An Intermediate Model for Code Generation from the Two-Hemisphere Model

Konstantins Gusarovs, Oksana Nikiforova
Department of Applied Computer Science
Riga Technical University
Riga, Latvia
email:{konstantins.gusarovs, oksana.nikiforova}@rtu.lv

Abstract—**Nowadays, models are widely used in software engineering. By using different types of models, it is possible to present business requirements, system architecture, test strategies, etc. It is also possible to use models as an input to an automated or semi-automated method that will produce other types of artifacts – other models, statistics, or even software code specified in a programming language. The authors of the present paper work in the area of Model-Driven Software Development (MDSD) by constantly improving the so-called two-hemisphere model that can be used for system modelling and later transformed into several types of artifacts, including Unified Modelling Language diagrams. The goal of the paper is to define an intermediate representation (or model) that can be used for code generation. The present research is the extended and expanded version of the authors' previous work.**

Keywords - two-hemisphere model; model transformation; code generation; model-driven software development.

I. INTRODUCTION

Model-Driven Software Development (MDSD) is one of the advanced approaches to the software development process that is still being developed and adopted by several researchers and enterprises. It seems that nowadays it is possible to distinguish two main groups of MDSD users: those treating models as an analytical tool that can help in better understanding of a problem domain, requirements, etc. [1], as well as those who see models as a high level executable ones that can be further used to produce a programming language code on a target platform [1]. This task can be achieved by using model transformation and code generation techniques, which can be done in an automatic [2] or a semi-automatic way.

The authors of the current paper also treat models as a source for producing the software definition in a chosen programming language and for a chosen platform. While several researchers undertake their efforts to produce a software code from the Unified Modelling Language (UML) [3] defined diagrams (for example, [2][4][5]), the current research is based on another approach to code generation, which is called the two-hemisphere model that has been developed by the authors [6]-[8].

The role of models in Software Development is still unclear, and it can be explained by the fact that MDSD is still at high level of vision [9], and while UML is de-facto industry standard [5], it can be hard for a business analyst, who is not a software engineer, to develop a set of UML diagrams. Even more, most UML diagrams describe the architecture of the system that conforms to object-oriented principles. As an example, the UML class diagram defines a set of classes that form the system, and the UML sequence diagram defines how use cases can be implemented using this set of classes, while the UML communication diagram defines relations between classes from a communication perspective – how classes (or their instances) interact with each other, and what information is passed, etc. It is possible to see that this set of diagrams used as a system analysis model basically corresponds to the core elements of code written in object-oriented programming language.

Nowadays, it is possible to find multiple tools that can be used to transform the system analysis model into the code. The main condition for code generation is that the model should contain both aspects of the system (i.e., static and dynamic), and both should be supported in code generation. Despite several limitations in code generation, which are mainly limitations of the tools rather than the transformation abilities [10], a lot of studies performed since 1980s demonstrate different sets of transformation rules for code generation from UML and mention exactly the dynamic aspect as the primary problem in code generation [11]. Therefore, the authors indicate that by creating a set of UML diagrams, one basically carries out the coding work. In addition, one also has to overcome the difficulties caused by UML usage. As the two-hemisphere model provides an ability to generate UML diagrams, which is enough for code generation, the authors of the paper assume that the two-hemisphere model already contains all the necessary information to get all the required constructions specified in the programming language. That is why in the present research the authors move further away from complex models specified in UML and use their own model. The research also attempts not to focus on the static aspects of the system, i.e., data structures and domain models, but rather on defining the dynamic capabilities of the system by the so-called intermediate model, which, in general, is like the adoption of the two-hemisphere model for the task of code generation.

The goal of the paper is to define an intermediate artifact that will serve as a "bridge" between the initial model (two-hemipshere model) and the target model (source code). Although UML can be used to cover this area, and there are the methods to generate UML diagrams from the two-hemisphere model, the authors would like to mention once again that most UML diagrams cover object-oriented architecture. This raises a need for the intermediate model that is target-architecture-agnostic and can be used to describe both static and dynamic aspects of the system. This model should also serve as a source model for the code generator, which means, it should cover necessary elements of the source code.

The paper is structured as follows. Section II gives an insight into related work. Section III provides a high-level overview of the two-hemisphere model. Section IV discusses the target model, which in this case is a code written in some programming language. Section V describes what is required to define the data structures for the system being built. Section VI covers the definitions that can be used for describing the capabilities of the dynamic system. Section VII provides examples of various applications of the proposed intermediate model along with the analysis of the respective applications. A short demonstration of intermediate model application is presented in Section VIII. Finally, Section IX concludes the paper, as well as provides an insight into the future research to be conducted in this area.

