UNIVERSITY OF NICE SOPHIA ANTIPOLIS

Parallelism Project

International Master 1, Computer Science

Evgeny Morozov, Alexandros Panagiotidis, Rareş Damaschin,
Muhammed Raheel, Mourjo Sen

April-May 2014

Contents

Introduction 2

Parts of the project 2

Design choices 3

  Meta-model 3

  Generated code 4

Project management 5

  Division of tasks 5

  Priority and order of tasks 5

  Version control and issue tracking 5

Meta-model design 6

Model definition language 8

Example models 8

  Marked Graph 8

  SDFG 9

  KRG 10

Checking of properties of input models 11

  Cycle detection 11

  Safety and liveness checking 14

    Marked Graph 14

    Synchronous Data Flow Graph 14

Code generation 14

  Java implementation 14

  C implementation 16

  MPI4Lectures implementation 16

Additional notes 18

  Improvements that can be made on the project 18

  Bibliography 18

Introduction

The goal of the project is to implement a multi-language code generator which takes a theoretical model of computation and produces executable code in multiple implementations. The generator takes either of the following as an input: marked graph, synchronous dataflow graph (SDFG), k-periodically routing graph (KRG). The code generator provides options to generate executable code in any of the following implementations: Java or C (using POSIX threads). The code generator can generate a working implementation of any input model as per the user's choice.

The code generator was implemented as an Eclipse plug-in, which can be deployed in Eclipse. The Eclipse plug-in was developed in the Eclipse Modelling Framework (EMF) and can be easily imported as a plug-in in Eclipse. With the help of EMF, the input models can be specified by the user using the syntax tree. The input can also be provided using a grammar with the help of Xtext.

The project was completed in about three weeks by five students of the International Computer Science M1 program (2013-2014).

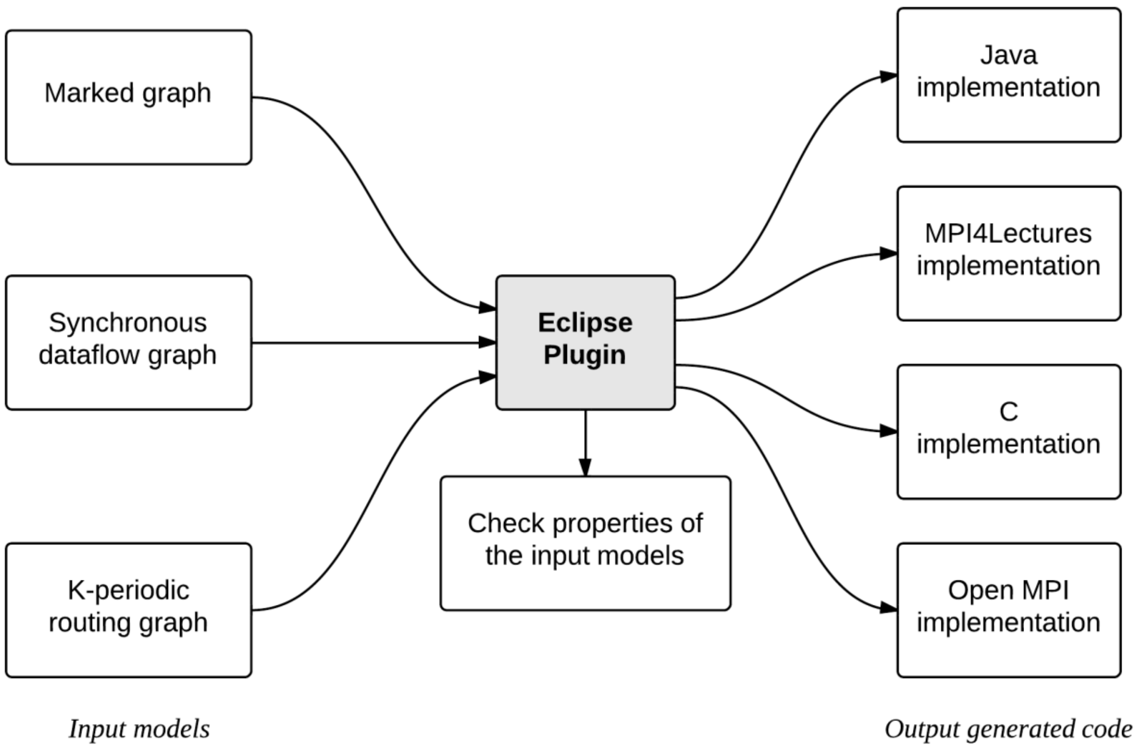


Figure 1: Graphical description of the project (Open MPI is not included in the final version)

Parts of the project

The project was divided into the following modules for independent development:

1. Theoretical description of the models (Marked Graph, SDFG and KRG) and correct definition of the theoretical models. The first stage of the project was to correctly define the theoretical models. This phase also included giving examples of models that the system should take.
2. Creation of the meta-models in EMF: After the initial definition of model was complete, the models were converted into meta-models in EMF, defining the structure of the models accepted as input from the user.

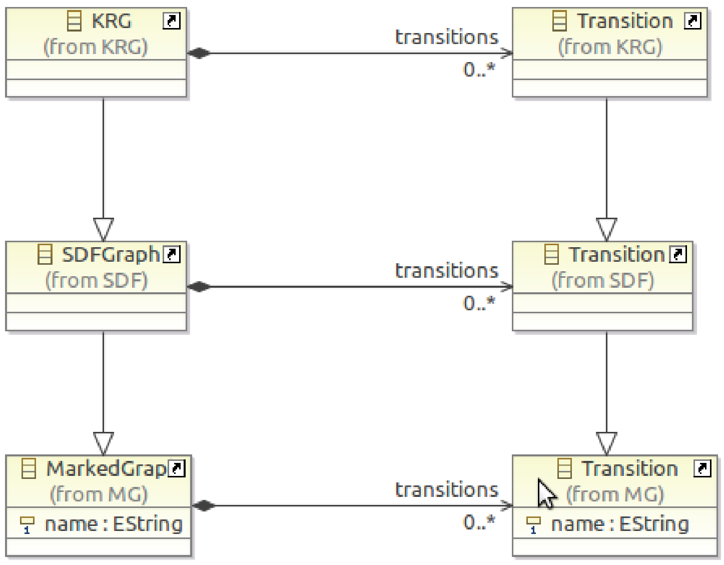
3. Code generation: The meta-models were used to generate executable code in the following languages, as closely related to the theoretical definitions as possible:
    a. Java: each Transitions is implemented as a class that implements the Runnable Interface and each Place is a class that contains a LinkedBlockingQueue which is a thread-safe FIFO queue. The transitions run until a minimum number of executions has been reached by each of transition(specified as a static field).
    b. C POSIX Thread: each Transition is implemented as a function that runs on a separate thread and each Place is implemented as a thread-safe FIFO queue. The transitions continuing firing (executing) until every transition has executed a fixed number of times (specified as a #define constant).
4. Verification of properties of the input models: Given a model as input, the plug-in checks its safety and liveness.

