J. Yaghob (Ed.): ITAT 2015 pp. 9–16
Charles University in Prague, Prague, 2015

Hybrid Flow Graphs: Towards the Transformation of Sequential Code into Parallel Pipeline Networks

Michal Brabec, David Bednárek

Department of Software Engineering
Faculty of Mathematics and Physics, Charles University Prague
{brabec,bednarek}@ksi.mff.cuni.cz

Abstract: Transforming procedural code for execution by specialized parallel platforms requires a model of computation sufficiently close to both the sequential programming languages and the target parallel environment. In this paper, we present Hybrid Flow Graphs, encompassing both control flow and data flow in a unified, pipeline based model of computation. Besides the definition of the Hybrid Flow Graph, we introduce a formal framework based on graph rewriting, used for the specification of Hybrid Flow Graph semantics as well as for the proofs of correctness of the associated code transformations. As a formalism particularly close to pipeline-based runtime environments which include many modern database engines, the Hybrid Flow Graphs may become a powerful means for the automatic parallelization of sequential code under these environments.

Keywords: compiler, graph, optimization, parallelism

1 Introduction

Parallelization of procedural code has been studied for several decades, resulting in a number of mature implementations, usually tightly coupled with compilers of FORTRAN or C. In most cases, the parallelization techniques are targeted at multi-threaded environment, sometimes augmented with task-based parallelism. This shared-memory approach is perfectly suited for numeric applications; however, there are niches of computing which may benefit from a different run-time paradigm. In particular, database systems make use of execution plans which explicitly denote the data flows between operators; similar graph-based representation is often used in streaming systems. Besides other advantages, the explicitly denoted data flow allows easier distribution of the computation across different nodes, compared to the complexity of potentially random access in the shared-memory model of computation.

Nevertheless, database and streaming systems lack the generality of procedural programming languages. To extend the applicability of such runtime systems towards general programming, a system capable of compiling (a subset of) a well-known procedural language is required. This paper demonstrates an important step towards such a system – an intermediate representation capable of describing procedural code using the language of data-flow oriented graphs.

Our approach is focused at target environments in which an application is presented in two layers – the declarative upper layer represented as a graph of operators, the lower layer consisting of a procedural implementation of individual operators. In such a setting, a language front-end transforms the application source code into an intermediate representation; then, a strategic phase decides where to put the boundary between the declarative upper layer and the procedural lower layer. Finally, the upper layer is converted into the final graph while procedural code (in source, bytecode, or binary form) is generated from the lower layer.

Realization of this idea involves many particular problems; in this paper, we deal with the intermediate representation which serves as the backbone connecting the front-end, strategy, and generator phases. Since the upper layer will finally be represented by a graph, it seems reasonable that also the intermediate representation be graph based. However, such a graph shall also be able to completely represent the lower, procedural part of the application. Although the theory of compiling uses graphs in many applications including the representation of the code, the control-flow and data-flow aspects of the code are usually approached differently. In our approach, the control flow and the data flow is represented by the same mechanism because the strategic phase must be able to make the cuts inside both the control flow and the data flow.

In this paper, we present *Hybrid Flow Graphs* (HFG) as an intermediate representation of procedural code, capable of representing control-flow and data-flow in the same layer. We will demonstrate two forms of the HFG, the *Sequential HFG* and the *General HFG*. The former form has its execution constrained to sequential, equivalent to the execution of the procedural code which it was generated from. The latter form exhibits independent execution of individual operations, showing available parallelism. Of course, making operations independent requires careful dependency analysis – we assume that the required points-to/alias/dependency analyses were made before the conversion to HFG, using some of the algorithms known from the theory of compiler construction.

We will describe the operational semantics of both the sequential and the independent forms of HFG and we will also demonstrate the conversion between them. We will use graph rewriting as the vehicle for the description of both the semantics and conversions. The use of graph

rewriting systems allows for some generality of the approach, both in the description and in the implementation. In particular, the set of operations provided by the intermediate code may be defined almost arbitrarily as long as their semantics is depicted by graph rewriting rules. On the other hand, the set of control-flow operations is fixed since it forms the non-trivial part of the transformation from sequential code.

In this paper, we will use the set of types and operators borrowed from the C# language, more exactly, from its standard bytecode representation called CIL [1]. This includes some artifacts specific to the CIL, namely the stack and the associated operations. Since the Sequential HFG mimics the behavior of the CIL abstract machine, the stack operations are preserved in the HFG. However, they must disappear in the transformation to the General HFG; otherwise, the presence of stack operations would prohibit almost any parallelism. Since the removal of stack operations is a non-trivial sub-problem, we include the corresponding (CIL-specific) algorithm in this paper, as well as the description of the preceding conversion of CIL into the sequential HFG.