II. Related Work

Having defined requirements for the intermediate model, the authors performed an analysis of the existing approaches to code generation in the MDSD area. Full analysis of the published articles is an area for a separate research itself; therefore, in this article the authors provide a brief overview of related studies.

The first example under consideration is [12]. Its authors propose an extensible intermediate model for code generation from UML sequence diagrams. The article describes metamodel and its possible extensions. The authors claim that their model can be used with different target languages; however, such languages must be object-oriented.

Another example is [13], where the author develops an intermediate model, called Hierarchical Syntax Char (HSC), which is used for the UML activity diagram conversion to the source code. Here, the author has chosen the Java programming language [14] as a target model. The HSC developed by the author once again recognizes the necessity for the object-oriented target language.

Authors of [15] also target object-oriented languages in their research. Even more, the approach described in [15] also defines the architecture of generated code by listing specific components of the system to be generated. Again, an intermediate model is developed to support multiple target platforms.

It is possible to find more studies in this area; however, it seems that they mostly aim at improving the code generation techniques from different types of UML diagrams. For example, studies [2][4][5] focus on code generation from UML diagrams, targeting at object-oriented languages. It should be noted that researchers usually target the object-oriented languages, which could probably be explained by the use of UML, since it already defines basic components of an object-oriented system. This, in turn, leads to limited coverage of target languages that do not support an object-oriented paradigm by the existing methods. Examples of such languages are provided in Section V of this paper. The authors consider that the current state of code generation from models is somehow limited to support only one paradigm, and, therefore, propose the intermediate model described in the paper.

III. The Two-Hemisphere Model: A High-Level Overview

One of the MDSD tasks is transformation from the source model to the target model. The task itself describes a need for at least two models – one that is defined in the beginning and is called the source model. This is an initial artifact that can be produced, for example, by a business analyst, while performing a requirement analysis. Another one is a target model, which can be almost everything, starting with the set of UML diagrams and ending with the software code defined in some programming language.

The authors of the present paper define the two-hemisphere model [6] as a source artifact. The model itself was first introduced in 2004 with the goal of describing the business requirements with as minimal set of diagrams as possible for an object-oriented system analysis. It introduces an idea of joining both static and dynamic aspects of the system in the model that consists of two diagram types. Later, several improvements were introduced to it, enriching the model and precising its elements in [7] and [8], and working on the supporting tool in [16]. The notation of the two-hemisphere model is presented in Figure 1.

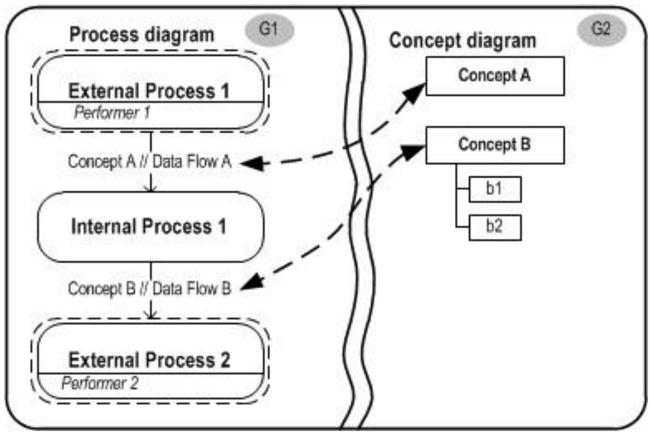

Figure 1. Two-Hemispher model notation.

The two-hemisphere model contains two diagrams:

- Concept model (labeled G2) is a set of concepts or datatypes used throughout the system or a given use case. Each concept has at least its name and a set of 0-n attributes. Each attribute consists of a name and data type, where data type might be a primitive value, such as a number, string literal, boolean type, etc., another concept or array/collection of the aforementioned. The notation of the concept model is similar to the one used for Entity-Relationship (ER) diagram [17], but relationships among concepts is not used as far as they are not meaningful at this level of abstraction and are generated automatically at the level of UML class diagram.

- Process model (labeled G1) is based upon the notation of Data Flow Diagram (DFD) [18] and is composed of two types of elements – processes and data flows. Processes show units of work inside the system. Data flows, in turn, interconnect processes, both defining the sequence of process execution and the data each process receives and produces. Here, data might be 0-n of the same data types, concept attribute use. Thus, the data flow might carry no data at all or a complex set of data. This way both diagrams are interconnected, i.e., concepts appear as the data flow content.

