Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

Modelling of Language Syntax and Semantics: The Case of the Assembler Compiler

Vadim Zaytsev[a]

a. Raincode Labs, Brussels, Belgium

Abstract Application of software language technologies, whether analytical, transformational, or generational, in an industrial context is usually a taxing endeavour, with high demands in qualification levels of developers involved in it. Yet, if applied successfully, in the right places and with the right amount of effort, they promise high returns in terms of optimisation, effectiveness, validity and verifiability. In this paper, we report on our experience on writing a compiler for a complex second generation legacy programming language originally intended to be used on a mainframe. The business case for this product deals with companies migrating their software systems off the mainframe to cloud native or PC. Leveraging the documentation, available domain knowledge, several sample projects and a test suite, as well as several proprietary DSLs, we successfully modelled syntax and semantics of hundreds of instructions of that language, to the point of producing a compiler with a very limited group of compiler developers in limited time. The compiler is currently deployed at some of our customers and has received a top technology award from Microsoft.

This report is meant to serve as a sample snapshot of how compilers can be built in the industry with software language engineering techniques. Traditional problems of compiler construction such as parsing or code optimisation either did not present a noticeable challenge or did not manifest themselves altogether in the course of this project, but MDE matters such as model transformation, modular design, the use of DSLs and meta-tools, were a constant concern. The focus of the report is in truthful representation of the domain as well as the details of the project, on reflection of the choices that were taken or could have been taken in the meantime, and on lessons learnt during the project.

Keywords Syntax; semantics; legacy systems; knowledge extraction; experience report; software language engineering.

Vadim Zaytsev. *Modelling of Language Syntax and Semantics: The Case of the Assembler Compiler*. Licensed under Attribution 4.0 International (CC BY 4.0). In Journal of Object Technology, vol. 19, no. 2, 2020, pages 5:1–22. doi:10.5381/jot.2020.19.2.a5

1 Introduction

The Raincode ASM370 compiler [Rai16] is a product that can be used to run programs written in the IBM mainframe assembler, on PCs or servers with .NET Framework or .NET Core. It is a proper compiler, since its input is a program text written in assembler, and it treats this assembler program as a normal compiler would have treated a program in any other *high level* language: parses it, constructs an intermediate model of it, annotates it, transforms the model and finally produces an executable file. (This goes against the industrial state of the art in modernisation of second generation languages, which usually entails mapping assembler instructions to statements in some other low level language such as C [Mic, War13, WZH04, B$^+$, Sou, War01, JSW99, War99] or semi-automated extraction of some sort of abstract models that could potentially guide system redevelopment [LB96, War00]). The technologies to implement it, are proprietary metaprogramming DSLs [Bla95, Bla01], as well as native .NET languages like C$^\sharp$ and framework-specific software languages like the LINQ language extension [MBB06], the Roslyn framework [NPC$^+$19], C$^\sharp$ libraries, WPF event handling API [Mic06], XAML user interface definitions [Mic08], etc. The development of the compiler involved overcoming many challenges like reimplementing proprietary macros, solving performance issues and dealing with self-modifying code [Zay17b], but ultimately led to a product that satisfies customers [Rai16] and wins awards [Pre16]. In this paper we focus on the modelling aspect and follow how the syntax and semantics of the ASM370 was formulated, extracted, transformed and finally evolved to enhance the already working compiler to target yet another platform.

The paper is structured as follows: subsection 1.1 goes deeper into the background of the issue, explains what ASM370 is, how it is used, why have we embarked on the journey of building a compiler for it in the first place, and what were the main abstract challenges of this endeavour; section 2 motivates the metamodel(s) for modelling syntax and semantics of the elements of the instruction set of the chosen language and explains how different elements of the models were manifesting themselves and were being used at various stages of development and execution; section 3 explains the first model transformation step where the syntax models conforming to the desired metamodel were extracted from faulty and only semi-structured documentation, debugged and corrected; section 4 switches to the semantic models and provides details on how their metamodel was designed and what is its link to the artefacts that eventually needed to be generated from the models; section 5 revisits a number of related research endeavours that either were used in this project or go in parallel to it but could be profitably combined into it. Finally, section 6 summarises the entire project in a somewhat verbose way in an attempt to provide valuable lessons to be learnt for future industrial projects of similar nature, or some tension points and open problems for future academic projects to solve and address.

1.1 Background and Problem Statement

High Level Assembler (HLASM from this point on) is a *second generation language* available on IBM mainframes (Figure 1). The other classical “generations” are the first (raw machine code), the fourth (DSLs) and the third (a catch-all category for all other general purpose software languages for programming, modelling, data definition, action description, screen specification, batch processing, and so on) [BJZ16, Zay17c]. Notwithstanding widespread third generation alternatives like COBOL or PL/I, HLASM is being used for a number of reasons, some typical arguments for a lower

Figure 1 – IBM System/360, the mainframe for which Basic Assembler Language (BAL) was originally developed in 1964. BAL went through several versions and gained the name HLASM in 1992. For detailed history of the IBM assemblers, please refer to Blagodarov et al. [BJZ16] The photo was taken by the paper author at Deutches Museum during MoDELS 2019 in Munich.

level language (fine-grained or bespoke memory management, error handling, tailoring and optimisations, bit-level interoperability), others being legacy consequences of the software development process (e.g., avoiding the costs of a 3GL/4GL compiler) [BJZ16]. The main reasons for us to develop an HLASM compiler was to provide the option for our customers to migrate their existing HLASM codebase in the scope of a global migration off the mainframe into PC, Azure, Cloud-Native, etc. In such cases this option would allow them to ignore the presence of HLASM assets in their codebase until better times, and use a modern IDE later to help reverse engineering it and rewriting it with modern technologies. These HLASM assets are not meant to be kept operational forever: usually we see barely several thousands lines of HLASM within a typical portfolio of 20–200 million lines of code in higher level languages (there are known cases up to 343 MLOC at Bank of New York Mellon [Mit12], but the largest portfolio to have been migrated by Raincode Labs, is 250 MLOC [Rai19]).

HLASM code typically covers basic features like date and format conversions, with a great fan-in (many other components relying on them extensively, up to millions calls per program). Such a setup means that replacement of HLASM components is feasible but extremely dangerous without proper tool support for testing and refactoring, and is undesirable to undertake *during* massive migration where resources are inevitably running thin anyway. In rare cases with enough inner administrative support it is possible to take the quality-first approach and work on improving the quality of the existing production codebase prior to migration [WPP$^+$19], but it is more common to postpone such activities until after the migration is complete [Fea04].

1.2 Main Challenges and Approaches

On the grand scheme of things, the two main engineering complications in implementing HLASM as a compiler, are its massive instruction set and its equally massive and wickedly flexible macro system. Good sources of information about both of them, can be found in IBM's official documentation: *Principle of Operation* [IBM17] for the instruction set (and a general overview of the z/Architecture), and *General Information* [IBM13] for the macro language (and other HLASM extensions). For the purpose of limiting the scope of this report, we focus here on the former.

The most noticeable challenges faced by the project, were the following:

- Dealing with legacy ecosystems: e.g., language design choices were motivated by punchcard-era technology and are hard to link to anything explainable nowadays.
- Customer inflexibility: given the extremely fragile nature of assembler code and the significance of this code to its owner, there were no compromises to be made in technical choices within the system as well as within the migration process.
- Efficiently executing low-level code written with the use of peculiar idioms up to and including self-modification at runtime.
- The massive scale of the instruction set: while typical high-level programming languages have 20–50 major constructs, HLASM was treated as a high-level language but contained almost a thousand of instructions, each of which had to be examined and implemented individually.
- Non-orthogonality of the instruction set: conceptually similar instructions tend to have subtly and counter-intuitively varying semantics.
- Artefact unavailability: we had no direct access to the original compiler for which our product was meant to be a replacement, and for legal reasons were not allowed to consult its source code, its other components (such as macro definitions) or existing alternative reimplementations.
- Lack of automation for initial steps: instead of tangible verifiable executable models as a starting point, we had to rely on documentation which was manually written, incomplete, contained errors and explicitly (legally) prohibited automatic derivations of commercial artefacts from it, even when it was possible.