The rest of the paper is organized as follows: We review the related work in Section 2. The hybrid flow graph and its semantics is defined in Section 3. We define the sequential hybrid flow graph in Section 4. In Section 5, we present an algorithm producing a sequential HFG from a source code. Section 6 presents the algorithms that transform a general HFG to a sequential HFG.

2 Related Work

The flow graph described in this paper is similar but not identical to other modeling languages, like Petri nets [2] or Kahn process networks [3]. The main difference is that the flow graph was designed for automatic generation from the source code, where the other languages are generally used to model the application prior to implementation [4] or to verify a finished system [5]. The flow graph is similar to the graph rewriting system [6], which can be used to design and analyze applications, but it is not convenient for execution. There are frameworks that generate GRS from procedural code like Java [7], though the produced graphs are difficult to optimize. The flow graph has similar traits to frameworks that allow applications to be generated from graphs, like UML diagrams [8] [9], but we concentrate both on graph extraction and execution.

The flow graph is closely related to graphs used in compilers, mainly the dependence and control flow graphs [10], where the flow graph merges the information from both. The construction of the flow graph and its subsequent optimization relies on compiler techniques, mainly *points-to* analysis [11], *dependence* testing [12] and *control-flow* analysis [13]. In compilers, graphs resulting from these techniques are typically used as additional annotation over intermediate code.

The flow graph is not only a compiler data representation, it is a processing model as well, similar to KPN graphs [14]. It can be used as a source code for specialized processing environments, where frameworks for pipeline parallelism are the best target, since these frameworks use similar models for applications [15]. One such a system is the Bobox framework [16], where the flow graph can be used to generate the execution plan similarly to the way Bobox is used to execute SPARQL queries [17].

2.1 Graph Rewriting

A *graph rewriting system* (GRS) is a set of rules R transforming one graph to another. The rewriting systems are very similar to grammars where each rule has a left side that has to be matched to the graph and a right side that replaces it.

The GRS are very simple and they are best explained by an example. Figure 2 shows the CIL based rewriting system, where each rule emulates a single CIL instruction.

We use extended rewriting rules where $*$ represents any node (wildcard) and we add Greek letters to identify unique wildcard nodes where necessary. Variable names (x or y) represent only nodes with data. Expressions are used to simplify data manipulation. This is necessary so we do not have to create a rule for each combination of instruction or inputs.

3 Hybrid Flow Graph

A *hybrid flow graph* (HFG) is an annotated directed graph, where nodes together with their labels represent operations and edges represent queues for transferring data. The direction of the edge indicates the direction of data flow. Figure 1 shows a general hybrid flow graph for a simple function (see Listing 1 for source code).

```
void SimpleLoop(int a) {
for(int i=a; i<5; )
{
i = call(i);
}
}
```

Listing 1 Simple function containing a single loop

A hybrid flow graph is a general concept that can be used regardless of operations O or data types T. A *platform specific flow graph model* is defined by specifying O and T based on the target platform. The platform specific definition is $HFG\_MODEL_{platform} = (O \cup \{special\ operations\}, T)$, where O and T are platform specific and *special operations* are platform independent (see Section 3.1). The hybrid flow graph semantics (explained in Section 3.2) are also platform specific.

As our research is focused on C#, we use CIL [1] instructions to specify node operations. The most common

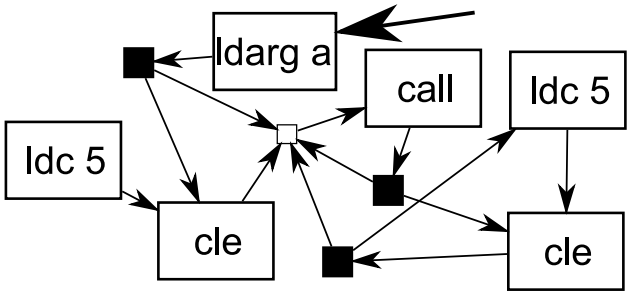


Figure 1: The HFG representation of the C# code from Listing 1

instruction in our examples are: *ldl x* (load variable or constant), *stl x* (write to a variable), *cle* (compare less or equal), *add* etc. (mathematical operations) [1]. We omit data types for the edges, because they are not important for the graph construction.

3.1 Representation of Control Flow

Control flow operations are transformed into platform independent *special operations* which interact with the data-flow carried through the platform specific *basic operations*. Graphical notation of special operations is in Figure 3, where a broadcast is a black square and a merge is a white square.

A *broadcast node* has a single input and a variable number of outputs. It represents an operation that creates a copy of the input for each output.

A *merge node* has a single output and three inputs. It represents an operation that accepts data from two sources and passes them to a single operation based on a condition, it is used to merge data flow after a conditional branch.