It is possible to see that the definition of the two-hemisphere model does not require specific software engineering knowledge – basically to create it, one has to analyze what business processes take place in the system being built, what data they consume and produce, and in which order it might be executed. Moreover, the two-hemisphere model can be obtained directly from the business domain, where business processes and data flows are somehow or other structured and supported in the form of the model specified in the analogical notation for business issues. Thus, the authors of the paper see this model as a great candidate for the MDSD source model, since the model only describes how the system works, rather than how the system should be built.

After the choice of the source model is performed, it is necessary to define what type of artifact is targeted. Little information on the chosen target model is provided in the next section. This model is nothing else, but the software code written in the chosen programming language, in other words, computer program.

IV. Computer Program: Definition

In order to define the concept of computer program, the authors would like to mention several definitions from ISO/IEC 2382:2015 standard [19] and analyze what is required to transform the source model to it.

First, it is necessary to define what a computer program is at a glance. In ISO/IEC 2382:2105, it is defined as a “syntactic unit that conforms to the rules of a particular programming language and that is composed of declarations and statements or instructions needed to solve a certain function, task, or problem”. By analyzing this definition, it is possible to see that a computer program should consist of the two main parts:

- Declarations that are used to describe data structures and variables that are used to solve the given task.
- Statements or instructions needed for the given task or problem solution. It is also possible to further analyze the standard and conclude that these elements are used to compose the algorithm – “finite ordered set of well-defined rules for the solution of a problem”.

By combining and analyzing these definitions and common knowledge about software engineering, it is possible to define the target model that consists of the two main parts:

- Data structure and variable definitions – to cover the static aspect of the system in development.
- Sequence of instructions or statements that use former part to cover the dynamic aspect of the developed system.

Again, it is possible to see that the chosen source model already provides an insight into these two aspects with the concept model being focused on the data structure definition and the process model describing the dynamic capabilities of the system, i.e., how data are transformed during the system operation, and in which order processes are invoked performing these transformations.

Thus, to transform the two-hemisphere model into the computer program, it would be necessary to transform every element (or a set of elements) of it to the appropriate element (or a set of elements) of the target model and to preserve the linkage between them based on the specific algorithm defined by the authors.

V. Converting Concepts To Data Structures

Data structures describe the static aspect of the system that is being analyzed and built. According to [19], data structure can be defined as “physical or logical relationship among units of data and the data themselves”. This definition can be linked to the concept model of the two-hemisphere model: by representing data types in form of units of data and defining the necessary relationship by utilizing already defined concepts, it is possible to extract all the necessary information to the form required for code generation.

In order to be programming language-agnostic, it is necessary to analyze how data structures might be represented in different programming languages and what information is shared between these representations.

The first case is object-oriented programming languages – here, it is possible to define data structure as a class with appropriate attributes. Each attribute can be described by its name and data type.

Some programming languages, for example, ECMAScript [20], are sometimes called object-based. While the latest ECMAScript standard allows for the definition of classes, it is also possible to use the so-called prototypes for the object blueprint definition. Prototype here is an object that has a set of fields and methods that can be used by all the other objects that are referencing this prototype. In a way, this is like the class in stricter object-oriented languages; however, prototypes usually do not support inheritance. In case of the static aspect description, a prototype should contain a set of fields, where each field has a name and data type.

Next, there are programming languages that are not object-oriented, for example, C programming language [21], where data structures are commonly represented with a `struct` syntax construction. Structure in C language can be viewed as a “weak class” – it is a set of data that consists of fields that are like class attributes. Each of them has a name and data type. However, structures in C language do not have methods (it is possible to reference the function via the pointer, which is a field of structure; however, it still will be a field, but not a method).

Other non-object-oriented programming languages, such as Erlang [22], can have different ways of representing the data structures. For example, in Erlang it is possible to define it as a `record`, which is similar to structure in C language and consists of fields of given types, or it is also possible to define it as a tuple, which can be viewed similar to the array. In case of record, each element has a name and can have a data type. In case of tuple, a name is omitted and replaced by an index; a data type, in turn, can be preserved.

Even using low-level assembly languages, it is usually possible to define the data structures. For example, one of the modern assemblers – flat assembler [23] – provides a way of defining the so-called structures consisting of a field, where each of them has a name and data type definition.

To sum up, it is possible to see that most programming languages require a data structure to have its own type (or name) and a set of fields/attributes, each of them having its own name and data type. Even considering some exclusions, such as Erlang tuples, where names are not preserved, it is possible to define an intermediate representation of a data structure that can be later used to generate a code in different programming languages.