Design choices

During the development of the project we made a number of design choices. The most significant ones are listed below.

Meta-model

1. Meta-models for Marked Graph, SDFG and KRG are separate (though they are contained in one .ecore file). Conceptually, a Marked Graph can be considered a specific case of a SDF Graph, which itself can be considered a specific case of a KRG. Therefore, we ideally wanted to have SDFG meta-model entities inherit from MG entities and KRG entities – from SDFG entities, ie.



However, practical limitations of EMF did not permit such a model. The model can work only if a derived class can "override" a relationship from a base class. In this case, an SDFGraph must contain only SDF Transitions (overriding the base relationship to MG Transition), which is not supported by EMF. Without such overriding an SDFGraph could contain MG Transitions, which is incorrect. What's worse, each SDF Transition *must* belong to a MarkedGraph (since it *is* an MG Transition), which is not possible when it already belongs to an SDFGraph. It was still possible to inherit only attributes (not relationships) from common base entities, but we decided that the extra complexity would not be worth the reduced duplication and opted to keep the models separate.
2. Use of ports: Initially we faced problems with ordering of the arcs that are incoming to places and transitions. Ordering was important, because the generated executable code would have to pass

parameters to user-supplied functions in the correct order. The system would have to keep track of what to expect from which arc. Also, deletion of arcs would cause problems if we used the order of creation of arcs. We decided to use ports for the incoming and outgoing arcs in for transitions and places. A port specifies the details about the tokens that are involved with the associated place/transition and the arc. A port also enforces consistent data-typing: the type of a port is the type of its associated place, which makes it impossible to specify different data types for a transition to a given place and a transition from the same place.

3. The notion of "nodes": Places and transitions inherit a base abstract type called "Node". This was done to make the verification of the properties of the models easier. For example, cycle detection could be done in a relatively easily without the distinction between places and transitions, making it a directed graph having nodes and arcs.
4. Token count for Select and Merge transitions: Select and Merge transitions of a KRG consume and produce exactly one token at each invocation. However, they use same InputPort and OutputPort classes in the meta-model, which allows "tokensConsumed" and "tokensProduced" to be specified. It would be possible to prohibit this by using different InputPort and OutputPort classes with inheritance between the two, but this would significantly complicate the meta-model. We decided that this extra complexity was not worth it to prevent a relatively minor source of user confusion. Note that the plug-in and the generated code still work even if the user specifies a number of tokens for a Merge or Select input or output – it simply has no effect.

Generated code

5. Generic and graph-specific code: we aimed to separate, as much as possible, code that is common to all graphs (of a given type) from code specific to a given graph in order to minimise the amount of generated code and therefore simplify the code generator. In Java this was done using base classes (common to all graphs) for the graph and the transitions, which are simplify copied from the plug-in to the output directory. The generated graph and transitions classes inherit from these base classes. In C #include, #define and function pointers are used to achieve the same goal. The place implementations are common to all graphs (in both Java and C) – they are simply thread-safe FIFO queues.
6. When to terminate: Models of computation assume infinite execution, but in practice, the generated code must eventually terminate. We considered several solutions:
   a. *Stop after each transition has executed exactly N times* – this works only for simple graphs, because some transitions may execute more than others (eg. a Select transition in a KRG).
   b. *Stop after a fixed amount of time* – this is the simplest way to ensure the program terminates, but may leave the user code that implements the transitions in an inconsistent state (only some of a transition's code may execute before its thread is terminated).
   c. *Stop after each input transition has executed exactly N times* – this does not work for graphs without input transitions.
   d. *Stop after every transition has executed at least N times* – this is the solution we implemented in the end. More precisely, all transitions continue executing until every transition has executed *N* times (a constant defined in the generated code). Once all transitions have executed *N* times the queues for all places are marked "closed", which means that no more data (tokens) can be written to them. This signals all transition functions to exit, which terminates the process cleanly.
7. Thread safety: C code uses mutexes to ensure that the queue implementation is thread-safe and semaphores to signal when a queue can be read from or written to. Java code uses a standard library class that provides a thread-safe, blocking queue.

Project management

Division of tasks

The project was divided into tasks, which were divided among the group. Each person was responsible for a certain task or area of functionality, but may have also contributed to other areas.

- **Evgeny Morozov**: Evgeny was the project leader and was responsible for project management, version control system (Git), C POSIX thread code generator (including the C FIFO queue implementation), Xtext language development, deployment of the plug-in and overall correctness and consistency of the plug-in code. He also contributed to the meta-model definition, Eclipse plug-in user interface, Java code for Marked Graph, user guide and project documentation.
- **Mourjo Sen**: Mourjo was responsible for verifying the correctness of the definitions of the theoretical models (and providing examples of models for the system to function on), designing the EMF meta-models and the verification of the safety and liveness properties (jointly with Alexandros), project documentation and testing of the entire system after completion.
- **Alexandros Panagiotidis**: Alexandros was responsible creating the initial EMF meta-models, the verification of safety and liveness properties (jointly with Mourjo), initial version of the Java code generator for Marked Graph and user documentation.
- **Rareş Damaschin**: Rareş was responsible for manually writing Java code to implement a Marked Graph, an SDF Graph and a KRG (which was later used as a template for the generated code) and the Java code generator for SDF Graph and KRG.
- **Muhammed Raheel**: Raheel was responsible for the MPI4Lectures implementation of the marked graph, SDF and KRG models and MPI4Lectures code generation.

Priority and order of tasks

Even though the project consisted of many separate tasks it was not trivial to divide them between 5 people, especially at the beginning, because of the dependencies between the tasks.

