Program generator architecture

Emanuil Markov

Technical University of Sofia

Sofia, Bulgaria

markov@takt-ltd.com

Authors Name/s per 2nd Affiliation (Author)

line 1 (of Affiliation): dept. name of organization

line 2: name of organization, acronyms acceptable

line 3: City, Country

line 4: e-mail address if desired

**Abstract**—This document describes an author designed architecture and applications of program generator system.

Keywords-component; program generation; real-time; control; automation

# Introduction

Automated program generation is not a new concept, although it has seen relatively little recognition until recently. Recent changes in the way software is developed may cause an upsurge in the use of program generators as a way of bringing enterprise software to market extremely quickly.

A program generator reads the project meta-data and produces well-formed code to a specific set of design patterns.

Program generators could be divided by two main classes – generators that generate source code that should be compiled and run on the target platform and generators that generate configurations that are executed by a configuration interpreter.

In both cases there a support from the target platform should be available, whether it is a compiler for the generated code or interpreter, precompiled to work on the target platform.

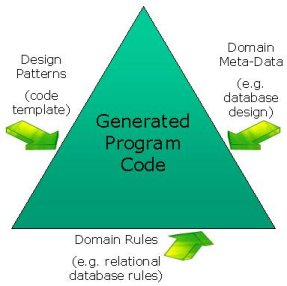
Program generators can also make a project very agile. A change in implementation is simply a different rendering of the same meta-data. Anything which is repetitive can be automated. Given the correct conditions, a lot of code can be automatically generated - the programmer is then free to "fill in the gaps".

For code to be generated, the following three vital areas must be 100% predictable and well understood (fig. 1) [1]:

1. The Design Patterns (the template to which the code will be produced)

2. The Domain Meta-Data (the topology that should be modeled in the code, usually augmented with extra data provided by the developer)

3. The Domain Rules (the rules that dictate the structure and behaviour of the domain meta-data. This area is normally encapsulated in the generator program itself)



1. Program generation rules

The data itself can be various and unpredictable, as long as it fits within the above constraints.

Configuration generators could be even more rapid for development and the user is not necessary obliged to know the programming language that they are written. On the other hand they have a strict structure that has to be followed with no exceptions.

# Advantages and Disadvantages of Program Generators

Some advantages of auto-generated code are:

* Ultra-consistent (machine generated)
* Changes the way programmers work: code tends to be much cleaner and simpler, as it only needs to do what is required. If requirements change later, it is enough modify the generator and produce a new version of the source code. The same is valid for the configuration generators.
* Stable and bug-free, "works first time" (the debugging takes place when writing the generator itself - hence the bulk of the debugging work has taken place BEFORE the project begins)
* Produced very quickly (just point the generator at the problem domain, and produce a brand-new, guaranteed up-to-date API)
* Customizable (assuming the source code for the generator itself is available, the code or configuration that will be generated can be easily customized)
* Programmers are free to concentrate on the areas of development that deserve their brainpower
* Code mentor (as the generated code is well written and 100% consistent, programmers are more likely to copy the style of this code and learn from it. So all project code is more likely to be consistent and very clear, despite being produced at an rapid rate)

Some disadvantages of auto-generated code:

* The generator must be written first
* Only applicable given a specific set of conditions
* There will always be some code that needs to be hand-crafted. The actual amount varies from project to project. Usually, the generated code acts as "support" for this hand-crafted code, i.e. an API, or a library supporting the specific area.
* For database code, the database must be well formed (i.e. properly designed and normalized). Generators generally do not work well with databases that have special, "unique" design features.

The last item can also be used to the project's advantage, as long as the database is being designed especially for this project. In effect, the code generator can be used as a database design validator: the design should be improved until a proper API can be generated. The database will be much better designed as a result. This will save a LOT of development time later on. However, if the database was created for an older project, and contains real data, then it might not be possible to use a code generator. In this instance, the generator could be customized to match with the specifics of the database; or the "traditional" route could be taken, and the API source hand-coded from scratch.

# PrGen Architecture

The program generator described here is targeted for configuring distributed real-time control systems. Each node of the distributed system is described separately. A graph representation of the control algorithms is selected as most of the control engineers will be able to create and configure the system.

There are two types of graphs defined:

* Signal Flow Graph (SFG) – a graph model built from functional blocks, very similar to Simulink®. It is mainly used to model the continuous part of the system (the dataflow [3]), to handle the I/O and communication drivers and to calculate predicates determining the behavior of the node.
* State Transition Graph (STG) – a graph model for modeling the finite automat, describing the general behavior of the node. The graph is represented like statechart described in [2].