This intermediate representation is provided in (1). Here, the data structure definition is described by its name and a set of attributes, each having its name and data type. Basically, one can notice that such a representation corresponds to the concept definition in a concept model of the two-hemisphere model. Thus, obtaining the data structure information from the source model is a simple task.

$$DS\ Def = \begin{pmatrix} Name \\ (\langle Attr_1 | Type_1 \rangle \ldots \langle Attr_k | Type_k \rangle) \end{pmatrix} \quad (1)$$

Data structure information is the first part of the proposed intermediate model that can be used for code generation. The second part is the information that can be used for describing the behavioral capabilities of the code. Definition of such a model is provided in the subsequent section.

VI. Definitions of System Behavior

System behavior in the two-hemisphere model is described with the help of the process model that provides the information on the processes that are executed inside the analyzed system, the data exchanged by these processes, and the sequence of execution. In order to convert this information to the code defined in a programming language, it is necessary to define a model that is capable of the programming language code description and can represent such a code.

One of the ways to represent the target model, i.e., programming language syntax constructions, is to use Abstract Syntax Tree (AST) [24]. This approach is widely used in the compilers, which translate textual representation of the code into ASTs and then build the machine instruction set out of them. Thus, AST is one of the possible intermediate models that can be used for code generation.

As mentioned above, AST is used to generate the machine instruction set that, in turn, can be represented in a way of the so-called Assembly Language [25], which is human readable representation of the machine instructions. Obviously, it is possible to use a similar approach when defining the dynamic part of the intermediate code representation. However, assembly languages for the modern processors can contain a lot of instructions, for example, x86-64 instruction set consists of ~1000 instructions [26], which would make the intermediate model complex to define correctly, while easy to transform to the appropriate code.

Yet another option to analyze is to look at cross-platform languages, such as Java [14] and .NET [27]. These languages are compiled into the so-called bytecode that can be defined as a “lightweight assembly”. Bytecode provides an alternative to more complex assembly languages by defining a reduced instruction set, for example, Java Virtual Machine (JVM) bytecode consists of ~200 instructions [28].

It is possible to use these representations to define the logic encapsulated in the process model and describe the dynamic aspect of the system under analysis. Though, at first, the use of AST seems to be a correct approach, the authors would like to note that ASTs usually define the syntax of a particular language. Therefore, AST defined for the Java language [14] will probably not be suitable for language such as Erlang [22], since the syntaxes of two are different. However, it might be used for code generation in JavaScript [20] or C [21].

Thus, the authors propose defining the intermediate model by using ideas that define the bytecode – use a reduced set of instructions in order to accomplish the task. It is necessary to define such instructions in a way that they can be used to cover the maximum number of possible target languages.

For this purpose, it is necessary to analyze the two-hemisphere model once again. It is possible to see that the dynamic aspect of the system under analysis is described by processes, each of which might accept and produce data flows. Data flows, in turn, can carry data in form of concepts or primitives. Therefore, the main logical element here is process. Processes can be turned into methods, functions, predicates, etc., depending on the chosen target language. Common characteristic of these targets is that they can have inputs and outputs, which correlate well with the process consuming and producing data flows.

Therefore, it is possible to define an intermediate representation of the process: it should have an identifier (it, for example, can be a name), a set of consumed data, which can be empty, and a set of produced data, which can also be empty. Transformation to such representation from the target model is straightforward – it is necessary to use the process name and collect all the possible data elements from the incoming and outgoing data flows to define the representation shown in (2).

$$P\ Def = \begin{pmatrix} Name \\ (\langle Input_1|Type_1\rangle \ldots \langle Input_k|Type_k\rangle) \\ (\langle Output_1|Type_1\rangle \ldots \langle Output_k|Type_k\rangle) \end{pmatrix} \quad (2)$$

In this representation, each process is transformed into a three-element tuple that consists of name, inputs and outputs. Both inputs and outputs are defined as sets of name-type tuples, where a name is a logical name of the input/output, i.e., parameter name, and a type is a data structure, primitive or array/collection of former types mixed in any way.

Such a representation allows for a wide range of possible target languages – it does not define whether the target language element is a class method or a free function, or any other kind of data processing primitive. It does not enforce a way on how parameters are passed – there are languages, for example, C# [27] or Python [29] that can allow returning multiple data structures from the function/method. Otherwise, it is possible to combine the outputs into a special data structure to guarantee a single returned item.

The authors propose calling this representation a "logical unit", since it corresponds to a single process being executed inside a system; however, its target representation may vary.

In order to define the interaction between the logical units, it is necessary to analyze what might happen inside the system and how processes might interact with each other.