Our first priority was to define the meta-model for one input language and generate code for one output language, that is, an "end to end" implementation of one of the 12 possible combinations of 3 input and 4 output languages. This would eliminate most of the unknowns and validate the meta-model. Marked Graph was the natural choice for the input language, since the other 2 are extensions of it. Java was chosen as the first output language, but others could work on C and MPI4Lectures at the same time. The initial tasks, therefore, were to create sample models, define the meta model, manually implement a Marked Graph in Java, in C and MPI4Lectures. These 5 tasks could all be done in parallel.

Once the meta-model definition of Marked Graph was stabilised we followed a similar process for SDF Graph and KRG, re-using the existing code as much as possible. Finally, safety and liveness checks were implemented in parallel with Xtext language syntax, documentation and bug fixes to code generation.

Version control and issue tracking

We used Git for version control and BitBucket to host the shared Git repository. We also used BitBucket's issue tracking system to keep track of which tasks remain outstanding and who is currently responsible for each.

Meta-model design

The meta-models designed for the three types of input models are given below:

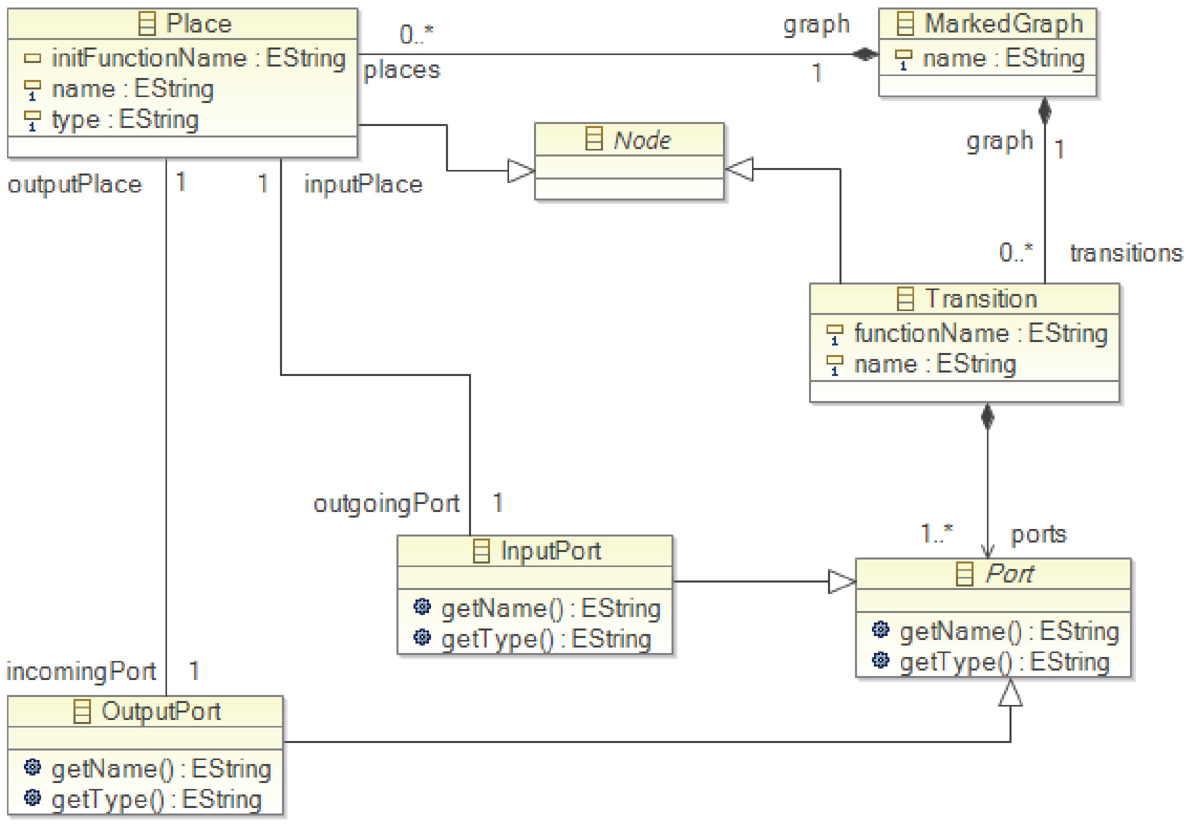


Figure 2: Marked graph meta-model

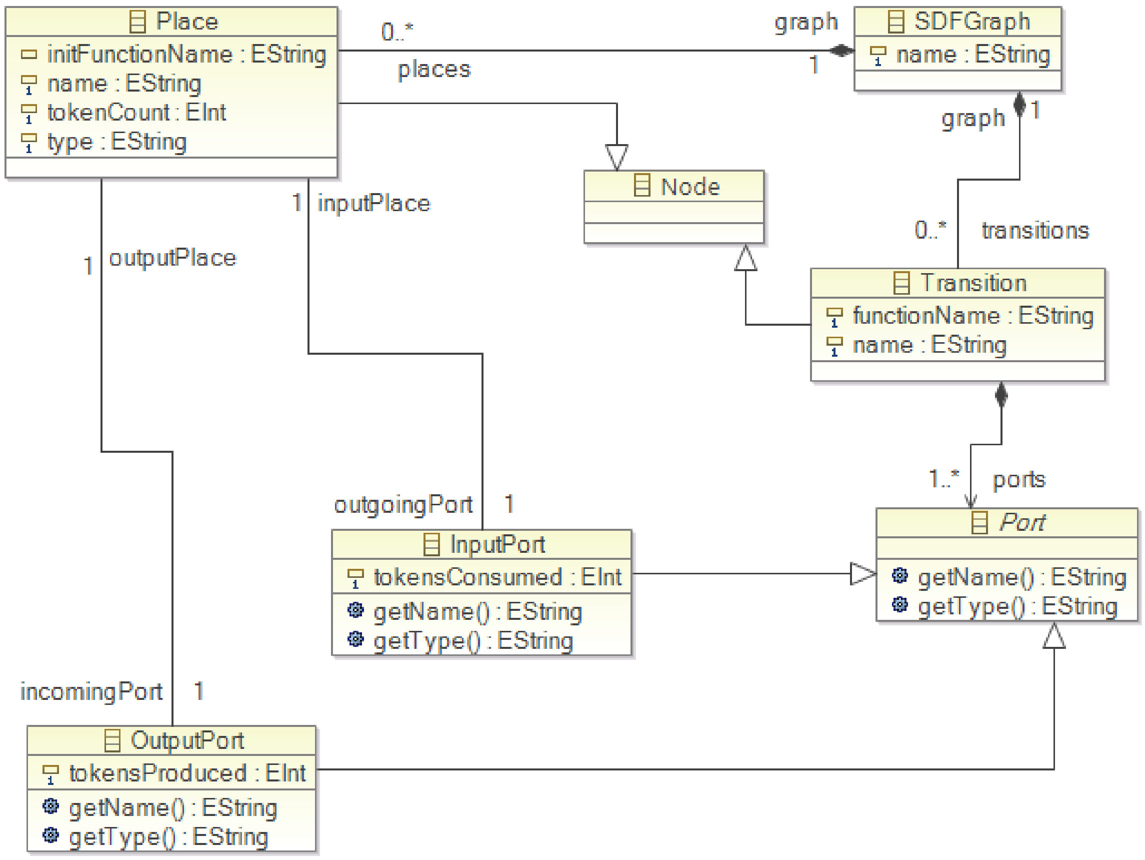


Figure 3: SDFG meta-model

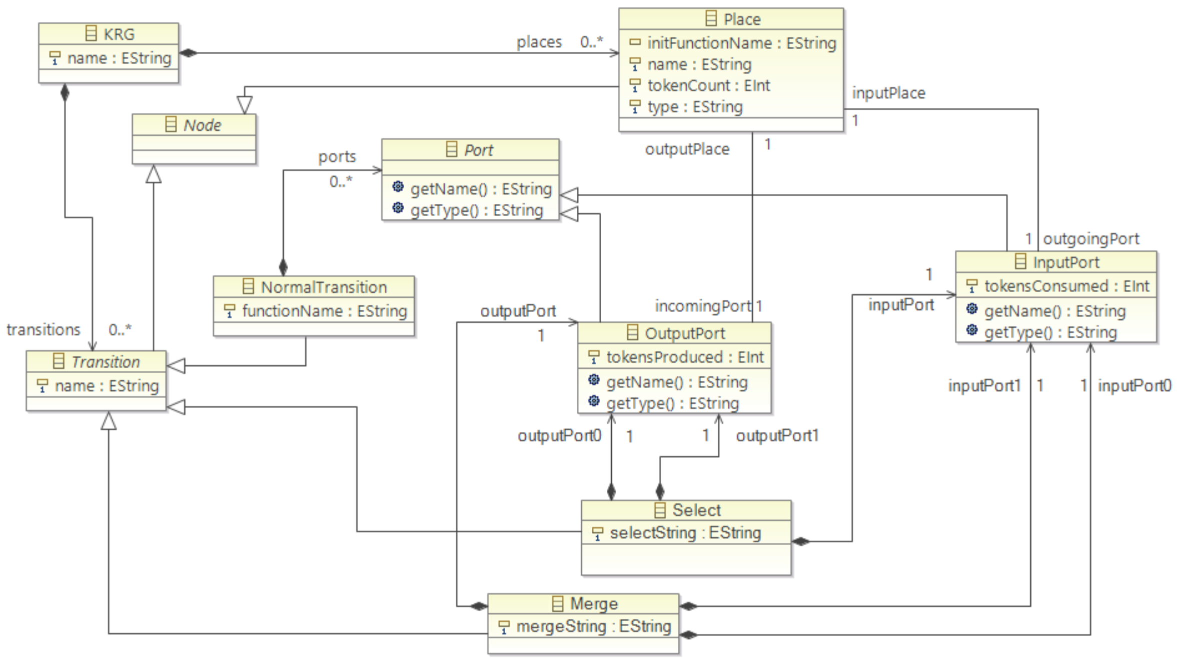


Figure 4: KRG meta-model

Model definition language