A *loop merge node* is a special version of the merge node with two inputs for conditional branches, it is used to merge data in loops. The input is split into two pairs, where each contains one data input and one condition input. The node first reads data from the first pair and then it loops, while the second pair has a positive value in the condition input.

3.2 Semantics of Hybrid Flow Graphs

The *hybrid flow graph semantics* define the way nodes process data and communicate. Basically, nodes represent operations or data and edges represent unbounded queues (FIFO) for data transfer.

Semantics are platform specific, similar to the hybrid flow graph model. We define the *hybrid flow graph semantics* using the *graph rewriting systems* according to the platform specific operations logic.

We define the HFG semantics for CIL using graph rewriting. Figure 2 shows the example of definition for the basic instructions. Instruction load constant (*ldc*) has a special behavior, it produces a single constant and then

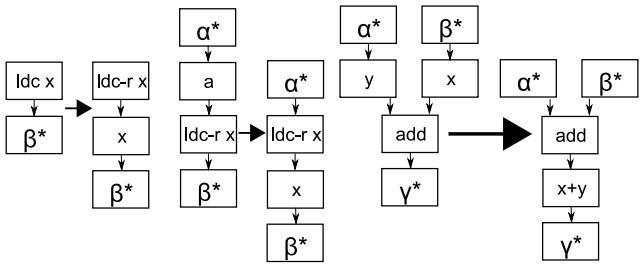


Figure 2: Semantics of selected basic operations

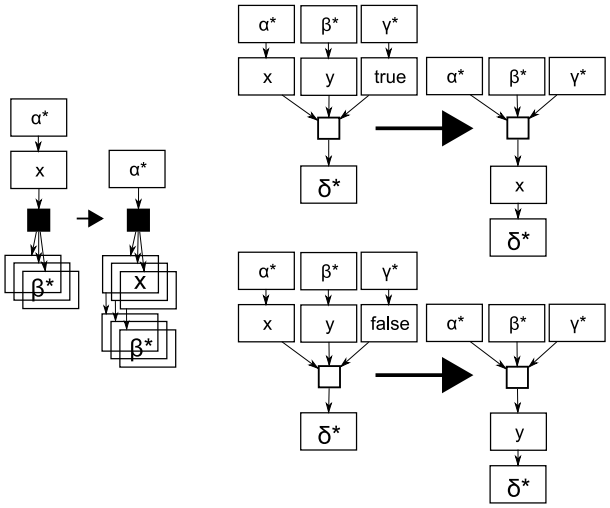


Figure 3: Semantics of selected special operations

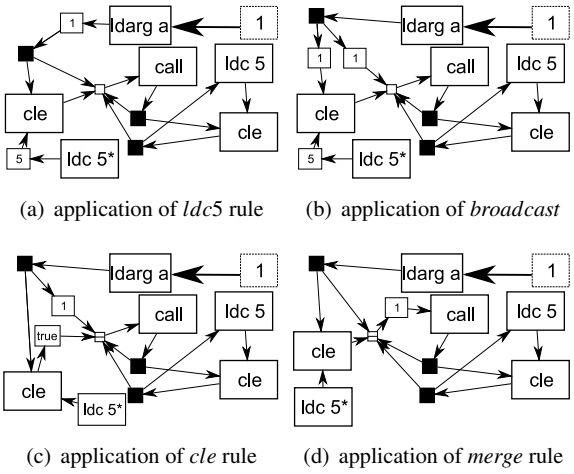
it repeats while it has an input, we specify this behavior by changing its label to $ldc - r$. The definition of special operations is illustrated in Figure 3, the figure contains the definition of broadcast and merge. We use special graphical notation for special operations, since they are platform independent (broadcast is black square, merge is a white square).

We can use the graph rewriting rules defined in Figure 2 and 3 to perform the computation of a hybrid flow graph. The computation of a flow graph with a for-loop is shown in Figure 4. Each part of Figure 4 contains the HFG state after application of a single rule and the computation starts after application the *ldc*5 rule.

4 Sequential HFG

In this section, we define the semantics of a sequential hybrid flow graph using the graph transformation systems. A *sequential hybrid flow graph* is a special HFG that uses a program counter a memory nodes instead of special operations defined in Section 3.1. It has a strict control flow that closely follows that of the source program. It is useless for the parallel pipelines or other applications, but it allows us to create a general HFG from the source code.