The simplest case is sequential invocation of processes, which takes the data produced by the first processes and passes it to the next one in the logical chain. To cover this case, it is necessary to define storage units for data process exchange – when doing further transformation, these definitions can become local or global variables, virtual or physical machine registers, etc. It is also necessary to be able to invoke any logical unit by passing its parameters to it and storing its result.

Next case is branching – branching in programming languages can be represented by various syntax constructions, starting with `if..else`, `switch` and ending with loops that, in turn, may contain premature exit conditions. It is also necessary to note that loops can be defined in several ways – a loop may have its condition checked before the next iteration execution, or after it. Despite different ways of branching, it is possible to analyze lower-level languages, such as assembly language [25], and different byte code implementations (for example, JVM [28] bytecode and .NET [27] intermediate language) to see that it should be possible to implement the necessary branching support by using several definitions. It should be possible to define labels, which can mark different states (or points) in the execution flows and instructions that would allow passing the control to these labels, i.e., branching instructions. Branching, in turn, can be conditional and non-conditional. In the first case, when the execution flow reaches an appropriate instruction, the so-called jump is performed to the appropriate label, which means a change in the next executed instruction. Conditional branching requires first performing the condition check and then, depending on the result of this check, performing or not performing the "jump".

By analyzing the possible logical unit execution flows, one can see that these two cases are enough to cover all the possible process execution sequences in the initial model. Thus, it is possible to define additional elements that are described further to be generated for the intermediate model.

First of these elements is label definition instruction. It is shown in (3). Here, the label is defined by its name, which can be any kind of symbolic identifier – numeric or textual.

```
Label<Name>
```
(3)

Next elements are branching instructions that are used with the labels. The first branching instruction is non-conditional branching that is shown in (4).

```
Jump<Label>
```
(4)

Non-conditional jump transfers the execution to the label defined in it, so it can be defined by a jump instruction followed by a label to be "jumped" to.

Next two instructions are conditional branching instructions, and they are presented in (5). Both instructions are similar, with only difference in the situation when branching should happen – when the condition is met or is not met.

```
JumpIf<Var, Label>
JumpIfNot<Var, Label>
```
(5)

These instructions require the boolean type variable to be checked. This variable is defined via its name, which will be discussed later. Otherwise, both instructions contain labels to be "jumped" to, depending on the value of this variable.

It is possible to see that conditions here are not the part of the branching instruction; instead branching instructions use variables that contain the result of the condition check. This means the necessity for the condition checking instruction, which is given in (6).

```
Check<Var, Condition>
```
(6)

This instruction has two parameters – a variable to store the condition check result and the condition to be checked. Here, the condition is a free-text phrase or a sentence.

It is possible to see that condition checking requires a variable to store the result, which later will be used by a conditional branching instruction. Therefore, it is necessary to be able to define the variable, which is supported by variable definition instruction presented in (7).

```
Var<Name, Type>
```
(7)

This instruction has two arguments – the name of variable and its type, which is the same as for data structures.

Last necessary instruction is the process invocation instruction. It is presented in (8).

```
Invoke<Process, Inputs, Outputs>     (8)
```

Here, the instruction has three parameters defined in it – the name of the process to be executed, its inputs, which are an array of appropriate variable names, and its outputs defined in the same way.

As it will be shown in the next section, these instructions are enough to define all the types of branching and possible execution flows, at least in the context of code generation from the two-hemisphere model.

VII. Examples of the Proposed Model Applications

In order to prove that the developed model is feasible and can be used for code generation, the authors propose analyzing several examples of its application.

The first example is sequential invocation of processes. In case of the Java [14] programming language, such a code can be written in a form shown in Figure 2.

```
c = f1(a, b);
d = f2(c);
```

Figure 2. Sequential process invocation in Java.

Here, method `f1` is invoked with arguments `a` and `b`, its invocation result is stored in variable `c`, and then used to invoke method `f2`. It is possible to define such an invocation sequence in the proposed intermediate model notation, which, in turn, is shown in Figure 3.

```
Invoke<f1, [a, b], [c]>
Invoke<f2, [c], [d]>
```

Figure 3. Sequential process invocation in the proposed model.

One can realize that the proposed model corresponds to the Java code, and it is possible to perform transformations from one to another.

Next example is presented in Figure 4. Here, several branching definitions are given – first, there is simple branching with only single condition check, which defines if method `f1` should be executed. Next, there is more complex `if..else` branching, and finally – branching using `switch`.

The same branching instructions are presented in Figure 5. Again, by studying both representations, it is possible to see their equality and ability to transform from one to another.

```
if (a == b) {
    f1();
}

if (c < d) {
    f2();
} else if (c == d) {
    f3();
} else {
    f4();
}

switch (e) {
    case 1:
        f5();
        break;
    case 2:
        f6();
        break;
    default:
        f7();
}
```

Figure 4. Branching in Java.

```
Check<Cond1, "a == b">
JumpIfNot<Cond1, L1>
Invoke<f1, [], []>

Label<L1>
Check<Cond2, "c < d">
Check<Cond3, "c == d">
JumpIfNot<Cond2, L2>
Invoke<f2, [], []>
Jump<L4>
Label<L2>
JumpIfNot<Cond3, L3>
Invoke<f3, [], []>
Jump<L4>
Label<L3>
Invoke<f4, [], []>
Label<L4>

Check<Cond4, "e == 1">
JumpIfNot<Cond4, L5>
Invoke<f5, [], []>
Jump<L7>
Label<L5>
Check<Cond5, "e == 2">
JumpIfNot<Cond5, L6>
Jump<L7>
Invoke<f6, [], []>
Label<L6>
Invoke<f7, [], []>
Label<L7>
```

Figure 5. Branching in the intermediate model.

Last situation to be covered by the proposed intermediate model is loops in the code. To show that these cases can also be covered, the authors propose considering the Java code provided in Figure 6. The appropriate intermediate model representation is given in Figure 7.

Here, three types of loops are presented. The first loop is `for loop`, which repeats for a given amount of time. This is controlled via a local loop variable `i`. The second loop is a loop with precondition, while the last one is a loop with post-condition. The second loop also involves possible premature exit via checking local variable `c` value.

It is possible once again to see that all the necessary cases are covered by the intermediate model with a single exception of incrementing the loop variable value in case of the loop with fixed iteration count. This, however, can be improved by adding additional instructions to the model. It is also worth noting that the two-hemisphere model notation does not allow defining such loops now, so this case is not covered fully.

```
for (int i = 1; i < 5; i++) {
    f1();
}

while (a < b) {
    if (c > 0) {
        break;
    }

   f2();
}

do {
    f3();
} (while e > 1);
```

Figure 6. Loops in Java.

It is possible to see that the proposed intermediate model allows covering all the possible cases that might be encountered in the initial model, as well as presents a solid way to enable code generation in various programming languages.

VIII. An Example of the Proposed Model Application

In order to demonstrate how the proposed model can be used in conjunction with the two-hemisphere model, the authors refer to the diagram first presented in [30]. Due to the fact that the research described here is still underway, the authors do not present a full system. Instead, the authors demonstrate only part of it that was used to evaluate the approach. As in the original work, only a process diagram is analyzed here. It is presented in Figure 8.

```
Label<L1>
Check<Cond1, "i < 5">
JumpIfNot<Cond1, L2>
Invoke<f1, [], []>
Jump<L1>
Label<L2>

Label<L3>
Check<Cond2, "a < b">
JumpIfNot<Cond2, L4>
Check<Cond3, "c > 0">
JumpIf<Cond3, L4>
Invoke<f2, [], []>
Jump<L3>
Label<L4>

Label<L5>
Invoke<f3, [], []>
Check<Cond4, "e > 1">
JumpIfNot<Cond4, L6>
Jump<L5>
Label<L6>
```

Figure 7. Loops in the intermediate model.

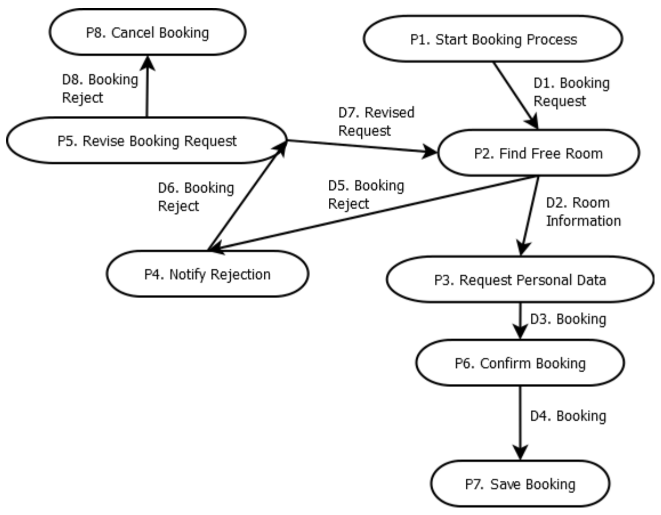

Figure 8. Example of the two-hemisphere model.

Here, the model describes a booking process in the hotel that starts with receiving a booking request with room preference details. After that there are two options: either room that fits the request is found or not (for example, due to the fact that rooms do not meet the criteria, or non-availability of the rooms in the given dates). If no room is found, a user is advised to revise the information and submit a new request. At this point, a user can also cancel the booking. If a room is found and request can be served, a user is asked to provide additional information, which is used to create the booking and store it in the database. Here, all elements are marked with identifiers – processes are marked P1…P8, dataflows – D1…D8. These identifiers are later used in the intermediate model that is presented in Figure 9. Here, it is possible to see how the intermediate model would look after transformation of the initial process model. It is also possible to see that additional information is required to produce it – such as conditions for branching.

```
Invoke<P1, [], [D1]>

Label<L1>
Invoke<P2, [D1, D7], [D2, D5]>
Check<Rejected,
      "booking is rejected">
JumpIfNot<Rejected, L2>
Invoke<P4, [D5], [D6]>
Invoke<P5, [D6], [D7, D8]>
Check<Canceled, "User canceled">
JumpIf<Canceled, L3>
Jump<L1>

Label<L2>
Invoke<P3, [D2], [D3]>
Invoke<P6, [D3], [D4]>
Invoke<P7, [D4], []>
Jump<L4>

Label<L3>
Invoke<P8, [D8], []>

Label<L4>
```

Figure 9. Example of the intermediate model.

It is also possible to trace the intermediate model back to the initial one and see that processes are invoked in the same sequence as defined by the initial business process analysis. This task could also be automated – it is possible to create a graph of all the possible branching and compare it with the initial model in order to check, if the defined process invocation sequence is preserved. Such a graph definition would allow verifying the correctness of the generated model. However, the algorithm to define such a verification graph is out of scope of this paper. However, it should be noted that it has already been developed and currently is under testing.

IX. Conclusions and Future Work

Previous research conducted by the authors on the use of the two-hemisphere model for generation of different types of UML diagrams, such as use case, sequence, communication, state or class diagrams, has demonstrated that the two-hemisphere model contains quite enough information to obtain the static elements of the system analysis model, as well as dynamic ones. The received UML model, according to the main statement of MDSD, provides an ability to generate a code as well. So far, as we have a transformation chain: the two-hemisphere model → UML diagrams → code, the authors can assume that the direct transformation, i.e., the two-hemisphere model → code is also feasible. In this paper, the authors have presented the intermediate model that can be used to enable direct code generation from the two-hemisphere model. The proposed model allows for code generation in different programming languages – object-oriented, object-based, procedural, etc.

While this paper describes how the model should look like and what artifacts it consists of, it is also necessary to define algorithms for transformation of the initial model to the intermediate one. This is the first part of future work. It might also be necessary to enrich the model itself to cover more cases, as well as develop algorithms for transforming this model into an actual code. This is also part of future research in this area.

Since the intermediate model described here is still being developed and the research about its definition and application is still being carried out, the authors define the evaluation of the proposed approach and additional validation of the achieved results as another part of future work. The goal is to test this model with a completely developed system and identify the possible gaps and improvement areas.

So far, the goal has been to develop the intermediate model as a basis for code generation. The proposed model covers both static and dynamic aspects of the system and should be compatible not only with the two-hemisphere model, but also with other types of the source model, since the model itself is simple enough to be generated from any type of initial data. The authors also consider the proposed model to be useful for code generation in different programming languages, since it does not enforce any paradigm to be applied and can be used to generate data structures and invocation flows of different types.

References

[1] B. Perisic, “Model Driven Software Development – State of the Art and Perspectives”, Invited Paper, INFOTEH 2014, Proceedings Vol. 13, pp. 1237-1248, 2014.

[2] F. Daniel and M. Matera, “Model-Driven Software Development,” in Mashups. Data Centric Systems and Applications, 1st ed., Berlin: Springer-Verlag Berlin Heidelberg, pp. 71-93, 2014.

[3] OMG® Unified Modeling Language® (OMG UML®), OMG [Online]. Available: https://www.omg.org/spec/UML/ [retrieved: September, 2019]

[4] M. K. Shiferaw and A. K. Jena, “Code Generator for Model-Driven Software Development Using UML Models” 2018 Second International Conference on Electronics, Communication and Aerospace Technology (ICECA), pp. 1671-1678, 2018.

[5] H. D. Gurad and V. S. Mahalle, “An Approach to Code Generation from UML Diagrams”, IJESRT - International Journal of Engineering Sciences & Research Technology, pp. 421-423, 2014.