In order to face them, we:

- Combined *compilation* (code generation) and *interpretation* (emulation) in one language processor [Zay17b].
- Used *notation-parametric recovery* [Zay12] and *semiparsing* [Zay14] to extract initial data ready for manual curation.
- Inferred *models of syntax and semantics* of HLASM [IBM13, IBM17] from available sources and *generated* compiler [BJZ16, GZ19] components from them.
- Applied *model repair* [CBSK12], *model refinement* [PMBM98] and *model transformation* [BP96] to improve our models.
- Designed several tool-supported *domain-specific languages* [MHS05] to express both commonalities and peculiarities in syntax and semantics.

ICM $R_1, M_3, D_2(B_2)$ [RS-b]

1 0 1 1	1 1 1 1	␣ ␣ ␣ ␣	␣ ␣ ␣ ␣	␣ ␣ ␣ ␣	␣ ␣ ␣ ␣	␣ ␣ ␣ ␣	␣ ␣ ␣ ␣
0xBF		R_1	M_3	B_2	D_2		

ICMY $R_1, M_3, D_2(B_2)$ [RSY-b]

1 1 1 0	1 0 1 1	␣ ␣ ␣ ␣	␣ ␣ ␣ ␣	␣ ␣ ␣ ␣	␣ ␣ ␣ ␣	␣ ␣ ␣ ␣	␣ ␣ ␣ ␣	␣ ␣ ␣ ␣	␣ ␣ ␣ ␣	1 0 0 0	0 0 0 1
0xEB		R_1	M_3	B_2	DL_2			DH_2		0x81	

ICMH $R_1, M_3, D_2(B_2)$ [RSY-b]

1 1 1 0	1 0 1 1	␣ ␣ ␣ ␣	␣ ␣ ␣ ␣	␣ ␣ ␣ ␣	␣ ␣ ␣ ␣	␣ ␣ ␣ ␣	␣ ␣ ␣ ␣	␣ ␣ ␣ ␣	␣ ␣ ␣ ␣	1 0 0 0	0 0 0 0
0xEB		R_1	M_3	B_2	DL_2			DH_2		0x80	

Figure 2 – Three different variants of the `INSERT CHARACTER UNDER MASK` instruction [IBM17, pp.7-262–7-263]

2 Modelling Syntax and Semantics of an Instruction

The main instruction set (basic instructions + extended mnemonics) of HLASM consists of 953 individual instructions, all of them described in the latest release of *Principles of Operation* [IBM17], a 1902 pages giant of a book. Our first attempt at modelling all of them was to classify each manually according to its syntax and semantics, into one or more of general classes like "addition" or "floating point" and inferring the final picture by composing known fragments. Unfortunately, this endeavour relatively quickly came to a halt due to our overestimation of the orthogonality of the language and underestimation of the scale. We have seen cases where each combination of conceptually different classes had to be uniquely implemented, effectively nullifying the non-explosion contribution of model composition. We have seen cases where there would be an addition instruction of a particular subkind but not a corresponding subtraction instruction, and the opcode that would have been logical for such a subtraction instruction to have, was devoted to something entirely unrelated. We have seen cases where within the same group different instructions were assigning a "condition code" (the result code of an operation) with different strategies or not assigning it at all. This all prevented the straightforward "top-down" remodelling of the instruction set and forced us to look more individually at the most relevant of them, slowly broadening our scope whenever possible and in general working our way upwards from the classification and properties already listed in the documentation or apparently required for the artefacts that were to be generated.

Despite the fact that we mostly treat HLASM as a high level language, it is an assembler with all the consequences attached to it — in particular, it allows to treat code as data and data as code. Hence, a program can read its own parts as data (so we need to imitate the memory model at runtime) and can alter its own code by simply writing over it (so we have to retain the knowledge of how to parse bytes into instructions at runtime as well). It is important to know this at this point of the story because it means that beside the traditional source-to-code compiler we have to generate an emulator capable of parsing and executing any instruction given its address in memory. The actual compiler will then resort to calling the emulator when the analysis of the compiled code indicates this necessity (i.e., when the code is modifying itself).

As an example, consider the `INSERT CHARACTER UNDER MASK` instruction, which structure is depicted on Figure 2. It has three variants, distinguished by programmers with the use of the right mnemonic: "ICM" is used for the basic version, "ICMY" for the version that uses a longer 20 bit displacement and "ICMH" for the ASM 370 version that "inserts" its "characters" into the higher 32 bits of the 64 bit register R_1 (otherwise all 32 bit operations are assumed to operate on the lower 32 bits of 64 bit registers). The formula "$R_1, M_3, D_2(B_2)$" after the mnemonic also refers to the encoding used by the programmers: it means that if a programmer writes "ICM 1,2,3(4)", it means the first register, the fourth base, a displacement of 3 and a mask equal to 2. Note that the order of operands is different for the programmer, for the emulator and for the description (we have no explanation for this difference, but learnt to accept legacy systems as they are and not as they should be; there was probably a very good reason for it at design time). So when the text of the description of the instruction's behaviour mentions its "second argument", they mean $D_2(B_2)$, which is the third of the arguments that the programmer writes down. Also, the programmer writes it down as a displacement with the base following it in parenthesis, while the memory layout puts the base before the displacement (which in turn can have its higher bits positioned even lower for the ICMY and ICMH variants). Handling three numbering schemes is too error-prone to be done manually, so we extract the knowledge of them, save it in the models and generate artefacts from it automatically as a way to avoid any mistakes when handling all 952 instructions.

To the right of the programmer's notation, we see "[RS-b]" or "[RSY-b]", which is the so-called "format" of the instruction. All instructions of the format RS-b are structured similarly: they start with eight bits of the fixed opcode (different per instruction variant), have the next four bits for the register, then the mask, then the base and then the displacement. Dealing with 60 reusable formats is easier than dealing with 952 individual formats, even though the documentation contains errors and slight inconsistencies about them (will be elaborated in section 3).

The little bit positioning scheme shows for each format, which bits correspond to which argument. These arguments are still too low level for modelling instruction behaviours: for example, the base and the displacement are never used independently, they are two parts of one conceptual entity representing an address in the memory. Hence, if we just implement the correct mapping between the format's bitwise parts and the conceptual runtime entities, we will be able to let ICM and ICMY to have exactly the same semantics. (The only difference is how the displacement is calculated, which is irrelevant when we already think on the level of addresses, registers and masks).

Figure 3 can help us see the information flows though the solution. The *Principles of Operation* [IBM17], broadly speaking, has four interested sources of information: chapter 5 contains its definition of formats, from which we extract intermediate format models to be tested, validated, completed and finally used to build a model of each instruction's syntax; chapters 7–20 contain many natural text sections, 381 for the fourth edition we mostly used in our project, which had to be consulted regularly to resolve ambiguities as well as to build models of each instruction's behaviour; appendix B contained several tables (called "lists") with basic information such as instruction's name, mnemonic, opcode and format, all ending up in each instruction's syntax model; and appendix C with another table concerning different strategies to calculate condition codes—the two-byte result codes— which will be explained in more detail in section 3.

The condition code (CC) models, together with the models of emulator's desired behaviour, lead to generating the emulator code, which forms a part of the runtime that both the compiler and the compiled program will use. Since the emulator needs

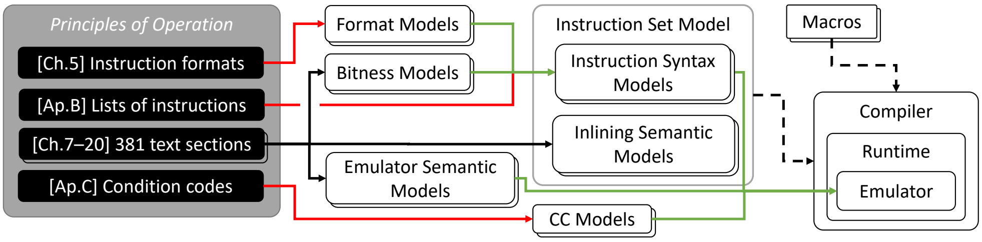

Figure 3 – A megamodel of the project, showing information sources, different kinds of models, relations among them and the final artefacts of the product. Dashed lines represent "used by" relationships, solid lines are model transformations: black ones are mostly manual effort; red ones (or dark grey in monochrome) are semi-automatic with error correction or tolerance built in; green (or light grey) are fully automated transformations. The artefacts on the far left are parts of a volatile manually created PDF. The artefacts on the far right are code. All artefacts in the middle with a "model" in their names are proper models, each having a formally defined metamodel to which they conform.

to parse the instruction from memory bytes, its code also incorporates the knowledge about syntax. The model of the instruction set (in XML) together with the definitions of all the used macros (in HLASM), is provided to the compiler together with the source program. This is done to increase configurability at the customer's site where they might want to turn off support for certain instructions without recompiling the compiler. Not shown on the diagram is documentation, derived from all the models put together and rendered in a human-readable form to simplify debugging and providing visual aid to our compiler developers.

The final model of the instruction set contains, for each instruction, the following components. **Name**, such as "insert character under mask" or "branch and link", as a short description of the instruction, intended for human comprehension. Besides naming a group of conceptually (but not always implementationally!) related instructions, it is essentially useless for the HLASM compiler except for the runtime if logging is turned on (this was used to test previously unseen customer code at customer premises outside our usual development environment). **Mnemonic**, such as "`ICM`" or "`BAL`", is intended for the HLASM programmers and used in the parser that reads an HLASM program in order to recognise instructions, macros and commands it uses. Please note that the compiler frontend uses a parser model and does not encode the syntactic structure explicitly in the parsing algorithm as some other compilers do: as indicated earlier, we need to retain the possibility of tweaking the instruction set at the customer's side without recompiling the compiler. Next is **operation code** (or "opcode" for short), such as `0xBF` for `ICM` or `0xEB81` for `ICMY`. The opcode is a bytecode-level representation of the instruction, that identifies it for the CPU (as well as for our emulator) and thus must also be used as a part of the code generation. This is known as "instruction decoding" [Kli19b]. "**Inlining**" is the name we use to refer to the "true" compilation where an input instruction is compiled to one or more individual atomic bytecode instructions of the target platform. For instructions that allow inlining, their model in the instruction set includes the explicit inlining conditions and the code template to be generated. This is required to enable customisation at the user side to switch inlining on or off per instruction for each particular customer. The microcode DSL will be explained in section 4.

3 Model Extraction

Now that we know what kind of information we need in our models and where we can get it from, let us focus on the information extraction process itself. We extracted the initial models directly from the HLASM manual PDF [IBM17] with an ad hoc developed technique similar to grammar extraction [LZ11, Zay12]: tolerant error-correcting heuristic-based semiparsing of unstructured data with its subsequent curation [Zay14]. As shown on Figure 3, of particular interest for that process are several multi-page tables in appendix B of the *Principles of Operation* [IBM17] that contain instruction summaries, coupling the name (intended for human comprehension, such as "branch and link" or "compute message authenticating code") with the mnemonic (intended for programmers and the parser, like `BAL` or `KMAC`), the opcode (hexadecimal identification of an instruction, spanning one or more bytes) and "characteristics" (97 flags describing frequently occurring behaviour like raising particular kinds of exceptions or having a particular bitwise format). Appendix C contains another useful 6-page table that defines per instruction the strategy used to set the condition code depending on the result of the local computation (an example strategy could be "0 if zero, 1 if negative, 2 if positive, 3 if overflow"). There are 88 such different strategies in total. Each of the tables contained a small number of mistakes and inconsistencies, so we cannot honestly claim to have derived the resulting models from the documentation, but rather reconstructed by heavily relying on several official information sources.

We found and (manually) fixed some errors and inconsistencies in the documentation, most commonly referred to formats of instructions. The problems mostly fell into one of the five categories:

- lacking formats that use only the first 8 bits for the opcode instead of 16 announced (three cases of the S format);
- referencing non-existing formats (e.g., RRF instead of RRF-c; or RSL on multiple occasions instead of RSL-a or RSL-b) which were used in earlier versions of the same document and have not been properly replaced [IBM04];
- slightly misstating a different format within the same group such as RRF-e instead of RRF-c (19 cases), RRF-b instead of RRF-a (4 cases), RRF-e instead of RRF-b (2 cases) or RRF-a instead of RRF-b (1 case), possibly related to plans of finding some uses for extra arguments but ending up not using them in the final version;
- lacking formats with varying uses of a register field (RR and RRE have variants with just one register instead of two for `SPM` and `IPM`; RR has a variant with a mask instead of a register for `BCR`);
- lacking formats for having an immediate value occupy the space normally allocated to an address field (10 cases of RSY-a) or to several fields (8 cases of RS-a).

Of these five, the first two are the most severe and not tolerably ignorable: the first one will yield undefined behaviour depending on the implementation of the final model-to-code transformation; the second one will even leave models non-well-formed. The third and the fourth categories would lead to performance problems: in the third case, data will be fetched without need, and in the fourth case, data will be fetched in a wrong format (and will need to be refined explicitly as an additional semantic step, adding to the modeller's manual effort). The last case is the most dangerous of all, since it can lead to not only unnecessary data fetches, but also fetches from incorrect addresses, occasionally leading to access violations at runtime in a very difficult to reproduce fashion.

One of the useful model properties that was not coded explicitly in any of the tables in the original documentation but was possible to extract manually after reading the textual descriptions of all instructions, was the *bitness* of their arguments — i.e., for each of the arguments to remember whether it is 8 bit long, 16 bit, 32 bit or 64 bit; whether it is signed or unsigned; whether the argument is in a binary coded decimal form; and whether the argument designates a pair of registers instead of just one (e.g., in `MR R4,R8` the first argument is a pair of registers R_4 and R_5 which are virtually concatenated and used as one 64-bit number).

All these examples serve to demonstrate problems that are typical in dealing with documentation of large legacy languages: information is partly missing, partly incomplete, partly subtly wrong. In retrospective, HLASM documentation was on the higher side of the quality spectrum: there were no legal reasons to stop us from using it, it was reasonably complete and quite thorough. All the bugs we found in it, were just manifestations of its manual nature and differences between a manually written 2000-pages text and an explicit executable verifiable model.

4 Semantic Steps Modelling

It has been mentioned that as a fallback mechanism to deal with self-modifying code and using code as data, we incorporate an emulator into our runtime. This runtime is used by the compiler, as well as by any compiled program, since it contains support for omnipresent data structures like binary coded decimals. Executing a HLASM instruction through the emulator is the safest way since it works in all circumstances, but it is also the slowest. Thus, if the compiler determines that more optimal ways of executing an instruction, are unsafe, or that there is no alternative (like for `EXECUTE` explained below), it compiles the instruction to a call to the emulator. This emulator has to perform basically five tasks:

- determining which instruction is the next one to be executed;
- parsing its arguments according to the syntactic model;
- fetching the required input data;
- actually executing the steps of core behaviour;
- possibly modifying the condition code;
- determining the program counter for the next instruction.

HLASM, since no low-level language would be complete without an `eval`-like construct [RHBV11], also has an instruction called `EXECUTE` whose actual execution means emulating another instruction from an arbitrary memory location, so one part of the emulator must be able to connect to the others as well.

Determining which instruction to execute next ("instruction decoding" [Kli19b]), in a broad sense, is just parsing [ZB14]. Having the knowledge about the general syntactic structure of each instruction (its length in bytes, positions of instruction-defining opcode bytes and their values) makes this a trivial generative task, with some handling of corner cases—e.g., the EXECUTE instruction mentioned above, does not return to the original call location if the executed instruction is a branching one.

Within the individual instruction execution part, there is also a syntactic part that fetches the required number of bytes from the memory, reconstructs actual values out of them (for instance, a 12 bit value would be constructed by masking and bitshifting one byte and disjuncting it with the other), and preparing them for use (for example, a memory address is composed out of base, index and displacement, which are never used individually). The next part is truly individual: for example, an arithmetic instruction actually adds, subtracts, bit-manipulates or otherwise transforms its input values and stores the result at the expected location. While the first parts (instruction identification and its arguments preparation) were fairly straightforward to infer from our models of syntax, the individual semantic part demanded more work.

In order to model the behaviour of each instruction explicitly, we defined "microcode", a DSL for modelling typical atomic semantic steps such as:

- Fetch a value from a register or from an address in memory
- Extend the sign from an 8-bit or 16-bit value to 32 bits
- Spread a 64-bit value over bytes in memory
- Compute an operation on values with a possible overflow
- Convert a value from a zoned decimal to a packed decimal
- Assign conditions to possible condition codes
- Set the highest bit of a value to 1
- Check a condition and perform one of other actions depending on it
- Perform an action looping through consequent bytes

Some of these steps, such as the conditional step or the loop, are naturally recursive in the sense of containing other semantic steps.

The idea of semantic steps and modelling the behaviour of each of the instructions as a sequence (in fact, a tree) of such steps, works to our satisfaction, with two adjustments. First of them was dealing with condition codes: in HLASM there are many instructions that change a special "magic" two-bit flag called "the condition code" as a result of their execution. For example, any addition instruction on machine words (AR, AGR, AGFR, A, AY, AG, AGF) or halfwords (AH, AHY, AHI, AGHI) assigns a condition code 0 when the result of the addition is zero and no overflow occurs, a condition code 1 when the result is negative without overflowing, a condition code 2 when the result is positive without overflowing, and finally a condition code 3 in the case of an overflow outside the expected bit length. However, actually assigning such condition codes is a costly operation — mostly due to the fact that the .NET Framework does not detect arithmetic overflows naturally, and there exists no perfectly reliable algorithm known for making such predictions, so one needs to perform the operation (addition in this example) on larger data types and then check whether the result would have fit in the smaller type. This by itself would not have been such a problem and would have been seen as necessary evil, but in our observation these condition codes are not always checked right after they are assigned, and it is not uncommon to have several consequent instructions overwriting condition codes without reading them.

```
case 0xBF: // INSERT CHARACTERS UNDER MASK (low) (ICM)
    i3 = mem[instructionAddress+3]; // the instruction is four bytes long
    i2 = mem[instructionAddress+2];
    uint r1 = (uint)(i1 >> 4); // the first argument designates a register
    uint v1 = (uint)GetR(r1); // read the value of the register since it is an input
    uint m2 = (uint)(i1 & 0xF); // the second argument is a mask
    uint b3 = (uint)(i2 >> 4); // base and displacement of the third argument
    uint d3 = (uint)((((uint)i2 & 0xF) << 8) | i3);
    uint a3 = Addr(GetRZ(b3) + d3); // compute the actual memory address based on them
     _bCodeCode = 0x0; // the condition code will be computed directly, not lazily
     var t1 = 0;
     if ((m2 & 8) != 0) // check the leftmost bit of the mask
         v1 = (uint)((v1 & 0x00FFFFFF) | ((uint)mem[a3+(t1++)] << 24));
     if ((m2 & 4) != 0) // check the second leftmost bit of the mask
         v1 = (uint)((v1 & 0xFF00FFFF) | ((uint)mem[a3+(t1++)] << 16));
     if ((m2 & 2) != 0) // check the third leftmost bit of the mask
         v1 = (uint)((v1 & 0xFFFF00FF) | ((uint)mem[a3+(t1++)] << 8));
     if ((m2 & 1) != 0) // check the last (rightmost) bit of the mask
         v1 = (uint)((v1 & 0xFFFFFF00) | mem[a3+(t1++)]);
     SetR(r1, (uint)v1); // assign the fetched value to the register
     if ((m2 != 0) && ((mem[a3] & 0x80) != 0))
         ConditionCode = 1; // CC=1 if leftmost inserted bit is set
     else
     {
         byte t2 = 0;
         for (int it=0;it<t1;it++)
             t2 |= mem[a3+it];
         if ((t2) == 0)
             ConditionCode = 0; // CC=0 if mask is zero or all inserted bits are zero
         else
             ConditionCode = 2; // CC=2 otherwise
     }
     return nextProgramCounter; // continue execution as normal
```

Figure 4 – ICM: Insert Character under Mask (emulator code in C♯)

Hence, it made sense (and a measurable impact on performance) to explicitly model the condition code computation instead and delay the actual computation of them until (if ever) it is actually required. Once the condition code has been accessed, it is computed and cached for possible repetitive uses later. Evolving our system to switch to this lazy condition code evaluation was done by tweaking the model transformation only, without any change to the models themselves.

Having witnessed the impact of this optimisation, we engaged in a separate performance analysis project [Mje17] to investigate other possible bottlenecks in the system and ways to overcome them. The HLASM compiler has never been performance-focused, but it could not afford to be entirely performance-ignorant, for both technical and marketing reasons. The findings included a list of hotspots — instructions that were both used often in the sample code of our customer, and took significant time to execute. For some of them we have proposed alternative implementations, coded directly in IL (the bytecode-level language of the .NET Framework and .NET Core). One of the biggest performance gains was the ICM instruction we have seen above. Normally, it takes a register, a mask and a memory address and fetches consecutive bytes from the memory location into the register according to the mask — for example, if the mask is 1001 binary, then there are two bytes fetched, one ends up as the highest byte of the target register, the other one as the lowest byte, and the two bytes in between remain unchanged. As it turns out, it is a common idiom among HLASM programmers to have ICM with a mask of 1111 which just straightforwardly fetches four consecutive bytes from a memory location into four consecutive bytes of a target register. The reasons to use ICM instead of the L (“load”) instruction that performs the same procedure is that `L` leaves the condition code unchanged while `ICM` assigns a value to it, allowing to fetch four bytes with one instruction and, for instance, check if they yielded zero with the next instruction.

The general implementation of `ICM` easily fills up the entire screen (see Figure 4: all comments except the one on the first line are added manually for readability, the rest of the program is fully generated), but its full-mask-simplified form is extremely simpler and consists of two IL-level instructions for the actual semantics and barely a dozen more for computing the condition code. This is how we arrived at the idea of *conditional inlining*: if the code is safe (not self-modifying) and if the inlining conditions are met (in this case, if the mask is `1111` binary), then the instruction is compiled to its shorter optimised form; otherwise the emulator is called with the right arguments. This technique has been clunkily named previously as “compilepretation” [Zay17b] since it combines aspects of traditional compilation (model-to-code) and interpretation (operational semantics). Introducing concurrency into the mix has bears another clunky name of “interpretisation” [Kli19a].

To verify the equivalence of the general operational semantics of the emulator and the partial optimised inlining semantics, we used an old program transformation technique called *supercompilation* (supervision + compilation), based on works of Lombardi [Lom67], Futamura [Fut71], Ershov [Ers77] and Turchin [Tur80]. It was designed specifically to transform executable models by observing/supervising their behaviour and compiling them to self-sufficient models that achieve the same effect while being smaller thanks to utilisation of additional (meta)data. For the example from Figure 4, if the mask is known to be `1111` binary, then all the `if`s checking for individual bits on lines 12, 14, 16 and 18 succeed, so we reach the same effect by immediately executing the positive branches of each conditional statement. Furthermore, the counter `t1` reliably increases by `1` so it does not need to be kept since all the instances of reading it become constants, and then it does not even have to be defined on line 11. Then, since all bytes of `v1` will be overwritten, there is no need to mask-carve the right ones with bit conjunctions and bit disjunctions. In fact, there is no need to read its original value into `v1` on line 5 at all, it will be overwritten in any case. In a similar series of near-trivial simplification steps we can prove that constructing `t2` in lines 25–27 and checking it for being equal to `0` is the same as checking `v1` directly for being equal to `0`. Theoretically we could have implemented such supercompilation steps to *infer* the inlining code based on the model of the emulator semantics and the known condition, but for technical reasons we decided to write the inlining code manually and then use supercompilation for *verification* of its correctness.

The microcode language had to be evolved in a significant way twice. The first change was to accommodate the inlining abstractions due to the differences between domains. For instance, assignment statements in the emulator assume some high level language that they will have to generate at some point down the toolchain (such as C♯) and thus rely on things like variables and even their automatic type inference, while the inlining semantic steps are meant to produce very low level bytecode constructs. Thus, they have to integrate well with their context of gaining input values and producing output values. For example, `AssignProgramCounter` is a microcode command that makes sense in both contexts (emulating and inlining), so it requires no special attention. However, `DeclareValue`, when used in an emulator, allows an optional initialiser that can be used to assign the value right after declaring it (since it is extremely easy to do on the level of C♯, and quite useful at times), but there are too many complications for inlining, so we disallow initialisers there. Similarly, the

`Argument` expression is an approximate equivalent of the `Variable` expression and its variants (`Value`, `Address`, etc), but requires an explicit type each time it is used, due to the low-levelness of IL.

The second change was related to the current activities around the HLASM compiler where we aim to use LLVM as a backend instead of the .NET Framework, so it covered a polishing pass over the microcode commands to remove the C♯ bias and simplify possible generation of C code such that our customers can execute their HLASM programs on arbitrary Linux machines without relying on .NET Core. These adjustments were relatively minor and concerned details like explicit and implicit type conversion rules in C and C♯ when dealing with signed and unsigned integers.

In section 2 and Figure 3 we mentioned that some models are kept until compile time or even runtime. To be more precise, in this case there is a model of inlining semantics used at compile time, but it conforms to a different metamodel, better suitable to the architecture of our compiler. The transformation from models conforming to the microcode metamodel to models conforming to this compile-time-specific metamodel, is not far beyond trivial in complexity, and does not challenge the state of the art in model transformation.

5 Related Work

Extracting fully structured curated data with heuristics from a semi-structured source, as we used in section 3, is related to many things: we have already mentioned grammar extraction based on textual cues [LZ11] and on known properties of anchor symbols [Zay12]. The bibliography of [Zay14] provided us with a comprehensive view on the topic of using all sorts of tolerant, permissive and error-correcting parsing. The research area of mining unstructured data has been active for at least two decades, and produced quite a number of various techniques [Fel99], mostly based on heuristics and/or data mining.

Optimising a compiler by specifically targeting *code idioms* [AS14] is not a new idea and has been successfully employed for almost three decades in FORTRAN compilers [HSVF08, PP91, PE95] and later even on the mainframe [KKM$^+$06]. In a contemporaneous project we are trying to find ways to identify such idioms automatically with graph mining [PNM$^+$19, FZM$^+$19, PBM$^+$19, NPF$^+$19] since their manual construction for each language is rather labour-intensive.

One of the substantial recent contributions to research on execution semantics of software languages was done by Tikhonova [Tik19]. In her terms, our microcode (section 4) defines a *semantic domain* and what we call models of instruction semantics together form a *semantic mapping* as *specification templates* (possibly with less sufficient formal rigour on our side). Conceptually Tikhonova's work on Constelle rhymes with our experience and is well aligned with it; however, technically even if Constelle was released before the start of our project, it is unlikely that we would have chosen to use it directly for the fear of relying on third party technology with unknown and unpredictable lifespan and maintainability status. On the other hand, the component-based executable semantics of funcons [vBMS19, M$^+$19] served as a major inspiration in this project to design the microcode, toned down by the fact that we needed it for one very specific language in the scope of one project, and Mosses initially planned funcons to serve as a playground for creating all kinds of different DSLs. Besides that, the approach obviously aimed at experimental forward engineering of small software languages, was successfully applicable to this project of reverse engineering semantics of a relatively large legacy software language.

Similarly, our own model transformation framework was developed in-house and was never meant to cover the entire domain of model transformation. There are much better general purpose academic frameworks like MOMENT that formalised a model transformation language in a term rewriting framework Maude [BCR06], which has also seen applications in transforming legacy software [BCR05].

One of the unmissable references in the field of assembler modelling is the work of Kennedy et al [KBJD13] who managed to model the Intel x86 assembler with type classes and dependent types in Coq. They never reached complete coverage of the language, but for the covered subset they provided auto-proven theorems on correctness (relating in-memory code to a verifiable formula). There are similar projects such as one by Schmaltz and Shadrin modelling joint semantics of C with macro assembler also for the purpose of verification [SS12] — notably the authors recognised later that the definition was leaky with respect to some stack manipulations and needed to cover the basic assembler as well [PSS12]. Even simpler methods of semantic modelling seeing programs as collections of execution paths with weak preconditions [WF03], are inherently incapable of modelling self-modifying code similar to well-used HLASM features omnipresent in industrial codebases. At the current point it does not seem possible for us to make a step from having constructed models of semantics for individual instructions, to inferring a full system specification suitable for verification and proving useful properties.

On a more technical side, Klimiankou recently published an interesting story centred primarily on parsing of instructions for IA-32 (which is the Intel assembler as opposed to our IBM assembler, but the two are very much alike) [Kli19b]. In our work his "instruction decoding" corresponds to the emulator figuring out at runtime which instruction to execute next and how to turn its bits into meaningful entities corresponding to its arguments in terms of which its core semantics is expressed. Klimiankou managed to build the fastest decoder for IA-32 commands [Kli19b] and was able to leverage it to migrate from switch-based dispatch (that we also use) to concurrent threaded code [Kli19a].

6 Conclusion and Lessons Learnt

In this document we have reported on a project that focused on extracting and refining models of syntax and semantics of instructions of High Level Assembler (HLASM) [IBM13, IBM17] with the final goal of building a compiler for that language [BJZ16, Rai16]. The project was seen as successful from our side, since it met the expectations of customers and was completed within a very limited time frame by a small team of people, even though the language consisted of hundreds of instructions and macros. `git` statistics show a total of 817 commits from May 2015 till May 2020 concerning the folders with the HLASM compiler after filtering out non-human committers like the nightbuild system: 423 are made by this paper's author, 244 by Ynès Jaradin, the lead architect of the project and a co-author of the original report [BJZ16], and 150 commits made by 13 other senior software developers occasionally contributing to the project.

Knowledge extraction was done ad hoc, yet according to the state of the art methodology [LZ11, Zay12, Zay14, Bav16, HKLM16]. Mining and extracting semi-structured data is an active field of research [Bav16, HKLM16], but there was no ready to use tool for this particular text-to-model transformation, and the cost of developing it was not that high, given prior experience and expertise of our developers. Model fixing was a labour intensive process due to its manual nature, but cross-checking different parts of the models with one another, as well as models extracted from different sources even within the same original document, was useful. Conceptually this was a straightforward application of abstract model repair [CBSK12].

Enriching the models with new information was, again, done with bespoke technology, and then redone to refactor away idiosyncrasies. The technology was not the bottleneck, but the metamodel was — in the sense that we needed to make sure the models contained all the information that can be properly expressed and that will be useful later at the code generation stage. Given the context of the project and the policies within our company, it seems unlikely that we would have used any available tools if they were adding technical dependencies of their own. However, it was crucial to employ as much automation as possible, to avoid introducing or propagating hard-to-catch bit-level errors. Existing model transformation frameworks were not used, and new ones were not developed—seemed like overengineering since we did not need any intricate expressiveness.

Models of the syntax of the instructions were more straightforward than the models of the semantics of them, and they were fairly structured already in the original documentation, so our part was limited to extracting them in a form suitable for automated processing, fixing inconsistencies and imperfections (possibly introduced by manual processing and typesetting instead of relying on generative techniques), augmenting the models with additional information that was not present in the source explicitly (even though it could have been, but it probably just never occurred to the documentation writers to summarise it), and generating the desired artefacts. In MDE terms, this was metamodelling in a low complexity domain. To model semantics, we had to read through thousands of pages of descriptions given in natural language, and encode it in a specially designed DSL (microcode, see section 4). The language design was challenging as it always is [Zay17a].

After further analysis of the performance of the compiled code of the HLASM emulator [Mje17] and the structure of our customers' source code, we came to the conclusion that our existing models of instruction semantics were insufficient. We had to invest significantly into enhancing them to cover not only the behaviour of the emulator, but also rules for conditional inlining that can be used if the code is safe (i.e., is not modifying itself). Later, they were enhanced to be rid of C♯ idiosyncrasies in order for us to be able to use it with an alternative backend such as LLVM. This step did not use any model-level profiling, so it needed manual lifting of the numbers crunched at the code level, to the inlining semantic models level.

The documentation was round-tripped: our tools try to produce the documentation inferred from our models, in a form that is as close as possible to the original, for easier comparison and visual verification. The main obvious change is that the original natural language description of the semantics of each instruction, is replaced in our case with microcode or semi-structured prose generated from it. Again, we followed the state of the art in what is desirable and advisable for generated executable language documentation, and gained expected results [ZL11].

For the framework to implement our model-to-text transformations we used T4, a Microsoft template language [MSD]. Several templates were developed: one for the documentation, one for several versions of the emulator, etc. The chosen technology turned out to be satisfactory, but did not contribute to the project in any overly significant way. The main reason for choosing it was its integration into the IDE that we were already using (Visual Studio .NET).

Some of the models had to be kept at hand beside the compiler (and be delivered as a part of the product) for the sake of the possibility to tailor the project to each specific customer by supporting older versions of the HLASM language, different sets of macros, etc (cf. Figure 3). The architecture of the compiler had to take this modularity into account on many levels. We are unaware of other industrial or academic compilers that go this far towards full configurability, and can allow the end user of the shipped compiler to dramatically alter the language definition of the language being compiled. At the frontend side (websites and mobile apps) comparable approaches are called "low-code" [RRM+14].

Given the context of the problem of implementing a massive low-level language from scratch, for legal reasons without looking at the baseline IBM assembler nor at its existing open-source partial replacements, in a team of very limited size within a rigid timeframe, this project was subjectively for us a very successful application of software language engineering, software modelling and model transformation. Determining the right level of abstraction and identifying the right elements to put in the metamodel, in order to automatically refine the models of both syntax and semantics of each of the instructions in the set, and produce final components of the compiler in a reliable and testable [GZ19] way, was a winning strategy that allowed us to avoid burnout, produce a viable product and deploy it to our customers' satisfaction.

References

[AS14] Miltiadis Allamanis and Charles A. Sutton. Mining Idioms from Source Code. In *Proceedings of the 22nd Symposium on the Foundations of Software Engineering (FSE)*, pages 472–483. ACM, 2014. doi:10.1145/2635868.2635901.

[B+] Ira Baxter et al. Mainframe Assembler Migration. Semantic Designs, http://www.semdesigns.com/Products/Services/MainframeAssemblerMigration.html.

[Bav16] Gabriele Bavota. Mining Unstructured Data in Software Repositories: Current and Future Trends. In *Leaders of Tomorrow Symposium: Future of Software Engineering (FOSE at SANER)*, pages 1–12. IEEE CS, 2016. doi:10.1109/SANER.2016.47.

[BCR05] Artur Boronat, José Ángel Carsí, and Isidro Ramos. Automatic Reengineering in MDA Using Rewriting Logic as Transformation Engine. In *Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR)*, pages 228–231. IEEE CS, 2005. doi:10.1109/CSMR.2005.14.

[BCR06] Artur Boronat, José Ángel Carsí, and Isidro Ramos. Algebraic Specification of a Model Transformation Engine. In *Proceedings of the Ninth International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 3922 of *LNCS*, pages 262–277. Springer, 2006. doi:10.1007/11693017_20.

[BJZ16] Volodymyr Blagodarov, Yves Jaradin, and Vadim Zaytsev. Tool Demo: Raincode Assembler Compiler. In Tijs van der Storm, Emilie Balland, and Dániel Varró, editors, *Proceedings of the Ninth International Con-*

ference on Software Language Engineering (SLE), pages 221–225, 2016. `doi:10.1145/2997364.2997387`.

[Bla95] Darius Blasband. The YAFL Programming Language. *Journal of Object-Oriented Programming*, 8(7):42–49, 1995.

[Bla01] Darius Blasband. Parsing in a Hostile World. In Elizabeth Burd, Peter Aiken, and Rainer Koschke, editors, *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE)*, pages 291–300. IEEE CS, 2001. `doi:10.1109/WCRE.2001.957834`.

[BP96] Michael R. Blaha and William J. Premerlani. A Catalog of Object Model Transformations. In *Proceedings of the Third Working Conference on Reverse Engineering*, pages 87–97. IEEE CS, 1996. `doi:10.1109/WCRE.1996.558881`.

[CBSK12] George Chatzieleftheriou, Borzoo Bonakdarpour, Scott A. Smolka, and Panagiotis Katsaros. Abstract Model Repair. In Alwyn Goodloe and Suzette Person, editors, *Proceedings of the Fourth International Symposium on NASA Formal Methods (NFM)*, volume 7226 of *LNCS*, pages 341–355. Springer, 2012. `doi:10.1007/978-3-642-28891-3_32`.

[Ers77] Andrey P. Ershov. On the Essence of Translation. In Erich J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–418. North-Holland, 1977.

[Fea04] Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall, October 2004.

[Fel99] Ronen Feldman. Mining Unstructured Data. In *Tutorial Notes of the Fifth International Conference on Knowledge Discovery and Data Mining*, KDD, pages 182–236. ACM, 1999. `doi:10.1145/312179.312192`.

[Fut71] Yoshihiko Futamura. Partial Evaluation of Computation Process — An Approach to Compiler-Compiler. *Systems, Computers, Control*, 2(5):45–50, 1971.

[FZM$^+$19] Johan Fabry, Vadim Zaytsev, Kim Mens, Siegfried Nijssen, Hoang Son Pham, Coen De Roover, Dario Di Nucci, and Tim Molderez. A Language-Parametric Toolchain for Mining Idiomatic Code Patterns. In *Programming 2019 Demos Track*. 2019. URL: `https://tinyurl.com/yx8gmjhj`.

[GZ19] Aynel Gül and Vadim Zaytsev. Mutative Fuzzing for an Assembler Compiler. In Dario Di Nucci and Coen De Roover, editors, *Proceedings of the 18th Belgium-Netherlands Software Evolution Workshop (BENEVOL)*, volume 2605 of *CEUR Workshop Proceedings*, pages 18–24. CEUR-WS.org, 2019. URL: `http://ceur-ws.org/Vol-2605/18.pdf`.

[HKLM16] Sonia Haiduc, Takashi Kobayashi, Michele Lanza, and Andrian Marcus. Mining & Modeling Unstructured Data in Software — Challenges for the Future (NII Shonan Meeting 2016-3). *NII Shonan Meeting Reports*, 2016. `http://shonan.nii.ac.jp/shonan/report/no-2016-3/`.

[HSVF08] Jiahua He, Allan Snavely, Rob F. Van der Wijngaart, and Michael A. Frumkin. Code Coverage, Performance Approximation and Automatic Recognition of Idioms in Scientific Applications. In *Proceedings of the 17th International Symposium on High-Performance Parallel and*

Distributed Computing (HPDC), pages 223–224. ACM, 2008. doi:10.1145/1383422.1383456.

[IBM04] *SA22-7832-03: z/Architecture Principles of Operation.* IBM, fourth edition, May 2004.

[IBM13] *GC26-4943-06: High Level Assembler for z/OS & z/VM & z/VSE Version 1 Release 6 General Information.* IBM, 2013.

[IBM17] *SA22-7832-11: z/Architecture Principles of Operation.* IBM, twelfth edition, September 2017.

[JSW99] Adrian Johnstone, Elizabeth Scott, and Tim Womack. Experience Paper: Reverse Compilation of Digital Signal Processor Assembler Source to ANSI-C. In *Proceedings of the 15th International Conference on Software Maintenance*, pages 316–325. IEEE CS, 1999.

[KBJD13] Andrew Kennedy, Nick Benton, Jonas Braband Jensen, and Pierre-Évariste Dagand. Coq: The World's Best Macro Assembler? In *Proceedings of the 15th International Conference on Principles and Practice of Declarative Programming*, pages 13–24. ACM, 2013. doi:10.1145/2505879.2505897.

[KKM+06] Motohiro Kawahito, Hideaki Komatsu, Takao Moriyama, Hiroshi Inoue, and Toshio Nakatani. A New Idiom Recognition Framework for Exploiting Hardware-Assist Instructions. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 382–393. ACM, 2006. doi:10.1145/1168857.1168905.

[Kli19a] Yauhen Klimiankou. Interpretizer: A Compiler-Independent Conversion of Switch-Based Dispatch into Threaded Code. In Manuel Mazzara, Jean-Michel Bruel, Bertrand Meyer, and Alexander K. Petrenko, editors, *Proceedings of the 51st International Conference on Software Technology: Methods and Tools (TOOLS)*, volume 11771 of *LNCS*, pages 59–72. Springer, 2019. doi:10.1007/978-3-030-29852-4_4.

[Kli19b] Yauhen Klimiankou. Rapid Instruction Decoding for IA-32. In Nicolaj Bjørner, Irina Virbitskaite, and Andrei Voronkov, editors, *Proceedings of the 12th A. P. Ershov Informatics Conference (PSI)*, pages 141–150. IPC NSU, 2019. doi:10.1007/978-3-030-37487-7_1.

[LB96] Tom Lake and Tim Blanchard. Reverse Engineering of Assembler Programs: A Model-Based Approach and its Logical Basis. In *Proceedings of the Third Working Conference on Reverse Engineering (WCRE)*, pages 67–75. IEEE CS, 1996. doi:10.1109/WCRE.1996.558872.

[Lom67] Lionello A. Lombardi. Incremental Computation. *Advances in Computers*, 8, 1967. doi:10.1016/S0065-2458(08)60698-1.

[LZ11] Ralf Lämmel and Vadim Zaytsev. Recovering Grammar Relationships for the Java Language Specification. *Software Quality Journal (SQJ); Section on Source Code Analysis and Manipulation*, 19(2):333–378, March 2011. doi:10.1007/s11219-010-9116-5.

[M+19] Peter D. Mosses et al. CBS-beta — PLanCompS. https://plancomps.github.io/CBS-beta/, 2019.

[MBB06] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proceedings of the International Conference on Management of Data*, SIGMOD, pages 706–706. ACM, 2006. doi:10.1145/1142473.1142552.

[MHS05] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, 2005. doi:10.1145/1118890.1118892.

[Mic] MicroAPL. Relogix Sample Translation (IBM Mainframe). http://microapl.com/asm2c/sampleibm.html.

[Mic06] Microsoft. Windows Presentation Foundation. https://docs.microsoft.com/en-us/dotnet/framework/wpf/, 2006.

[Mic08] Microsoft. Extensible Application Markup Language (XAML) overview in WPF. https://docs.microsoft.com/en-us/dotnet/desktop-wpf/fundamentals/xaml, 2008.

[Mit12] Robert L. Mitchell. The Cobol Brain Drain. *Computerworld*, May 2012. URL: https://www.computerworld.com/article/2504568/the-cobol-brain-drain.html.

[Mje17] Roar Mjelde. *HLASM: Optimization of a z/Architecture Emulator*. INF319, Universitas Bergentis, 2017.

[MSD] MSDN. Code Generation and T4 Text Templates. https://msdn.microsoft.com/en-gb/library/bb126445.aspx.

[NPC+19] Cyrus Najmabadi, Jared Parsons, Heejae Chang, Tomáš Matoušek, Sam Harwell, Manish Vasani, Jason Malinowski, et al. The .NET Compiler Platform (“Roslyn”). https://github.com/dotnet/roslyn, 2019.

[NPF+19] Dario Di Nucci, Hoang-Son Pham, Johan Fabry, Coen De Roover, Kim Mens, Tim Molderez, Siegfried Nijssen, and Vadim Zaytsev. A Language-Parametric Modular Framework for Mining Idiomatic Code Patterns. In Anne Etien, editor, *Proceedings of the 12th Seminar on Advanced Techniques & Tools for Software Evolution (SATTOSE)*, volume 2510 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2019. URL: http://ceur-ws.org/Vol-2510/sattose2019_paper_3.pdf.

[PBM+19] Yunior Pacheco, Jonas De Bleser, Tim Molderez, Dario Di Nucci, Wolfgang De Meuter, and Coen De Roover. Mining Scala Framework Extensions for Recommendation Patterns. In Xinyu Wang, David Lo, and Emad Shihab, editors, *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 514–523. IEEE, 2019. doi:10.1109/SANER.2019.8668019.

[PE95] Bill Pottenger and Rudolf Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. In *Proceedings of the Ninth International Conference on Supercomputing (ICS)*, pages 444–448. ACM, 1995. doi:10.1145/224538.224655.

[PMBM98] Wei-Jin Park, Sang-Yoon Min, Doo-Hwan Bae, and Pyeong-Soo Mah. Object-oriented model refinement technique in software reengineering. In *Proceedings of the 22nd International Computer Software and Applications Conference (COMPSAC)*, pages 32–38. IEEE CS, 1998. doi:10.1109/CMPSAC.1998.716633.

[PNM+19] Hoang Son Pham, Siegfried Nijssen, Kim Mens, Dario Di Nucci, Tim Molderez, Coen De Roover, Johan Fabry, and Vadim Zaytsev. Mining Patterns in Source Code using Tree Mining Algorithms. In Petra Kralj Novak, Tomislav Šmuc, and Sašo Džeroski, editors, *Proceedings of the 22nd International Conference on Discovery Science (DS)*. Springer, 2019. doi:10.1007/978-3-030-33778-0_35.

[PP91] Shlomit S. Pinter and Ron Y. Pinter. Program Optimization and Parallelization Using Idioms. In David S. Wise, editor, *Conference Record of the 18th Annual Symposium on Principles of Programming Languages (POPL)*, pages 79–92. ACM Press, 1991. doi:10.1145/99583.99597.

[Pre16] Presse Box. Microsoft zeichnet Raincode als "Top Performer for Mainframe Migration" aus. https://www.pressebox.de/inaktiv/raincode-gmbh/Microsoft-zeichnet-Raincode-als-Top-Performer-for-Mainframe-Migration-aus/boxid/807202, July 2016.

[PSS12] Wolfgang J. Paul, Sabine Schmaltz, and Andrey Shadrin. Completing the Automated Verification of a Small Hypervisor — Assembler Code Verification. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods (SEFM)*, volume 7504 of *LNCS*, pages 188–202. Springer, 2012. doi:10.1007/978-3-642-33826-7_13.

[Rai16] Raincode. The Raincode ASM370 compiler for .NET and .NET Core. https://www.raincode.com/technical-landscape/asm370/, 2016.

[Rai19] Raincode Labs. Bankia Chooses Raincode Labs for PACBASE migration. https://www.raincodelabs.com/blog/bankia-chooses-raincode-labs-for-pacbase-migration/, 2019.

[RHBV11] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The Eval That Men Do — A Large-Scale Study of the Use of Eval in JavaScript Applications. In Mira Mezini, editor, *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP)*, volume 6813 of *LNCS*, pages 52–78. Springer, 2011. doi:10.1007/978-3-642-22655-7_4.

[RRM+14] Clay Richardson, John R. Rymer, Christopher Mines, Alex Cullen, and Dominique Whittaker. New Development Platforms Emerge For Customer-Facing Applications. https://www.forrester.com/go?objectid=RES113411, June 2014.

[Sou] Soukhman. Assembler to C translator. http://www.soukhman.com/Asm2CobolCJava/Asm2CobolCJava%20SG.html, page long defunct, but described in detail at http://www.semdesigns.com/Products/Services/Soukhman.html.

[SS12] Sabine Schmaltz and Andrey Shadrin. Integrated Semantics of Intermediate-Language C and Macro-Assembler for Pervasive Formal Verification of Operating Systems and Hypervisors from VerisoftXT. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments (VSTTE)*, pages 18–33. Springer, 2012. doi:10.1007/978-3-642-27705-4_3.

[Tik19] Ulyana Tikhonova. Reusable Specification Templates for Defining Dynamic Semantics of DSLs. *Software and System Modeling*, 18(1):691–720, 2019. doi:10.1007/s10270-017-0590-0.

[Tur80] Valentin F. Turchin. The Use of Metasystem Transition in Theorem Proving and Program Optimization. In J. W. de Bakker and Jan van Leeuwen, editors, *Proceedings of the Seventh Colloquium on Automata, Languages and Programming (ICALP)*, volume 85 of *LNCS*, pages 645–657. Springer, 1980. doi:10.1007/3-540-10003-2_105.

[vBMS19] L. Thomas van Binsbergen, Peter D. Mosses, and Neil Sculthorpe. Executable Component-based Semantics. *Journal of Logic and Algebraic Programming*, 103:184–212, 2019. doi:10.1016/j.jlamp.2018.12.004.

[War99] Martin P. Ward. Assembler to C Migration Using the FermaT Transformation System. In *Proceedings of the 15th International Conference on Software Maintenance (ICSM)*, pages 67–76. IEEE CS, 1999. doi:10.1109/ICSM.1999.792571.

[War00] Martin P. Ward. Reverse Engineering from Assembler to Formal Specifications via Program Transformations. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE)*, page 11. IEEE CS, 2000. doi:10.1109/WCRE.2000.891448.

[War01] Martin P. Ward. The FermaT Assembler Re-engineering Workbench. In *Proceedings of the 17th International Conference on Software Maintenance (ICSM)*, pages 659–662. IEEE CS, 2001. doi:10.1109/ICSM.2001.972783.

[War13] Martin P. Ward. Assembler Restructuring in FermaT. In *Proceedings of the 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 147–156. IEEE, 2013. doi:10.1109/SCAM.2013.6648196.

[WF03] Geoffrey Watson and Colin J. Fidge. A Partial-Correctness Semantics for Modelling Assembler Programs. In *Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM)*, pages 82–90. IEEE CS, 2003. doi:10.1109/SEFM.2003.1236210.

[WPP+19] Leszek Włodarski, Boris Pereira, Ivan Povazan, Johan Fabry, and Vadim Zaytsev. Quality First! A Large Scale Modernisation Report. In Xinyu Wang, Zhenyu Chen, and Jinjun Hu, editors, *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering — Industry Track (SANER IT)*, pages 569–573, 2019. doi:10.1109/SANER.2019.8668006.

[WZH04] Martin P. Ward, Hussein Zedan, and T. Hardcastle. Legacy Assembler Reengineering and Migration. In *Proceedings of the 20th International Conference on Software Maintenance*, pages 157–166. IEEE CS, 2004. doi:10.1109/ICSM.2004.1357800.

[Zay12] Vadim Zaytsev. Notation-Parametric Grammar Recovery. In Anthony Sloane and Suzana Andova, editors, *Post-proceedings of the 12th International Workshop on Language Descriptions, Tools, and Applications (LDTA)*. ACM DL, June 2012. doi:10.1145/2427048.2427057.

[Zay14] Vadim Zaytsev. Formal Foundations for Semi-parsing. In Serge Demeyer, Dave Binkley, and Filippo Ricca, editors, *Proceedings of the Software Evolution Week (IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering), Early Research Achievements*

Track (CSMR-WCRE 2014 ERA), pages 313–317. IEEE, February 2014. doi:10.1109/CSMR-WCRE.2014.6747184.

[Zay17a] Vadim Zaytsev. Language Design with Intent. In Don Batory, Jeff Gray, and Vinay Kulkarni, editors, *Proceedings of the 20th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 45–52. IEEE, 2017. doi:10.1109/MODELS.2017.16.

[Zay17b] Vadim Zaytsev. On the Need of Compilepretation for Legacy Languages. In Laurence Tratt, Adam Welc, and Stefan Marr, editors, *Workshop on Modern Language Runtimes, Ecosystems, and Virtual Machines (MoreVMs 2017)*, 2017.

[Zay17c] Vadim Zaytsev. Open Challenges in Incremental Coverage of Legacy Software Languages. In Luke Church, Richard P. Gabriel, Robert Hirschfeld, and Hidehiko Masuhara, editors, *Post-proceedings of the Third Edition of the Programming Experience Workshop (PX/17.2)*, pages 1–6, 2017. doi:10.1145/3167105.

[ZB14] Vadim Zaytsev and Anya Helene Bagge. Parsing in a Broad Sense. In Jürgen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Proceedings of the 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2014)*, volume 8767 of *LNCS*, pages 50–67. Springer, October 2014. doi:10.1007/978-3-319-11653-2_4.

[ZL11] Vadim Zaytsev and Ralf Lämmel. A Unified Format for Language Documents. In Brian A. Malloy, Steffen Staab, and Mark G. J. van den Brand, editors, *Post-proceedings of the Third International Conference on Software Language Engineering (SLE)*, volume 6563 of *LNCS*, pages 206–225. Springer, 2011. doi:10.1007/978-3-642-19440-5_13.

About the author

Vadim Zaytsev is the Chief Science Officer of Raincode Labs, the largest independent compiler company in the world. His interests currently revolve around applying model-based, model-driven and low-code automation techniques in development of compilers, program transformations and other tools that help renovating legacy software systems.
Contact email: vadim@grammarware.net.
Websites to visit: http://grammarware.net or http://grammarware.github.io.