After defining the meta-models we created a domain-specific language (DSL) using the Xtext plug-in to enable the user to create models easily. It was already possible to create models using the default tree-view editor provided by Eclipse, but a DSL allows much quicker and easier editing and a more human-readable representation. With the plug-in installed it is enough to create a new file extension **.dataflow** to get syntax validation and auto-completion.

The language closely follows the meta-model, however it aims to be compact, while also being easy to read and understand. For example "from <Place>" was used as the syntax to indicate an incoming transition arc, because it's short and intuitive (compared to something like "InputPort" or "IncomingArc"). Transition arcs are defined as part of a transition, rather than separately. This way the user does not have to mentally jump between multiple lines of code to understand or edit a transition. The language also aims to re-use syntax elements that would be familiar to a Java or C programmer, such as array size notation for the token count and semicolons to end element definitions.

The **User Guide** includes full details of the language syntax. Note that it's possible to switch between the tree-view editor and the Xtext text editor for a given model. Our plug-in also includes the ability to convert a model initially defined in the tree-view editor (and stored as XMI) to the language.

Example models

Marked Graph

1. Division of two numbers: This simple model takes two numbers A and B and computes the division A/B.

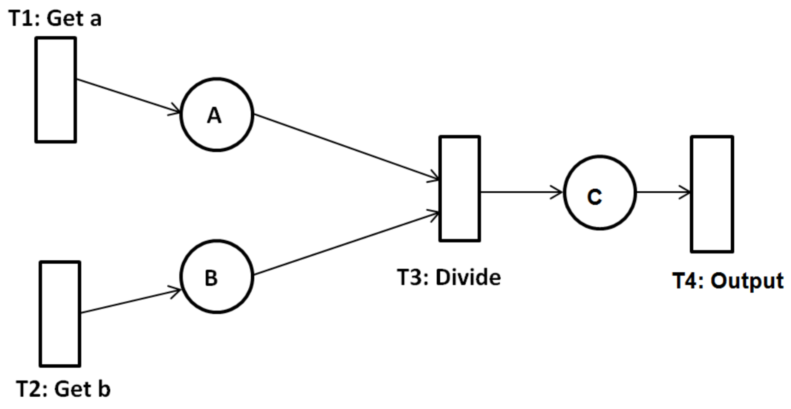


Figure 5: Marked graph for division of two numbers

2. Example from Parallelism Lab Session 4: The transitions in this example basically just copy the inputs to the outputs except transition T1 which adds a random number to the value in P2 and outputs it to P1. T6 computes the minimum of the two of its inputs. This example demonstrates nested cycles.