[6] O. Nikiforova and M. Kirikova, “Two-hemisphere model Driven Approach: Engineering Based Software Development”, Scientific Proceedings of CAiSE 2004 (the 16th International Conference on Advanced Information Systems Engineering), pp. 219-233, 2004.

[7] O. Nikiforova, “Two Hemisphere Model Driven Approach for Generation of UML Class Diagram in the Context of MDA”,

e-Informatica Software Engineering Journal - Volume 3, Issue 1, pp. 59-72, 2009.

[8] O. Nikiforova, “System Modeling in UML with Two-Hemisphere Model Driven Approach”, Proceedings of The 50th Scientific Conference of Riga Technical University, Computer Science, Applied Computer Systems, pp. 37-44, 2010

[9] A. Noureen, A. Amjad, and F. Azam, “Model Driven Architecture - Issues, Challenges and Future Directions,” JSW, vol. 11, No. 9, pp. 924-933, 2016.

[10] J. Sejans and O. Nikiforova, “Practical Experiments with Code Generation from the UML Class Diagram”, Proceedings of MDA&MDSD 2011, 3rd International Workshop on Model Driven Architecture and Modeling Driven Software Development In conjunction with the 6th International Conference on Evaluation of Novel Approaches to Software Engineering, pp. 57-67, 2011.

[11] O. Nikiforova, “Object Interaction as a Central Component of Object-Oriented System Analysis”, Proceedings of the 2nd International Workshop „Model Driven Architecture and Modeling Theory Driven Development” (MDA&MTDD 2010), pp. 3-12, 2010.

[12] E. B. Omar, B. Brahim, and G. Taoufiq, “Automatic code generation by model transformation from sequence diagram of system’s internal behavior”, International Journal of Computer and Information Technology Vol. 01 Issue 02, pp. 129-146, 2012.

[13] Z. Wang, “A JAVA Code Generation Method based on XUML”, IOP Conference Series: Materials Science and Engineering, pp 1-8, 2019.

[14] Java | Oracle [Online]. Available: https://java.com/ [retrieved: September, 2019]

[15] A. Lasbahani, M. Chhiba, and A. Tabyaoui, “A UML Profile for Security and Code Generation”, International Journal of Electrical and Computer Engineering (IJECE), pp 5278-5291, 2018.

[16] O. Nikiforova, U. Sukovskis, and K. Gusarovs, “Application of the Two-Hemisphere Model Supported by BrainTool: Football Game Simulation”, Proceedings of the 4th Symposium on Computer Languages, Implementations and Tools, organized within the International Conference of Numerical Analysis and Applied Mathematics (ICNAAM 2014), pp. 1-4, 2014

[17] P. Chen, “The Entity-Relationship Model - Toward a Unified View of Data”, ACM Transactions on Database Systems, pp. 9-36, 1976.

[18] W. Stevens, G. Myers, and L. Constantine, "Structured Design". IBM Systems Journal. 1974, vol.13, no.2, pp.115-139, 1974.

[19] *ISO/IEC 2382:2015 Information technology -- Vocabulary*. [Online]. Available from: https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:ed-1:v1:en [retrieved: September, 2019]

[20] *Standard ECMA-262* [Online]. Available: https://www.ecma-international.org/publications/standards/Ecma-262.htm [retrieved: September, 2019]

[21] D. M. Ritchie and B. W. Kernighan, The C Programming Language, Second Edition. - USA: Prentice Hall, 1988.

[22] *Erlang Programming Language* [Online]. Available: https://www.erlang.org/ [retrieved: September, 2019]

[23] *flat assembler* [Online]. Available: https://flatassembler.net/ [retrieved: September, 2019]

[24] D. Grune and C.J.H Jacobs, Parsing Techniques – a Practical Guide. - USA: Prentice Hall, 1988.

[25] D. Salomon, Assemblers and Loaders. - USA: Prentice Hall, 1993.

[26] Intel® 64 and IA-32 Architectures Software Developer Manuals | Intel® Software [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm [retrieved: September, 2019]

[27] .NET | Free. Cross-platform. Open Source. [Online]. Available: https://www.microsoft.com/net/ [retrieved: September, 2019]

[28] The Java® Virtual Machine Specification [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se12/html/index.html [retrieved: September, 2019]

[29] Welcome to Python.org [Online]. Available: https://www.python.org/ [retrieved: September, 2019]

[30] K. Gusarovs and O. Nikiforova, “Workflow Generation from the Two-Hemisphere Model”, Applied Computer Systems, Vol.22, pp. 36-46, 2017.