We use special operations for variables and the stack (CIL virtual machine is stack based [1]), these operations



(a) application of *ldc*5 rule (b) application of *broadcast*

(c) application of *cle* rule (d) application of *merge* rule

Figure 4: Computation of a hybrid flow graph

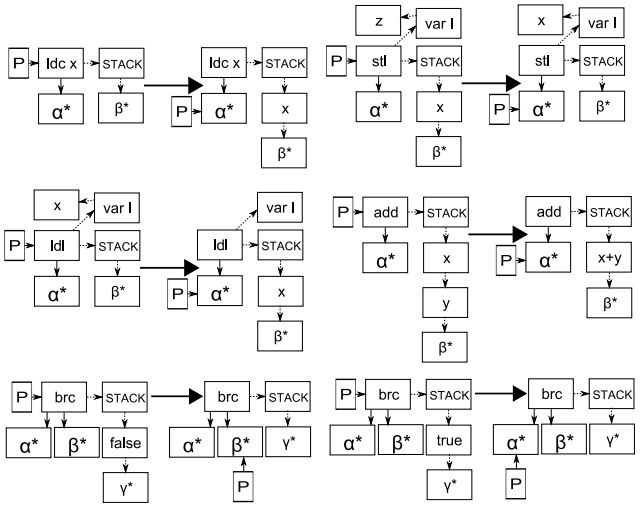


Figure 5: Semantics of selected CIL sequential HFG operations

are used to model the data flow. Plus we add a special node *P* (program counter) to model the control flow.

4.1 Semantics of a CIL Sequential HFG

The *sequential hybrid flow graph semantics* is defined using the graph rewriting, same as any other hybrid flow graph. The transformation rules for the CIL based sequential hybrid flow graph are defined in Figure 5, we present only the rules necessary for our examples.

The GRS rules defined in Figure 5 are completely based on the CIL instruction definition [1]. The control flow is controlled by the node *P*, which follows the edges between instructions. The instruction edges are constructed according to the following rules: 1) The standard instructions follow one another 2) The branches have two outgoing edges, one following the jump target and the other leading to the next instruction.

The data management is handled by the stack and variable operations, where each instruction modifies the

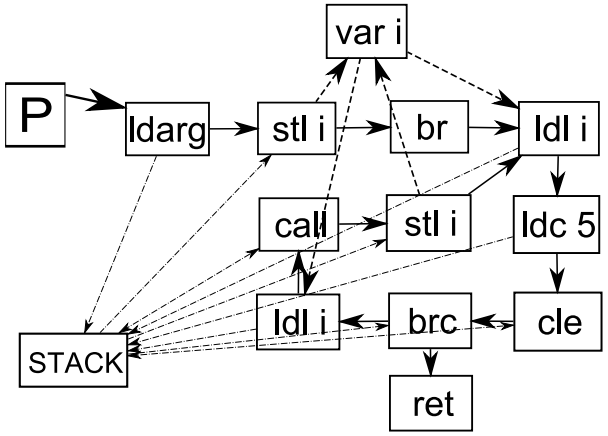


Figure 6: Sequential HFG created from code in Listing 1

proper number of stack levels and variables, based on the instruction definition.

The actual computation is the same as in the case of the .NET virtual machine, the control flow follows the same path, variables and stack contain the same value at after every evaluated instruction. Graphically, the computation is similar to Figure 4.

5 Transformation of CIL to Sequential HFG

In this section, we explain how to create a sequential hybrid flow graph from a CIL code. We present an algorithm that produces a sequential HFG that follows the CIL computation step by step, including control and data flow. It allows us to define the advanced transformation algorithm in Section 6.

The basic transformation of a CIL code to a sequential HFG requires two steps: 1) create nodes based on the instructions 2) create edges based on the control and data flow.

The entire transformation is in Algorithm 1, function v maps nodes to operations. The node P is basically a program counter, it points to the actual instruction and it is used to handle the control flow. The input sets I, M, P_i are obtained from the source code. Sr_i and Sw_i are based directly on CIL specification [1]. The transformation is best presented on an example, Listing 1 contains CIL source code and Figure 6 contains the resulting graph.

6 Transformation of Sequential HFG to General HFG

In this section, we present an algorithm that transforms a procedural code to a hybrid flow graph. We create the CIL sequential HFG (Section 5) and convert it to a general HFG that requires neither the program counter *P*, nor the

Algorithm 1 Transformation of CIL to Sequential HFG

Require: I – set of instruction addresses
M – set containing all variables
C_i, P_i – code and parameter of the instruction i
Sr_i – number of stack values read by instruction i
Sw_i – number of stack values written by instruction i
Ensure: V – nodes of the flow graph
$v(V_i)$ – label of the node V_i
E – edges of the flow graph