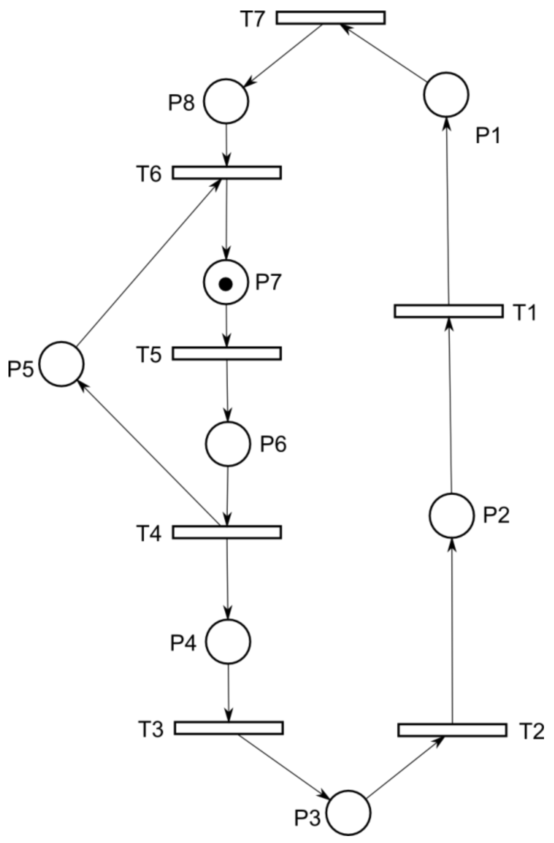


Figure 6: Marked graph from lab 4 of parallelism

SDFG

1. Maximum computation of two numbers: The transition “getA” takes input and max computes the maximum of every two consecutive inputs.

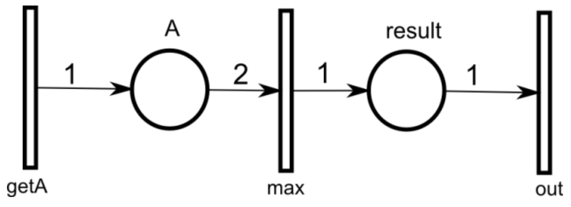


Figure 7: Simple example of an SDFG that calculates the maximum of every two inputs

2. Second example: This example demonstrates an unsafe SDF.

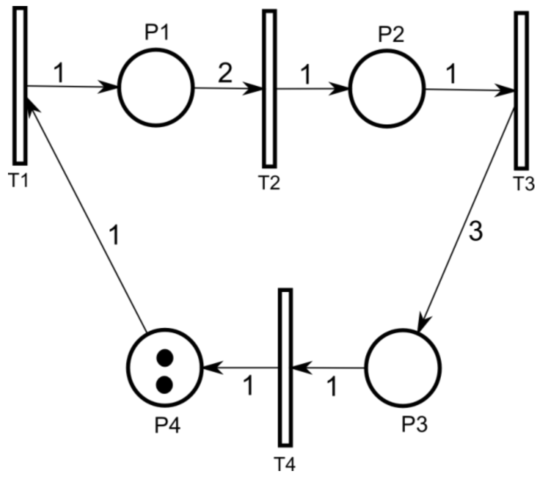


Figure 8: Another example of SDFG

KRG

1. A simple example: This is a simple example to demonstrate the "select" transition in KRG. The inputs are alternately sent to P2 and P3.

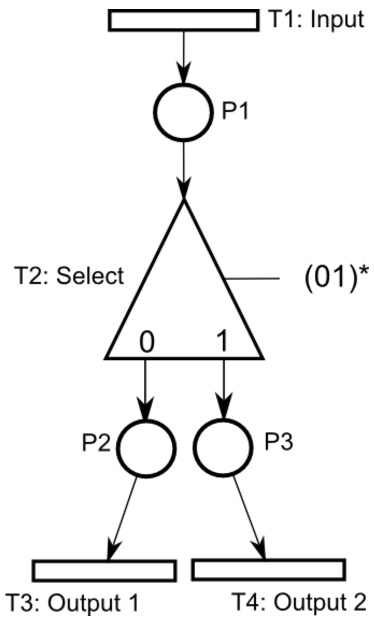


Figure 9: A simple example of a KRG

2. A more complex example[1]

[1] The example is inspired from: http://www-sop.inria.fr/aoste/personnel/Julien.Boucaron/slides/synchron_v3.pdf

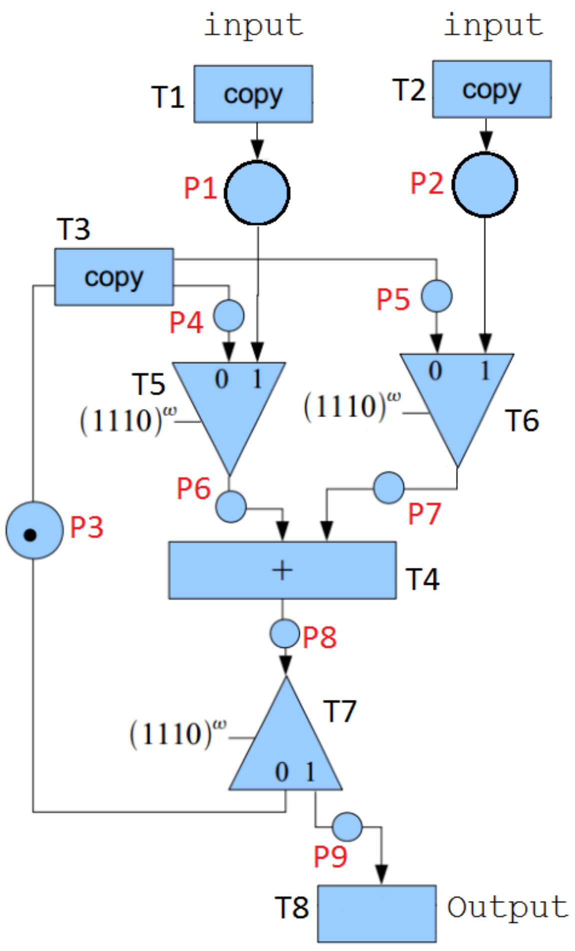


Figure 10: A more complex example of a KRG

Checking of properties of input models

The project can check an input model for safety and liveness properties. The first step was to find cycles in the input model. We developed an algorithm for finding cycles in an unweighted, directed graph. After the cycles are found, we then check for liveness and safety.

