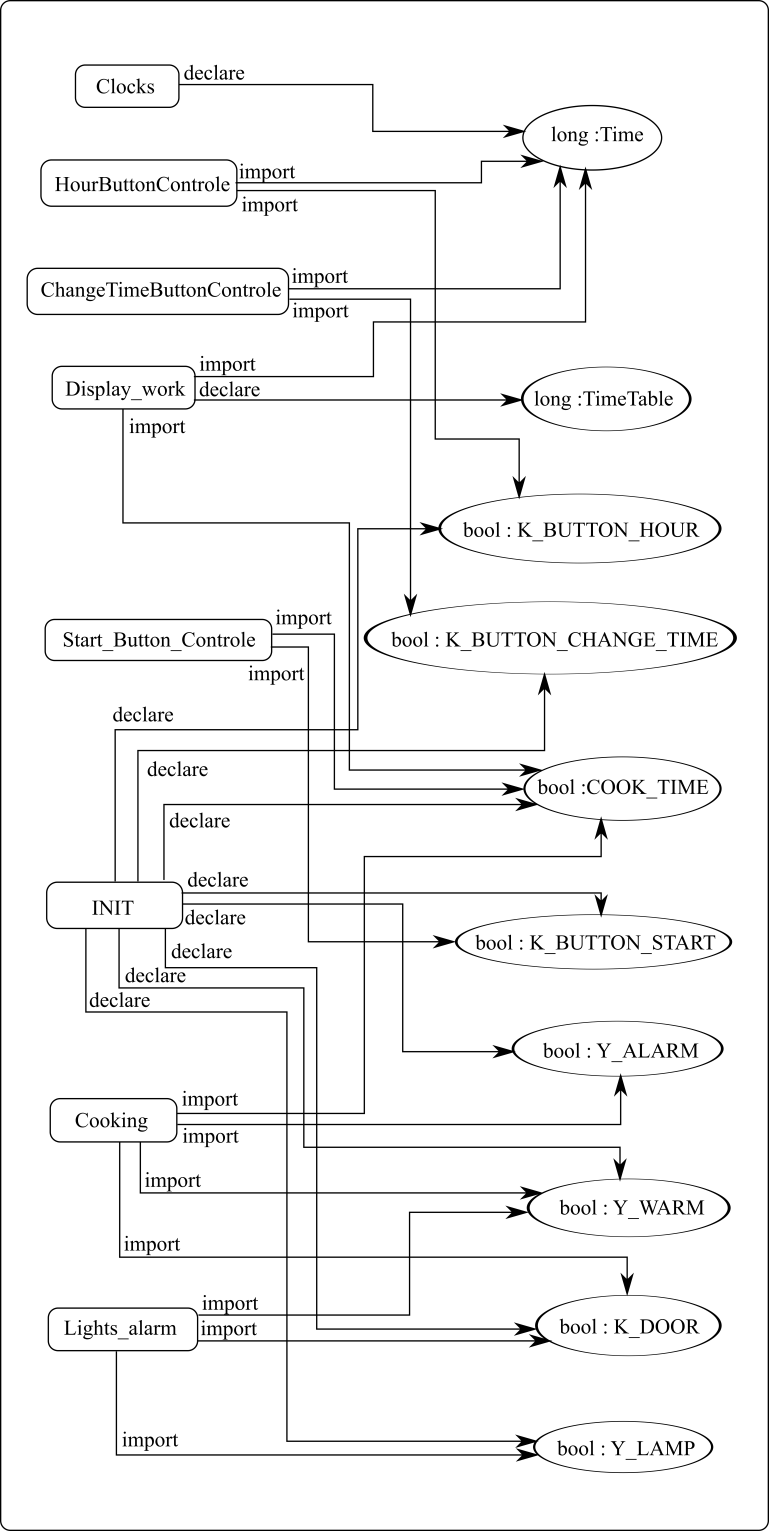
1. Example of a process state diagram

Names of process states are inside the nodes. They are connected by arrows, with conditional expressions (switch conditions) above them in brackets. Actions which will be done with switching to that state are marked after the slash.

## Data communication diagram

A graphical notation has been developed for data communication diagram. Notation was based on the UML state machine diagram [1][3] (where we have borrowed a process node shape) and on the activity UML diagram (where we have borrowed a variable node shape and arrows labeling). An example of such diagram is shown on the Fig. 2.

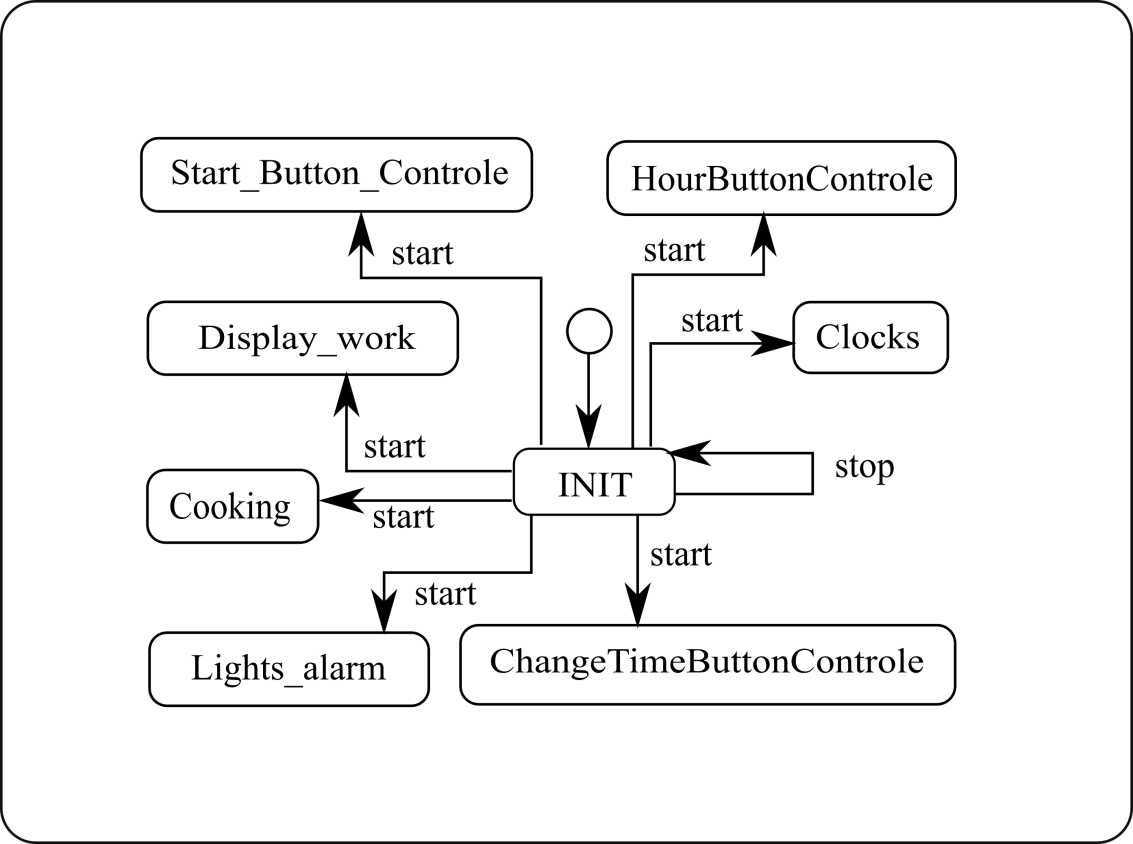
Processes names are marked in round-corner nodes, names of variables are written in elliptical diagram nodes. Process nodes are connected with variable nodes by arrows, and connection type is labeled above them (e.g. imported variables are marked as ‘import’, declared variables are marked as ‘declare’).



1. Example of a data communication diagram

## Process control diagram

We have been analyzing graphic notations for diagrams to develop a process control diagram. Our diagram is based on the UML state machine diagram [1][3], from where we have borrowed a start node labeling, and the activity UML diagram, from where we adopt the node shape and marking. An example of such diagram is shown on the Fig. 3.



1. Example of a process control diagram

# Graph storage formats

To choose the graph storage format for module’s output diagram files, we have perfomed analysis of graph representation formats. The result of analysis is shown in the Table 2. The following formats were considered: GraphML [8][9], Gv [10], Xgml [11], Gml [12], Node list [17], Edge list [17], Pajek [13], Leda [14][18][19], TLP [15][16], Gw [14][18][19] and GEXF[20][21].

1. Graph representation formats

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Format name | Format’s abilities | | | | |
| 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 |
| Pajek | + | + | + | + | Gephi, Igraph, NetworkX, NetDraw |
| Leda | + | + | + | + | Igraph, OGDF, NetworkX, LEDA |
| TLP | + | + | + | + | OGDF, Gephi, Tulip |
| Gw | + | + | + | + | LEDA |
| GEXF | + | + | + | - | Gephi, OGDF, NetworkX, Tulip |

Such an evaluation criterion as the ability to save coordinates is important because it makes possible to save graph layout (by a third-party layout tool) and diagram’s modifications 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 [4][5][6].

An ability to convert one file format to another is showed on the Fig. 4. We found that here is a two families of graph file formats, inside which conversion is possible.



1. Example of a process managing connection diagram (толщина рамок, размер шрифта)

GraphML and GML file formats look most suitable for use in this work. We made a more detailed analysis of these file formats, the result of which is showed 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 format has a language with a human-readable syntax; GraphML format has an language with XML-like syntax. GraphML diagrams have twice the size of the same GML diagrams. Also GraphML allows to store links, and that’s important, because in each processes diagram node we want to have a link to the state diagram for that process.

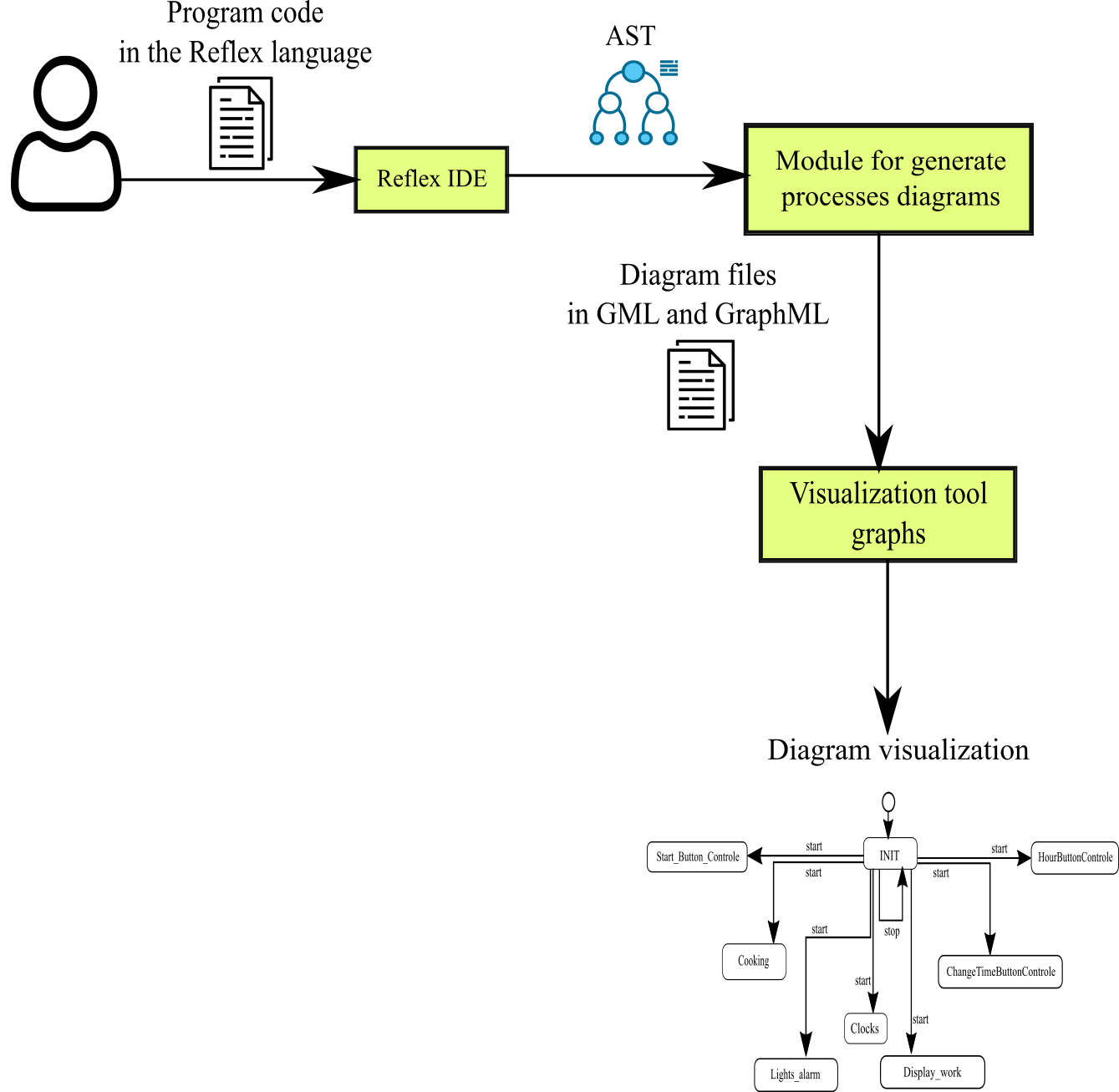
As a result, GML was (has been?) chosen for the process state diagram because of small file’s sizes (the module would generate such diagram for every process in the program). GraphML was chosen for processes communication diagrams by data and managing, because it allows to store the links in graph’s nodes.

# Module’s architecture

The developed module uses the Model–view–controller (MVC) software design pattern, which enables quick modification of the architecture. Firstly, the controller starts the model generator for processes state diagrams, then the view part creates GML files of those diagrams. After that, the controller starts the model generators for diagrams of connection between processes. Then it uses a graph separator module to divide diagram’s models if they have independent parts. After that the diagram generators create GraphML files of diagrams with links in each process diagram node to a state diagram for that process (which was generated before).

# Module’s implementation

The module was implemented as an Eclipse plugin (to simplify integration in the Reflex IDE) using Xtext and Xtend technologies. After opening the Reflex IDE and writing code, the user/one can click the save file button, and then processes diagrams of connection by data and managing, and state diagram will be automatically generated. We recommend opening them with the yEd [7] tool that supports automatic graph layout. The sequence of user actions is showed on the Fig. 5.



1. User steps scheme

# The conclusion

As a result, a software module for building process diagrams according to the specification in the Reflex language was created. The module creates processes diagrams of connection by data and managing, and state diagram. 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 decreased 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 factors.

##### References

1. D. Harel “Statecharts: A visual formalism for complex systems. Science of computer programming”, 8(3), 231-274, 1987.
2. V. E. Zyubin, “Process-orinted programming: Tutorial”. Novosibirsk state university. (in Russian), 2011.
3. UML state machine (2020). Available at: https://en.wikipedia.org/wiki/UML\_state\_machine (accessed 27 April 2020).
4. Habr: Visualization of large graphs for the smallest (2019). Available at: https://habr.com/ru/company/ods/blog/464715/ (in Russian), (accessed 27 April 2020).
5. H. Gibson, J. Faith, P. Vickers, “A survey of two-dimensional graph layout techniques for information visualization” Information Visualization, 2012. Available at: http://leonidzhukov.net/hse/2018/sna/papers/gibson2013 (accessed 27 April 2020).
6. M. Chimani, C. Gutwenger, M. Jünger, G. W. Klau, K. Klein, P. Mutzel. The Open Graph Drawing Framework (OGDF). Chapter 17 in: R. Tamassia (ed.), Handbook of Graph Drawing and Visualization, CRC Press, 2014.
7. yWorks: yEd Graph Editor (2020). Available at: https://www.yworks.com/products/yed (accessed 27 April 2020).
8. The GraphML File Format (2019). Available at: http://graphml.graphdrawing.org (accessed 27 April 2020).
9. V. N. Kasyanov, “V. N. Kasyanov GRAPH REPRESENTATION LANGUAGE GRAPHML: BASIC TOOLS 1” (2012). Available at: https://docplayer.ru/31018473-V-n-kasyanov-yazyk-predstavleniya-grafov-graphml-bazovye-sredstva-1.html (accessed 27 April 2020).
10. The DOT Language. Available at: https://graphviz.gitlab.io/\_pages/doc/info/lang.html (accessed 27 April 2020).
11. yFiles for Java Developer's Guide: XGML : Chapter 9. Input and Output (2016). Available at: https://docs.yworks.com/yfiles/doc/developers-guide/xgml.html (accessed 27 April 2020).
12. yFiles for Java Developer's Guide: GML : Chapter 9. Input and Output (2016). Available at: https://docs.yworks.com/yfiles/doc/developers-guide/gml.html (accessed 27 April 2020).
13. V. Batagelj, A. Mrvar, “Pajek: Exporting pictures from Pajek to EPS”. Available at: http://vlado.fmf.uni-lj.si/pub/networks/pajek/doc/draweps.htm (accessed 27 April 2020).
14. Native File Format for Graphs: Algorithmic Solutions Software GmbH (2006). Available at: http://www.algorithmic-solutions.info/leda\_guide/graphs/leda\_native\_graph\_fileformat.html (accessed 27 April 2020).
15. D. Auber, P. Mary, “Tulip: Tulip software graph format (TLP)”. Available at: https://tulip.labri.fr/TulipDrupal/?q=tlp-file-format (accessed 27 April 2020).
16. D. Auber, P. Mary, “Tulip: LGPL: License”, Available at: https://tulip.labri.fr/TulipDrupal/?q=licence (accessed 27 April 2020).
17. A. Prokhorov, N. Larichev, “Computer visualization of social networks”, 2020, Available at: https://compress.ru/article.aspx?id=16593#NetDraw (accessed 27 April 2020).
18. Algorithmic solutions software GMBH: LEDA Graphs for Java / Algorithmic Solutions Software GmbH (2020). Available at: https://algorithmic-solutions.com/index.php/products/leda-graphs-for-java (accessed 27 April 2020).
19. Algorithmic solutions software GMBH: LEDA Graphs for C++ / Algorithmic Solutions Software GmbH (2020). Available at: https://algorithmic-solutions.com/index.php/products/leda-for-c (accessed 27 April 2020).
20. S. Heymann, M. Bastian, M. Jacomy, C. Maussang, A. Rohmer, J. Bilcke, A. Jacomy, “GEXF File Format”. Available at: https://gephi.org/gexf/format/schema.html (accessed 27 April 2020).
21. S. Heymann, M. Bastian, M. Jacomy, C. Maussang, A. Rohmer, J. Bilcke, A. Jacomy, “GEXF 1.2draft Primer”. Available at: https://gephi.org/gexf/1.2draft/gexf-12draft-primer.pdf (accessed 27 April 2020).