## State Transition Graphs

Each node contains at least one STG defining its logical behavior (although there are some implementations containing only one state with an infinite loop to itself). The STG has one entry point (initial node) with no other function than pointing where exactly should start the execution of the STG when the system is started. For each state of the STG can be attached one or more SFG. For each state one or more transitions should be defined. A transition to the same state is acceptable. To make a decision which transition should be executed in each state should be built a Binary Decision Diagram (BDD) using values from predicates defined in SFGs of the node.

For each STG is defined an execution period defining how often the graph activates and executes its current state. The execution period can be modified if needed in runtime (e.g. when the system is in idle state scanning inputs and refreshing outputs at a high rate is not necessary).

STGs can also be viewed as separate tasks in an operating system. Each STG has a predefined priority and is executed accordingly.

The execution of each STG follows the next steps:

* Executing each of the current state SFGs in the preset order, one after another.
* Calculating the BDD to define which transitions will be activated.
* Activating transition to move the execution of the SFG to the next state.

There are two types of transitions defined: synchronous and asynchronous. When a synchronous transition is executed the graph execution stops and the task goes in sleep state, until the execution period expires. When an asynchronous transition is activated the task executes directly the next state pointed by the transition, without waiting the period to expire.

Infinite loops in STG are acceptable. The entry point of a STG can start with states needed for preparation of the system to operate normally. Once these preparations are ready, the system loops in the operating cycle for ever (as long as power is not interrupted or the task is killed by an external task manager).

## Signal Flow Graphs

SFG models the data flow of the system. Typically the SFG entry points are functional modules representing I/O drivers, communication drivers or in specific cases data calculated by other SFGs. Then the data is passed to other functional modules, which make transformations, check constraints and conditions and produce output for I/O drivers, communication, user visualization, database logging, etc. Each functional module in the SFG is executed only once per SFG execution. This is ensured by the connections between them. There are two types of connections: activating and nonactivating. The SFG execution starts with functional modules which don’t have activating inputs. After their execution remaining modules are checked for activation. The modules that have all their activating inputs set, start execution and so on until the last module of the SFG finishes its job.

Each Functional Module (FM) can have up to three types of inputs (unlimited count) and two types of outputs. The inputs of FM can be:

* Link inputs (activating or nonactivating). The activating inputs should be connected to an output of another FM in the same SFG. The nonactivating inputs can be linked to an output of any FM from any SFG in the system node. In some specific cases link inputs can remain unlinked (they can use a preset default value). In that case the program generator would produce a warning but will continue working. This behavior is made available to make possible for the system developer to use standardized FMs and to be able to connect only as much inputs as needed by current application.
* Parameter inputs. The parameter inputs are always nonactivating. They can also be linked to an output of another module (e.g. an adaptive controller, changing its parameter regarding the feedback) or they can be set by the developer to constant values during design.
* Internal state inputs. These inputs are also nonactivating. They are typically used by FMs to store internal data (mainly to instruct the system how much memory should be spared for the internal needs of each FM). In very specific cases they can also be modified like the parameter inputs (e.g. PID controller which stores its integral part in an internal state input. There are cases when another module wants to modify the internal state of the PID controller to achieve some specific performance to boost

Each input can be linked to only one source of data. An input which takes data from two different sources (e.g. from outputs of two separate preceding FMs) is not available.

Regarding the type of storage of data the inputs can be link inputs (data is stored outside the FM) or static inputs (data is stored inside the FM). Static inputs can also be linked to a special type of output – “point to point output”.

FMs can have two types of outputs:

* Static outputs – can be a data source of unlimited number of link inputs.
* Point to point outputs – cannot be a data source of inputs. Instead can be linked to a static input and change its value.

Outputs can remain unlinked. Each static output (linked or unlinked) with a Boolean type can be used as a predicate of the BDD of the STG. Currently the BDD receives only a Boolean predicates. This is a drawback because if a continuous value needs to be checked, additional comparator FM should be added to the SFG that will produce a Boolean output.

All inputs and outputs can hold matrix values, allowing complex system models to be made.

## PrGen Environment

The described system consists of two separate components:

* A program generator based on .Net, used for creating and analyzing the configurations. It produces binary files that are loaded in the distributed node. The generator is running on Microsoft windows only.
* A configuration interpreter – can be run on different hardware and software environments. Currently supported Microsoft Windows and Windows CE, MicroOS II.

The program generator has implemented validation algorithms for checking of the configuration graphs. It checks the BDDs of STGs and links and activation sequences in SFGs. The most common errors (like missing link to an activation input port, improperly connected output port, data type mismatch between input and output, improperly created BDD, etc.) are diagnosed during validation and are cleared in design time. As shown in figures 2 and 3 the generator has easy to use intuitive graphical representation of STGs and SFGs. Although, there are some specifics (in creating and configuring graphs, setting FM parameters, etc.), that require training for the control engineers, before they are able to produce configurations alone (typically 2 days). There is an open possibility for the developers to add their own FMs, creating definitions in the program generator and adding DLLs to the configuration interpreter. The FM structure has standardized template which allows end users to create compatible FMs corresponding to their needs.

The configuration interpreter reads the generated configuration and executes it. It has two execution phases: load phase and execution phase. During load phase the following actions take place:

* Memory allocation for FM I/O. Thus during execution phase no new memory allocations are needed. This ensures a predictable execution time for every SFG.
* Opening network connections. Typically TCP and UDP connections are opened during startup and are kept alive. In case of connection failure a restore procedures are implemented. In some cases for soft real time tasks opening and closing of connection during execution phase is acceptable.
* Initialization of the driver subsystems. Depending on the target device serial ports, ADC, DAC, I/O etc., are initialized and prepared for work.
* Creation of tasks (system threads) for each STG described in the configuration.

During execution phase all STGs are started ordered by their preset priority and executed accordingly.

Special considerations should be made when deploying the configuration on an operating system where no multitasking is available. All operations should be nonblocking in order to avoid deadlocks.

In order to make the system compatible with most of the currently used industrial networks OPC communication is embedded in the configuration interpreter, external tool for OPC configuration is created and

# PrGen Applications and result analysys

The described system effectiveness is proved by successful implementations in numerous projects.

So far the biggest application of the described system is the Building Management System (BMS) of Sofia Outlet Center (deployed 2010). The whole system was built by 3 PLCs with numerous remote I/Os, connected to the interpreter (running on PC with Windows Vista) by Modbus UDP connection. The logic for controlling and monitoring lighting, elevators, circuit breakers, ventilation fans, water and power meters of each shop and the lighting scenes was implemented by the program generator. The task was solved very rapidly, due to the possibility to make a logical template for each object (lighting circle, circuit breaker box, etc) and to multiply the objects just changing the communication addresses. The final configuration had over 3000 FMs in total of 7 SFGs, and 3 STGs (one for each controller). MySQL library of FMs was implemented to achieve full database log of all events, alarms and meter readings. Also an operator GUI was deployed making a full SCADA implementation.

Other significant application was the inventory system of Plama refinery (deployed 2011). To connect with all the various measurement equipment Modbus TCP, Modbus RTU and Melsec protocols were used. Implemented calculation of volume and mass stored in the tanks (calculations based on strap tables), monitoring of input and output fuel transactions, alarms for leak detection, web module for remote access by the customs authorities and operator module for local monitoring and document and report printing.

Compared to similar systems like the PLC languages by standard IEC61131 Functional Block and Sequential Function Chart [4] the described generator shows the following advantages:

* Ease of use – it is faster to learn how to create configurations and configuring is simpler and faster.
* Database integration - running the interpreter on a PC gives possibility to directly input all process data in the database server.
* Easier support – there is possibility to divide and group different logical tasks in separate graphs, also it is possible to rename FMs, thus producing clean and easily readable configuration.
* Easy integration with visualization software

Another interesting range of applications of the described program generator is given by the ability to model complex hybrid systems. It was used to make a simulator of a docking crane for operator training. The simulator received the operator commands by computer joystick and additional buttons, and various standard situations was emulated (e.g. loading and unloading of the crane, rotating while loaded, shaking of the crane actuator, overloading, etc.). To achieve this functionality a thorough identification of the crane has been made. Final experiments showed 98-99% accuracy of the simulator compared to the model built with Matlab.

Comparing the simulator to Mathworks Simulink Coder [5] (former Realtime Workshop), Embedded Coder [6] and Stateflow [7] the following points could be considered:

* Simulink Coder produces generic C code, which still has to be ported to a target system in order to control the simulation hardware. On the other side once the program interpreter is ported to work with the target platform, further changes in the configuration does not require any code changes.
* Each time a design change of the simulator is made the code should be ported again to the target platform.

# Conclusion

A good thing ☺

##### References

1. Matt Stephens “Automated Code Generation - The Fastest Way to Write Software?“ May 6, 2002,

<http://www.softwarereality.com/programming/code_generation.jsp>

1. David Harel, “Statescahrts: A visual formalism for complex systems” Science of Computer Programming 8 (1987) 231-274
2. Krishna M. Kavi, Bill P. Buckles, “A Formal Definition of Data Flow Graph Models” IEEE Transactions on computers. vol. C-35, No. 11, November 1986
3. GX IEC Developer Training Manual 2010
4. Mathworks Simulink Coder Documentations R2011a

<http://www.mathworks.com/products/simulink-coder/description1.html>

1. Mathworks Embedded Coder Documentation R2011a

http://www.mathworks.com/products/embedded-coder/description1.html

1. Mathworks Stateflow Documentation R2011a

http://www.mathworks.com/products/stateflow/description2.html