Cycle detection

The algorithm abstracts the input model (marked graph, for example) into a graph, using the common base class of transitions and places. The algorithm basically takes every arc A⇨B and finds all paths from B to A.

The algorithm is described below:

1. Initialise set of cycles to empty
2. For all arcs, do
    a. Let n1 be the source node of the arc
    b. Let n2 be the destination node of the arc
    c. Do a DFS of the graph from n2. The DFS returns all nodes **that can be reached from n2**. Call this the source tree. ("source" because we want to find all paths from n2 to n1 for a cycle)

d. Do a "reverse DFS" of the graph from n1. By "reverse DFS" we mean taking the opposite direction of the edges. Thus, the "reverse DFS" finds all the nodes **that can reach n1**. Call this the destination tree. ("destination" because we want to find all paths from n2 to n1 for a cycle)

e. For all common nodes in the source tree and the destination tree, we have a cycle (with respect to the considered edge).

   i. Find the path from the root of the source tree to the common node.

   ii. Find the path from the node in the destination tree to the root.

   iii. Concatenate the two paths, and this forms a cycle.

   iv. Check if a node appears more than once (if yes, discard the cycle, since we are interested in minimal cycles and not maximal cycles)

f. Add cycles obtained for this arc to cycle set only if it has not been encountered before

3. Return the set of cycles

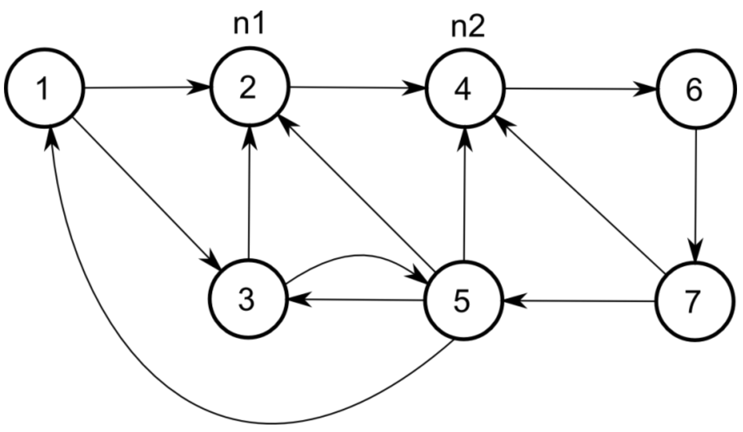


Figure 11: An example graph to demonstrate the cycle detection algorithm

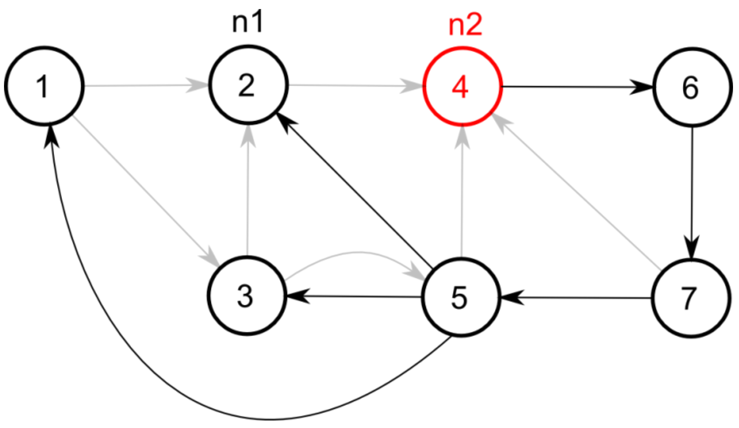


Figure 12: The "source" tree (in black) obtained by a DFS from n2 (in red)

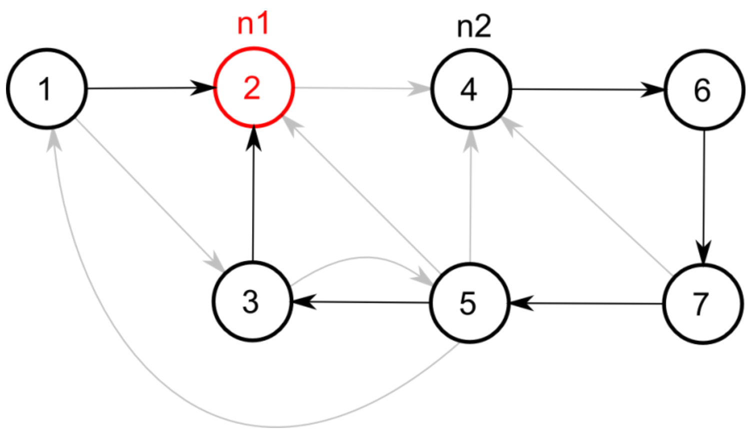


Figure 13: The "destination" tree (in black) obtained by a "reverse DFS" from n1 (in red)

Let us take an example (see figure 11). Let us demonstrate one iteration of the algorithm, i.e. to find cycles between two nodes. Let the arc under consideration be the arc from the node n1 to n2. We do a DFS from n2 (figure 12) to obtain the "source" tree. Then we do a reverse DFS from n1 (figure 13) to obtain the "destination" tree. The paths obtained by taking each common node in the source and destination trees and concatenating the paths are given in the following table:

Path to common node from the root (n2) of the source tree root	Path from common node to the root (n1) of the destination tree	Cycle detected
4 ⇨ 6	6 ⇨ 7 ⇨ 5 ⇨ 3 ⇨ 2	4 ⇨ 6 ⇨ 7 ⇨ 5 ⇨ 3 ⇨ 2
4 ⇨ 6 ⇨ 7	7 ⇨ 5 ⇨ 3 ⇨ 2	4 ⇨ 6 ⇨ 7 ⇨ 5 ⇨ 3 ⇨ 2
4 ⇨ 6 ⇨ 7 ⇨ 5	5 ⇨ 3 ⇨ 2	4 ⇨ 6 ⇨ 7 ⇨ 5 ⇨ 3 ⇨ 2
4 ⇨ 6 ⇨ 7 ⇨ 5 ⇨ 2	2	4 ⇨ 6 ⇨ 7 ⇨ 5 ⇨ 2
4 ⇨ 6 ⇨ 7 ⇨ 5 ⇨ 3	3 ⇨ 2	4 ⇨ 6 ⇨ 7 ⇨ 5 ⇨ 3 ⇨ 2
4 ⇨ 6 ⇨ 7 ⇨ 5 ⇨ 1	1 ⇨ 2	4 ⇨ 6 ⇨ 7 ⇨ 5 ⇨ 1 ⇨ 2

Taking the set of cycles, we obtain the following cycles:

- 4 ⇨ 6 ⇨ 7 ⇨ 5 ⇨ 3 ⇨ 2
- 4 ⇨ 6 ⇨ 7 ⇨ 5 ⇨ 1 ⇨ 2
- 4 ⇨ 6 ⇨ 7 ⇨ 5 ⇨ 2

These three cycles are obtained only for one edge. When we run the algorithm on all edges, the set of cycles obtained are:

- 2 ⇨ 4 ⇨ 6 ⇨ 7 ⇨ 5 ⇨ 1
- 3 ⇨ 5 ⇨ 1
- 3 ⇨ 2 ⇨ 4 ⇨ 6 ⇨ 7 ⇨ 5 ⇨ 1
- 4 ⇨ 6 ⇨ 7 ⇨ 5 ⇨ 3 ⇨ 2
- 4 ⇨ 6 ⇨ 7 ⇨ 5 ⇨ 2
- 5 ⇨ 3
- 6 ⇨ 7 ⇨ 4
- 6 ⇨ 7 ⇨ 5 ⇨ 4

To implement the cycle detection algorithm, we used a tree data structure which was originally written by Yohann Coppel (ycoppel@google.com). Of course we had to modify it to suit our needs.

Safety and liveness checking

Marked Graph

After all cycles have been found in the input model, the safety of a marked graph was verified by checking if all places of the marked graph belonged to at least one cycle or not. If there is even one place that does not satisfy this condition, the marked graph is not safe.

The liveness of a marked graph was verified by checking that all cycles has at least one token in the cycle. If there is a missing token in any cycle, the marked graph is not live.

Synchronous Data Flow Graph

The algorithm for checking liveness and safety in a SDF graph is based on [1], in which they constructed an algorithm for a self-timed SDF graph to detect liveness and safety. First, we need to check if the SDF graph is consistent, which means that there exist a solution to the homogenous linear equation system, corresponding to the topology matrix of the SDF graph. The existence is checked by the rank of the matrix, computed by the Gauss Elimination technique. This is a sufficient condition for the graph to be live, but we also need to check for every strongly connected component that does not have any deadlock. To achieve this, the strongly connected components must be detected, using the cycle detection algorithm used in a MG graph. Moreover, the exact solution of the homogenous linear system must be computed in order to check if there are enough initial tokens, but it has not been implemented due to lack of time. To implement the algorithm, material and code was used from [2], [3], and the Wikipedia of Gauss Elimination.

Code generation

This section describes the implementation details of the code generation part of the project. Refer to the **User Guide** for a description of code generation from the user's perspective. See also the **Design choices** section above.

Java implementation

The main design of the implementation was provided by the sample POSIX C implementation provided by Professor Millo. The principal difficulty in translating was the fact that Java does not have mutable variables so several alternatives were tried until using 1-element arrays was decided as the simplest method to emulate mutables in Java.

Code generation for java is handled by the plugin files:

MGGenerateJavaHandler.java, SDFGenerateJavaHandler.java and **KRGGenerateJavaHandler.java** generate the files for the project. The model independent files are placed inside a folder called TemplateJava for each of the three folders MG, SDF, KRG. The location of the output is chosen through a popup file explorer.

They call the following files, respectively, to generate the model dependent code.

MG/MGtoJava.java has two main parts:

getUserFunctionsSkeleton method returns a string with a skeleton of the Functions*.java file that must be edited by the user(described later in this section).

caseMarkedGraph is the main case function that is called. This function builds the contents of the MarkedGraph*.java file by calling caseTransition and casePlace to fill the code for transitions and places.

SDF/SDFtoJava.java is similar to MGtoJava.java, the difference being that it also uses the tokensConsumed field in the model to specify how many tokens a transition function reads/writes.

KRG/KRGtoJava.java is an extension of SDFtoJava.java which includes also generating Merge and Select functions for the merge and select transitions in the model. It calls caseMerge and caseSelect for this task.

The generated Java project contains 6 files:

The name of the files will have '*' replaced with the name of the graph that is specified in the model.

.project and .classpath for the Eclipse IDE.

src/Place.java is the implementation of a place component of the graphs. It uses a LinkedBlockingQueue as a FIFO for reading and writing tokens to a place. In the case of the Marked Graph it accepts elements of type Object, for SDF and KRG because of the weighted arcs it also accepts arrays Object[].

src/Functions*.java is the generated file that initially contains a skeleton structure with all the functions that need to be written by the user. This includes transition functions and place input functions(for initial marking). In order to have input functions generated, the user must specify a initFunctionName in the model.

src/MarkedGraphBase.java is a base abstract class for the Marked Graph java implementation that is used to separate most of the non model dependent code from the rest of the code. This file is the same for any Marked Graph model. Outline overview:

TransitionBase is the abstract class that each transition has to extend and implement in the MarkedGraph*.java file. It outputs to the primary console the inputs and outputs of every transition fired.

execute is the static method that is called in the main and handles starting and joining the threads of the program.

src/SDFGraphBase.java is similar to it's MarkedGraphBase.java predecessor with one change. It accomodates working with Object[] arrays and displays the inputs outputs of transitions accordingly.

src/KRGBase.java is again similar to it's KRGBase.java predecessor. The new features this time are the MergeBase and SelectBase classe:

MergeBase is the abstract class that each merge transition has to extend and implement. It outputs the contents of the token as well as what port was read from. It uses a String called bitString to hold the bit word given in the model.

SelectBase is the abstract class that each select transition has to extend and implement. It outputs the contents of the token as well as what port was written to. It uses a String called bitString to hold the bit word given in the model.

src/MarkedGraph*.java is the model dependent part of the generated code that does not require any user interaction afterwards.