1: $V := \{V^{\mathsf{P}}, V^{\mathsf{STACK}}\}$
2: $V := V \cup \{V_i^{\mathsf{INSTR}} : i \in I\}$
3: $v(V_i^{\mathsf{INSTR}}) := \langle C_i\ P_i \rangle$ for each $i \in I$
4: $V := V \cup \{V_m^{\mathsf{VAR}} : m \in M\}$
5: $v(V_m^{\mathsf{VAR}}) := \langle \mathsf{var}\ m \rangle$ for each $m \in M$
6: $E := \emptyset$
7: **for all** $i \in I$ **do**
8: **if** $i = branch$ **then**
9: $E := E \cup \{(V_i^{\mathsf{INSTR}}, V_{P_i}^{\mathsf{INSTR}})\}$
10: **end if**
11: $E := E \cup \{(V_i^{\mathsf{INSTR}}, V_{i+1}^{\mathsf{INSTR}})\}$
12: **end for**
13: **for all** $i \in I$ **do**
14: **if** $i = stl$ **then**
15: $E := E \cup \{(V_i^{\mathsf{INSTR}}, V_{P_i}^{\mathsf{VAR}})\}$
16: **end if**
17: **if** $i = ldl$ **then**
18: $E := E \cup \{(V_{P_i}^{\mathsf{VAR}}, V_i^{\mathsf{INSTR}})\}$
19: **end if**
20: **if** $Sw_i > 0$ **then**
21: $E := E \cup \{(V_i^{\mathsf{INSTR}}, V^{\mathsf{STACK}})\}$
22: **end if**
23: **if** $Sr_i > 0$ **then**
24: $E := E \cup \{(V^{\mathsf{STACK}}, V_i^{\mathsf{INSTR}})\}$
25: **end if**
26: **end for**

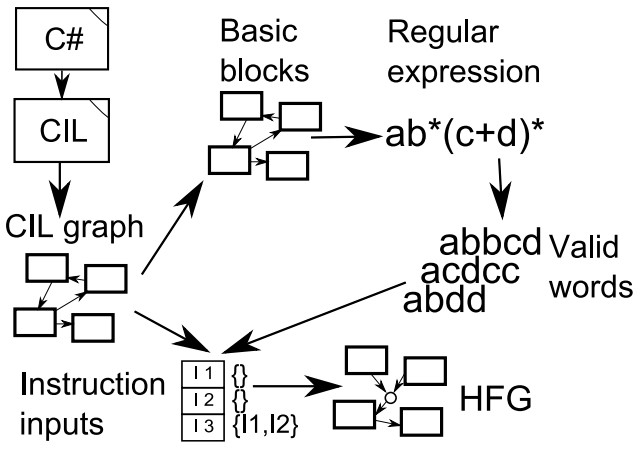


Figure 7: Transformation from CIL to a general HFG

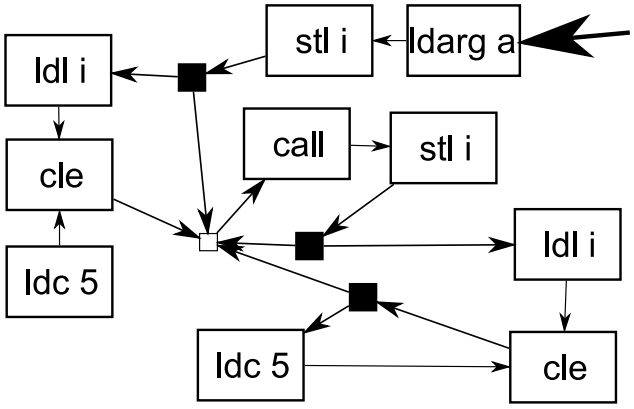


Figure 8: General HFG of the SimpleLoop function before empty operation elimination

memory nodes. The input for the algorithm is an intermediate code; in our case CIL.

The basic idea of the algorithm is that every CIL instruction is transformed to a node. The data and control flow is then modeled using the sequential HFG presented in Section 5. We emulate computation on the graph and note what data can reach which instruction, by what path. Finally, we create edges according to the collected information and we introduce special nodes where necessary.

6.1 Flow Graph Construction

To construct a complete flow graph, we create nodes based on instructions and use a modified sequential HFG to create edges and special nodes. We use basic blocks to model all valid execution paths which we use to create all the necessary edges. Algorithm 2 describes the entire process, the other algorithms are discussed in the following sections. The result of Algorithm 2, applied to the code in Listing 1, is the HFG in Figure 1 and Figure 8 shows the graph before NOP elimination.

The first step of the algorithm is to create basic nodes according to the instructions of the source code (lines 1 to 3 in Algorithm 2).

To crate the edges, we generate necessary execution paths using the *CIL path generator* (Section 6.4) and analyze all the possible inputs by the *CIL reachability* algorithm (Section 6.3). Basically, we emulate computations for every valid execution path recording the possible inputs, and the paths leading to them, for every instruction.

First, we convert the basic blocks to a regular expression (function *to_regular_expression*), then we generate all the valid paths using the *CIL path generator*. Next, we have to update the G_{CIL} so it contains the correct number of iterations (function *update*) for every loop (this due to the IDs assigned to branches as shown in Figure 9). Finally, we use the *CIL reachability* to create sets $S_{[0:N]}$ containing all the possible inputs for each instruction (lines 4 to 10). $S_{[0:N]}$ represents sets S_0 to S_N that contain inputs for instructions indexed 0 through N.

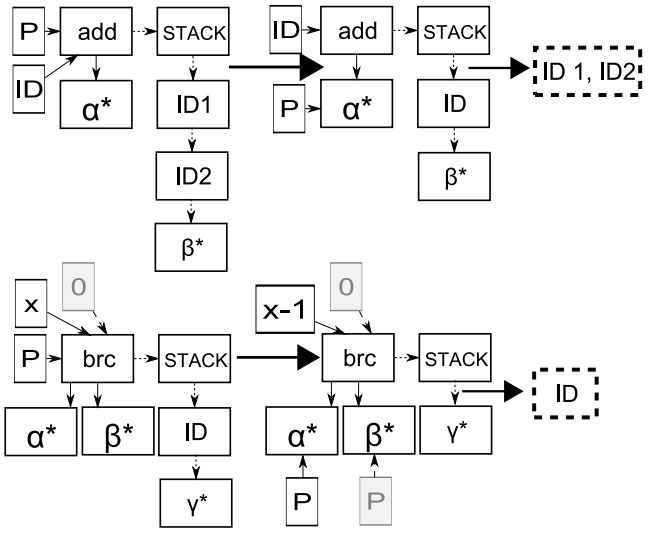


Figure 9: Symbolic semantics of selected Sequential HFG operations

We can create edges, once the input sets $S_{[0:N]}$ are ready. We start by discarding duplicate sources, because each instruction can produce only one value, which can reach the instruction i through multiple paths (different iterations inbetween), we can keep any instance as they are equivalent (lines 12 to 16). Then we check the number of inputs left, if there is the same number of sources as the instruction i has inputs then we simply create edges (lines 17 to 21).

If there is more sources than i has inputs then we create merge nodes to select the correct input value and we connect them to a condition that decides what input is used. We iterate over the sources and we compare the attached paths, the paths always start the same (i.e. the first instruction) and they continue until there is a conditional branch that changes the execution order for one of them. We create a *merge* node, connect both sources to it and we connect the branch condition to the condition input. This way, we reduce the number of incoming edges, until it matches the number of inputs.

The last step of Algorithm 2 is the elimination of empty operations that no longer have any purpose. This means mainly loading and storing local variables (*ldl* and *stl*). *Eliminate NOP* removes any nodes that do not change the data.

6.2 Symbolic Semantics of Sequential HFG

We use modified semantics for the CIL sequential HFG defined (Section 5). The modified rules work with instruction identifiers instead of actual values. This way, we can analyze which instructions consume the value produced other instructions. The rule modification is shown in Figure 9.

We assign each instruction an identifier equal to its index ($\forall i \in I : id(i) = i$). The identifiers of branches decide how many times the branch is $true$ before it is $false$, we use this to drive the control flow (explained in the next section). Last modification is that the rules output identifiers of consumed values, this is indicated by the bold dashed boxes on the right side in Figure 9.

Algorithm 2 Hybrid flow graph construction

Require: I – set of instructions
B – graph of basic blocks
G_{CIL} – modified CIL graph
R_{CIL} – modified rules for CIL graph
Ensure: N – nodes of the flow graph
E – edges of the flow graph

1: $N := \{N_i : i \in I \wedge v(N_i) = O_i\}$
2: $regex := to\_regular\_expression(B)$
3: $S_{[0:N]} := \emptyset$
4: **for all** $W \in CIL\_path\_generator(regex, G_{CIL})$ **do**
5: **for all** $s_i \in CIL\_reachability(I, update(G_{CIL}), R_{CIL})$ **do**
6: $S_i := S_i \cup s_i$
7: **end for**
8: **end for**
9: **for all** $s_i \in S_{[0:N]}$ **do**
10: **for all** $p \in s_i$ **do**
11: **while** $\exists q \in s_i : q.first = p.first \wedge q \neq p$ **do**
12: $s_i := s_i \setminus q$
13: **end while**
14: **end for**
15: **if** $|s_i| = input\_count(i)$ **then**
16: **for all** $p \in s_i$ **do**
17: $E := E \cup \{(p,i)\}$
18: **end for**
19: **else**
20: **while** $|s_i| > input\_count(i)$ **do**
21: $a := s_i[0]$
22: $b := s_i[1]$
23: $V := V \cup \{merge\}$
24: $E := E \cup \{a.first, merge\}$
25: $E := E \cup \{b.first, merge\}$
26: $E := E \cup \{merge, i\}$
27: **while** $a.second[pos] = b.second[pos]$ **do**
28: $pos := pos + 1$
29: **end while**
30: $E := E \cup \{pos, merge\}$
31: **end while**
32: **end if**
33: **end for**
34: $eiminate_N OP(V,E)$

6.3 CIL Reachability

We use the modified HFG semantics to locate all possible inputs for each instruction i, where an input is an instructions producing a value consumed by i. We do this by emulating CIL computation and storing the inputs for each instruction, implemented by Algorithm 3.

Algorithm 3 CIL reachability

Require: I – set of instructions
G_{CIL} – modified CIL graph
R_{CIL} – modified rules for CIL graph
Ensure: $S_{[0:N]}$ – sets of sources for each instruction
1: $path := ()$ – empty path
2: **while** $\exists r \in R_{CIL} : applicable(G_{CIL}, r)$ **do**
3: $i := program\_counter(G_{CIL})$ – current instruction
4: $consumed := apply(G_{CIL}, r)$ – apply rule
5: **for all** $id \in consumed$ **do**
6: $S_i := S_i \cup \{(id, path)\}$
7: **end for**
8: $path := path \cdot i$ – update execution path
9: **end while**

Algorithm 3 uses several simple functions: *applicable* tells whether the rule can be matched on the graph, *apply* applies the rule and returns the input IDs (dashed boxes in Figure 9), *program_counter* returns the instruction connected to P.

6.4 CIL Path Generator

We have to examine all valid execution paths to make sure that the HFG is complete. We do this by creating a *regular expression* representing the control flow and then generating all valid words (paths).

We create basic block graph based on the source code, this is a very simple algorithm explained in [10]. The basic block graph can be interpreted as a *finite state machine*, the source code is finite, producing only a finite number of states. We disregard the actual values produced in the blocks that define the actual control flow. Instead, we treat the basic block graph as a finite state machine that accepts a set of words where each word defines a valid order of basic blocks. Therefore the execution paths can be represented by a *regular expression* [18]. Based on the basic blocks, we create a regular expression modeling all the valid execution paths.

Algorithm 4 CIL path generator

Require: R – regular expression
Ensure: W – set of words
1: **for all** $l \in [1 : 3 * length(R)]$ **do**
2: **for all** $w \in \{v : v = e^* \wedge e \in \Sigma \wedge |v| = l\}$ **do**
3: **if** $valid(w, R)$ **then**
4: $W := W \cup \{w\}$
5: **end if**
6: **end for**
7: **end for**

Algorithm 4 generates all valid execution paths, where the function *valid* simply checks if the word is valid according to the regular expression and Σ is the alphabet. We limit the word length, thus limiting the loop iteration count, because multiple loop iterations do not add new information (see Section 6.5 for more details).

6.5 Algorithm Correctness

In the following sections, we discuss the reasoning behind the hybrid flow graph construction presented in Section 6, mainly why it is equivalent to the input source code. Here, we present only the main ideas of a proof, because a complete proof would require too much space.

We have to focus mainly on three parts of the transformation algorithm: 1) the CIL sequential hybrid flow graph is equivalent to the CIL source code 2) the analyzed execution paths contain all the possible inputs 3) the merge nodes created according to the execution paths are placed correctly.

6.6 CIL HFG Equivalence To Code

We prove the equivalence of the CIL code and the CIL hybrid graph (Section 5) by making sure that they have the same control and data flow. We can do this, because the CIL graph contains a node for program counter and nodes for variables and stack.

We base the proof on the fact that the HFG operations behave the same way as the instructions. The program counter is passed from one instruction to another in the same way as in CIL – control flow is the same. The instructions store data using special operations (nodes) for stack and variables in the same way they do in memory – data flow is equivalent. We omit the full proof, because it would be too long and technical.

6.7 Flow Graph Logic

We cannot use the same approach for a general hybrid flow graph as for CIL sequential HFG, because a general HFG does not model the platform specific memory features and control logic. Instead, we focus on the transformation of a CIL sequential HFG to a general HFG. We prove their equivalence by tracing all valid execution paths.

Lemma 1 (Necessary execution paths)**.** *Let W we a set of words produced by the CIL path generator algorithm (Algorithm 4) and $E := \{e* : e \in \Sigma\}$ set of all words, then $\bigcup_{w \in W} reachable(w) = \bigcup_{e \in E} reachable(e)$, where the function reachable represents the* CIL reachability *algorithm.*

Proof. The lemma says that the paths generated by Algorithm 4 produce all possible inputs for all instructions. This is true because there is only a limited number of paths that can produce new inputs.

The basic block graph can be viewed as a finite state machine. The regular language accepted by the state machine can be infinite, but only due to the *Pumping lemma* [19], which means infinite loop iterations.

The important effect of loops is that the instructions use values produced outside the loop in the first iteration and

then they consume values produced in the loop. The only possible exception is a conditional branch in the loop, but we check the paths with both branches taken in the first iteration. □

Lemma 2 (Data flow path merging)**.** *Let pairs* $(S_{[1:N]}, P_{[1:N]})$ *be sources for instruction i, then* $input\_count(i) = N \implies (S_{[1:N]}, i) \in E(HFG)$ *or* $input\_count(i) < N \implies \exists m \in V(HFG) \wedge (S_{[1:N]}, m) \in E(HFG)$

Proof. The lemma says that either all the sources were used to create appropriate edges or that a merge nodes were introduced if there is too many sources.

The first implication is true, because Algorithm 2 creates an edge for every source (lines 17 to 20).

The second implication is correct, because the algorithm creates merge nodes when there is too many sources (line 25). The merge is created based on the execution path stored for each input, it is connected to the last instruction of the prefix shared by both paths (lines 29 to 32). This means that the merge is driven by the conditional branch that changed the execution path and decided what source would be used. □

7 Conclusions

We designed the flow graph to represent a procedural code along with important information about its structure and behavior. We defined the HFG semantics using graph rewriting, which is basically a grammar on graphs. We defined semantics for both the sequential and general HFG using the formalism, thus making the sequential HFG a special version of the general HFG. We designed an algorithm that transforms a sequential source code (C# compiled to CIL) to a general HFG using mainly the HFG semantics and graph algorithms. Therefore, the algorithm can be modified for other platforms.

We have implemented a prototype of the general HFG that uses the defined semantics and is capable of completely parallel computation. The prototype serves as the proof of concept, showing that the HFG is defined correctly. It is implemented using the Bobox framework [16].

Acknowledgments

This paper was partially supported by the Grant Agency of Charles University (GAUK) project 122214 and by the Czech Science Foundation (GACR) project P103/13/08195.

References

[1] "TG3. Common Language Infrastructure (CLI). Standard ECMA-335, June 2005."

[2] Peterson, J. L.: Petri nets. ACM Comput. Surv., **9 (3)** (Sep. 1977), 223–252, [Online]. Available: http://doi.acm.org/10.1145/356698.356702

[3] Gilles, K.: The semantics of a simple language for parallel programming. In Information Processing: Proceedings of the IFIP Congress **74** (1974), 471–475

[4] Josephs, M. B.: Models for data-flow sequential processes. In: Communicating Sequential Processes. The First 25 Years, Springer, 2005, 85–97

[5] Ezpeleta, J., Colom, J. M., Martinez, J.: A Petri net based deadlock prevention policy for flexible manufacturing systems. Robotics and Automation, IEEE Transactions on **11 (2)** (1995), 173–184

[6] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Graph transformation systems. Fundamentals of Algebraic Graph Transformation (2006), 37–71

[7] Corradini, A., Dotti, F. L., Foss, L., Ribeiro, L.: Translating java code to graph transformation systems. In: Graph Transformations, Springer, 2004, 383–398

[8] Geiger, L., Zündorf, A.: Graph based debugging with fujaba. Electr. Notes Theor. Comput. Sci. **72 (2)** (2002), 112

[9] Balasubramanian, D., Narayanan, A., van Buskirk, C., Karsai, G.: The graph rewriting and transformation language: Great. Electronic Communications of the EASST **1** (2007)

[10] Allen, R., Kennedy, K.: Optimizing compilers for modern architectures. Morgan Kaufmann San Francisco, 2002

[11] Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for Java. ACM SIGPLAN Notices **41 (6)** (2006), 387–400

[12] Muchnick, S. S.: Advanced compiler design implementation. Morgan Kaufmann Publishers, 1997

[13] Reps, T.: Program analysis via graph reachability. Information and software technology **40 (11)** (1998), 701–726

[14] Geilen, M., Basten, T.: Requirements on the execution of Kahn process networks. In: Programming languages and systems, Springer, 2003, 319–334

[15] Navarro, A., Asenjo, R., Tabik, S., Cascaval, C.: Analytical modeling of pipeline parallelism. In: Parallel Architectures and Compilation Techniques, 2009, PACT'09, 18th International Conference on, IEEE, 2009, 281–290.

[16] Falt, Z., Kruliš, M., Bednárek, D., Yaghob, J., Zavoral, F.: Locality aware task scheduling in parallel data stream processing. In: Intelligent Distributed Computing VIII, ser. Studies in Computational Intelligence, D. Camacho, L. Braubach, S. Venticinque, and C. Badica, Eds., Springer International Publishing, 2015, **570**, 331–342

[17] Falt, Z., Čermák, M., Dokulil, J., Zavoral, F.: Parallel SPARQL query processing using Bobox. International Journal On Advances in Intelligent Systems **5 (3,4)** (2012), 302–314

[18] McNaughton, R., Yamada, H.: Regular expressions and state graphs for automata, 1960.

[19] Jaffe, J.: A necessary and sufficient pumping lemma for regular languages. ACM SIGACT News **10 (2)** (1978), 48–49