It contains Transition* classes, where '*' is the name of the transitionFunctionName specified in the model for a specific transition. Transitions are connected to places in the implemented readInputs method(for input places) and executeTransition method(for output places)

Places are defined in the beginning of the file. They have input functions defined if the user has chosen an inputFunctionName for that place.

src/SDFGraph*.java is similar to MarkedGraph*.java with the distinction that it also has weights on inputs and outputs which depend on the tokensConsumed field specified in the model.

src/KRG*.java is an extension of SDFGraph*.java with the addition of Merge and Select type transitions. Both classes have their setup method implemented in order to specify a bitString and the connected input and output places.

C implementation

The C code uses POSIX thread (pthread) to implement transitions. It is based on the code provided by Professor Millo.

The FIFO queue implementation (**fifo.c**) provides basic read and write operations (**fifo_read** and **fifo_write** functions) as well as a "close" operation, which prevents further writes to the queue, but allows any existing data in it to be read. This is useful for allowing any transitions already executing to complete. Any further calls to **fifo_read** or **fifo_write** return 0, which indicates to the caller that the queue is closed and should no longer be used. Transition functions use this as a signal to exit. As described above, the program terminates when every transition has executed at least a certain number of times. This is implemented by closing all queues, which signals all transition functions to exit, which avoids any problems from forcefully terminating threads.

main.c contains the main() function and all other functions that are common to all graphs (of the type being generated). **graph.h** contains all code specific to the given graph.

The code can be compiled on Linux without any additional dependencies and on Windows using the third-party **pthread-win32** library (the relevant parts of which are included in the **win32** directory).

MPI4Lectures implementation

MPI4Lectures code generation part was not complete and thus not a part of the plugin.
Below are the details of the MPI4Lectures Implementation of the models of marked graphs, SDF and KRG.

MG.java this class implements the Marked graph model of calculating difference (subtraction).

Transitions can be considered as processors in MPI and processors are identified by their rank.
In this model we have 3 transitions/processors.
In MPI values sent and received among the processors are in the form of Bytes. So the values are converted to and from bytes.
IntToBytesConvertion () function is utility function that convert the integer value into bytes and return it.

ByteArrayToIntArray () function is another utility function that convert the byte array into integer array and return the integer array.

So according to the model: if it is **processor 1** it will sends the value of A to Processor 3 by converting it into bytes using the above mentioned utility functions.

If it is **processor 2** it will sends the value of B to Processor 3 by converting it into bytes using the above mentioned utility functions.

Finally if it is in **processor 3** then it will received the values sent by the processor 1 and processor 2 in bytes and after converting the values from Bytes to Integers it will then calculate the difference of the two numbers.
In a MG model, a transition (in this case: **Difference**) fires as soon as all input tokens are received (in this case: values A and B). Specifying the **no_of_iterations** variable value, which is currently set to 5, will iterate according to specified value.

SDF.java this class implements the simple SDF model of calculating maximum.

Again transitions can be considered as processors in MPI and processors are identified by their rank. But now in SDF model we have weights on the arcs and again we have 3 transitions/processors.

So according to the model: if it is **processor 1** it will fires/sends the value of A and the value of B simultaneously to Processor 2 (by converting it into bytes using the utility functions) as the weight on the arc is 2 for processor 2 so it can receive two tokens (values).

If it is **processor 2** , then it will received the values sent by the processor 1 in bytes and after converting the values from Bytes to Integers it will then calculate the maximum of the two numbers received and send the maximum value to the output processor 3.

Finally if it is in **Processor 3** then it will received the value sent by the processor 2 in bytes and after converting the values from Byte to Integer it will display the maximum number (Note that it will received only 1 token/value as the weight on processor 3 is 1).

In this SDF model, a transition (in this case: maximum) fires as soon as all input tokens are received (in this case: values A and B). Specifying the **no_of_iterations** variable value, which is currently set to 5, will iterate according to specified value.

KRG.java this class implements the simple KRG model.

Again transitions can be considered as processors in MPI and processors are identified by their rank.
In this model we have **4 transitions/processors**. Processor 1 sends the values to Processor 2 (which is the select transition node) and according to the select Bits, values are sent to either output transition/processor 3 or 4.

So according to KRG model: if it is **processor 1** it will fires/sends the random value to Processor 2 (by converting it into bytes using the utility functions).

If it is **processor 2** , then it will received the values sent by the processor 1 in bytes and after converting the values from Bytes to Integers it will then decide based on the **select bit pattern** stored in an array whether to send/forward the token/value to processor/transition 3 or 4. If the bit is 0 it will send it to processor 3 else if bit is 1 it will send the value to processor 4.

If it is in **processor 3** then it will received the value sent by the processor 2.

And if it is **processor 4** then it will received the value sent by the processor 2.

Additional notes

Some requirements in the project description were not implemented due to time constraints.

1. OpenMPI: We chose not to implement the OpenMPI output language. Since none of us had any experience with OpenMPI (unlike the rest of the output languages) and it appeared to be a reasonably complex library we decided from the beginning to implement it last, if we had time available. We decided that the limited time available near the end of the project was better spent on improving existing functionality than attempting an OpenMPI implementation.
2. Property checking: we did not have time to implement flow control for SDFG or liveness or safety checks for KRG.

Improvements that can be made on the project

The cycle detection algorithm could be made more efficient by using Tarjan's strongly connected components algorithm[2]. By running our cycle detection algorithm on the strongly connected components returned by Tarjan’s algorithm, we can obtain a much more efficient cycle detection algorithm since every cycle is a strongly connected component and Tarjan’s algorithm is more efficient than the algorithm used in the cycle detection here (but Tarjan’s algorithm does not return cycles).

Bibliography

[1] Liveness and Boundedness of Synchronous Data Flow Graphs, Eindhoven University of Tenchology, Electronic Systems Group, A.H. Ghamarian et al, 2006.

[2] Synthesis of embedded software from synchronous dataflow specifications, S.Bhattacharyya et al, Journal on VLSI Signal Process. Syst, 1999.

[3] A coupled hardware and software architecture for programmable digital signal processors, Chapter 2, University of California, PhD thesis, E. Lee, 1986.

[2] http://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm