Domain-specific programming assistance in an embedded DSL for generating processor emulators

Katsumi Okuda
okuda@csg.ci.i.u-tokyo.ac.jp
The University of Tokyo
Okuda.Katsumi@eb.MitsubishiElectric.co.jp
Mitsubishi Electric Corporation

Shigeru Chiba
chiba@acm.org
The University of Tokyo

ABSTRACT

This paper presents a design approach for developing an embedded domain-specific language (DSL) with domain-specific programming assistance. To demonstrate the proposed approach, we describe the design of our processor description language called MELTRANS, which is an embedded DSL hosted by Java. MELTRANS is used to generate a fast processor emulator with dynamic binary translation. Although such embedded DSLs have provided only poor domain-specific assistance for programming, MELTRANS improves this poorness. Our idea is to let the integrated development environment (IDE) of the host language provide better programming assistance using domain-specific knowledge. To this end, we decompose a program in MELTRANS into several Java classes corresponding to a different concern and set a rule for the description order. The language runtime takes each class and generates not only the code for processor emulators but also superclasses for classes written later. The user writes each class describing a concern as the subclasses of a generated class. The generated classes enable the user to benefit from programming assistance by Java IDEs in a more domain-specific style. While MELTRANS is hosted in Java, the generated emulators are written in C++. To validate our design, we implement several emulators in MELTRANS and perform experiments with them. The results show that our domain-specific programming assistance can effectively reduce the amount of code that needs to be written by the user, and the generated emulators achieve over 1,000 MIPS.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Specification languages**; *Simulator / interpreter*; Software design engineering;

KEYWORDS

Domain specific languages, Processor description languages, Processor emulators, Instruction set simulators, Dynamic binary translation

SAC '21, March 22–26, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8104-8/21/03.
https://doi.org/10.1145/3412841.3442000

ACM Reference Format:
Katsumi Okuda and Shigeru Chiba. 2021. Domain-specific programming assistance in an embedded DSL for generating processor emulators. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3412841.3442000

1 INTRODUCTION

Domain-specific languages (DSLs) are widely used in many domains to develop software efficiently for a particular use. Because DSL users can concentrate on the domain-specific problem, the development in DSLs is more efficient than in general-purpose programming languages. One successful application of DLSs is processor specification to generate a processor emulator or to design a processor. The DSLs for this purpose are called processor description languages (PDLs) or architecture description languages [16]. Several PDLs, including nML [10], ISDL [9], LISA [21], EXPRESSION [23], Harmless [12], and HPADL [13], have been developed and used in academia and industry.

Some PDLs are implemented as embedded DSLs, which are libraries or frameworks in their host languages. Examples include Pydgin [15] hosted in Python and ArchC [1] hosted in SystemC [20]. An advantage of embedded DSLs is that they can borrow the host language's tools, including their integrated development environments (IDEs), thereby reducing the development cost of DSLs. However, the programming assistance by IDEs is not satisfactory; more domain-specific assistance for convenience and correctness should be provided.

In this paper, we present a design approach to the embedded DSLs enabling domain-specific programming assistance by their host's IDEs. The DSLs are carefully designed to let the IDEs provide better auto-completion and error detection by domain-specific knowledge. To this end, the DSL compiler/runtime generates support programs while the users write a DSL program. The DSL program is split into multiple components for different concerns; the DSL compiler reads a component written earlier and generates a support program for later components. The domain-specific knowledge is encoded into the support program, and the IDE refers to this support program for providing domain-specific assistance when other components are written. An example of the support program is the superclass of the class written by the DSL user.

The contributions of this work are three-fold:

- We reveal that domain-specific programming assistance by IDEs is poor for embedded DSLs and present our approach to address this problem. Our approach exploits program

generation such that programming assistance for the host language can be regarded as domain-specific.

- We conduct a case study by developing a practical embedded DSL called MELTRANS and demonstrate the domain-specific assistance that is available when the user is writing a program in MELTRANS.
- We experimentally confirm that the domain-specific assistance provided by MELTRANS can effectively reduce the amount of code to be written by the user, the generated emulator can emulate a processor in over 1,000 MIPS, and MELTRANS is general enough to generate an emulator for several commercial instruction sets (ARM, MIPS64, SH, RH850, RISC-V, and RX).

In the rest of this paper, we first reveal that domain-specific programming assistance by IDEs is inadequate for embedded DSL by using a processor description language as an example. We then propose a design approach for developing an embedded DSL with domain-specific programming assistance. We also present our experiments and related work. Finally, we conclude this paper.

2 MOTIVATING PROBLEM

The growing adoption of cross-platform virtualization and the rise in instruction set architecture (ISA) diversity are resulting in a need for an efficient method to develop a fast processor emulator. One promising approach is to generate processor emulators using the program written in PDLs. In general, practical processor emulators need to implement dynamic binary translation (DBT) to execute the guest program rapidly. The implementation of emulators with DBT is complicated without a PDL. The emulator with DBT translates guest instructions into host native instructions at runtime. Without a PDL, the developer of emulators needs to describe how the emulator translates guest instructions into host instructions. The developer is required to have a deep understanding of not only guest instructions but also host instructions. When we use a PDL, the PDL compiler can generate code for the translation on behalf of the developer.

A PDL can be implemented as an embedded DSL. An advantage of this approach is that users can exploit an existing tool-chain for the host language of the embedded DSL. However, a drawback of this approach is that it provides poor domain-specific programming assistance.

Suppose that we describe the MUL instruction of the ARM processor in such an embedded DSL. The MUL instruction multiplies values in the source registers and stores the result in the destination register. Listing 1 is the pseudo-code expressing the semantics of the MUL instruction, which is quoted from the ARM reference manual [11]. In a Python-based embedded DSL, the Pydgin PDL [15], the same semantics is implemented by the execute_mul function shown in Listing 2. This function appears to be very similar to the pseudo-code in Listing 1. Thus, the user can describe the execute_mul function by mostly copying the pseudo-code in the reference manual. The user does not have to specify how to translate the MUL instruction into the host instructions when implementing the processor emulator in Pydgin.

Expressions inst.cond, inst.rm, and inst.rn in Listing 2 represent the values of the instruction fields of MUL, which are cond, Rm, and Rn, respectively. Figure 1 shows the excerpt of the bit encoding of the MUL instruction taken from the ARM reference manual. The MUL instruction has five instruction fields: cond, S, Rd, Rm, and Rn. Expressions inst.cond, inst.rm, and inst.rn are method calls on inst, which is a function parameter to execute_mull, and they return the values of those instruction fields.

The type of inst is the Instruction class, which must be defined by the Pydgin user. Listing 3 is an example of the Instruction class for ARM, which is included in the source tree of Pydgin. We renamed the methods in the class such that the names are consistent with the field names in Figure 1. The methods in Instruction extract the value of an instruction field from the binary representation of the instruction. The binary representation is available from self.bits. For example, method rd extracts instruction field Rd in bits 16–19.

The Pydgin user can expect that the IDE for Python, which is the host language, reads the definition of the Instruction class and provides programming assistance to the user. For example, when the user writes execute_mul, auto-completion can be expected. When inst. is typed, a modern IDE would show the list of the word candidates that could follow inst., and this list would include cond, rm, and rn.

However, this auto-completion lacks domain-specific assistance. Even if the IDE correctly infers that the type of inst is the Instruction class, the candidates for the auto-completion would include imm24,

Listing 1: MUL instruction in the reference manual (ARM)

```
1  if condition_passed()
2      d = UInt(Rd);
3      n = UInt(Rn);
4      m = UInt(Rm);
5      setflags = (S == '1')
6      operands1 = SInt(R[n])
7      operands2 = SInt(R[m])
8      result = operand1 * operand2
9      R[d] = result<31:0>
10     if setflags then
11        APSR.N = result<31>;
12        APSR.Z = IsZeroBit(result);
```

Listing 2: MUL instruction in Pydgin (ARM)

```
1  def execute_mul( s, inst ):
2    if condition_passed( s, inst.cond ):
3      Rn, Rm = s.rf[ inst.rn ], s.rf[ inst.rm ]
4      result = trim_32(Rn * Rm)
5      s.rf[ inst.rd ] = result
6
7      if inst.S:
8        s.N = (result >> 31)&1
9        s.Z = result == 0
10
11     if inst.rd == 15:
12       return
13   s.rf[PC] = s.fetch_pc() + 4
```

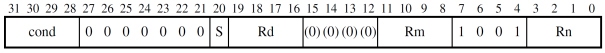

Figure 1: Encoding of the MUL instruction (ARM)

Listing 3: Definition of the Instruction class

```
1  class Instruction( object ):
2    ...
3    @property
4    def rd( self ): return (self.bits >> 16) & 0xF
5
6    @property
7    def rm( self ): return (self.bits >> 8) & 0xF
8
9    @property
10   def rn( self ): return self.bits & 0xF
11   ...
12   @property
13   def imm24( self ): return self.bits & 0xFFFFFF
14   ...
```

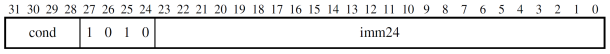

Figure 2: Encoding of the B instruction (ARM)

which is the method for extracting the imm24 field of the B (branch) instruction shown in Figure 2. The reason is that this method is included in the Instruction class. In Pydgin, all of the methods for extracting an instruction field are included in the Instruction class. Although imm24 is not available in the body of execute_mul, if the user selects imm24 after typing inst., the IDE would not warn the user of that incorrect selection. Note that the selection is valid from the host-language perspective of Python; it is invalid only from the domain-specific perspective of Pydgin.

To mitigate this problem, some readers might change the design of Pydgin to enable the user to define a different version of the Instruction class for a different instruction. For example, the type of inst passed to execute_mul could be MulInstruction, whereas that to execute_b could be BInstruction. Then, they could provide only the methods available for their instruction. However, this design requires the user to exert extra effort; numerous Instruction classes should be defined; thereby complicating the type inference by the IDE. If the host language were statically typed, that design would require the user to pay extra attention to the type of the inst parameter.

The domain-specific programming assistance in embedded DSLs is poorer than the assistance in external DSLs. If Pydgin were a standalone external DSL, it would report a compilation error when the user writes inst.imm24 in the body of the execute_mul function. A dedicated IDE for Pydgin would provide better programming assistance to prevent writing inst.imm24 mistakenly in the body of execute_mul.

Developing a dedicated IDE for an embedded DSL might be another option to mitigate the poor domain-specific assistance. Developing an external program-analysis tool might be another option. However, these options decrease the benefit of embedded DSLs; the user would not be able to use the DSL as a library for its host language. The user would be forced to write a program in a particular IDE, which may be unfamiliar. Moreover, developing IDEs for DSLs from scratch is too costly, according to the literature [3].

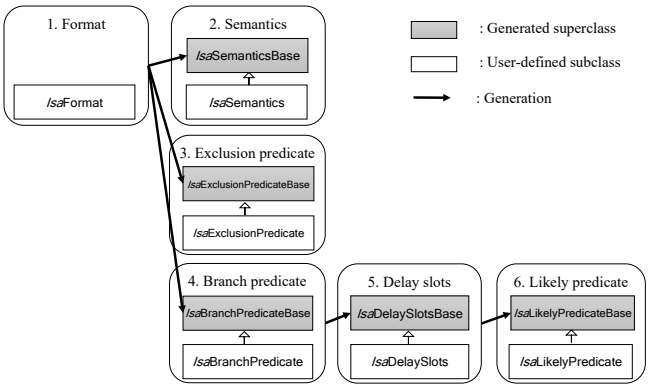

Figure 3: Structure of a program written in MELTRANS

3 MELTRANS: A PDL WITH DOMAIN-SPECIFIC PROGRAMMING ASSISTANCE

We present our design approach to an embedded DSL that enables domain-specific programming assistance. Our idea is to let the host-language IDE provide better programming assistance by domain-specific knowledge. To this end, a DSL program is split into multiple components for different concerns. We take advantage of the fact that some concerns contain useful domain-specific knowledge for the description of other concerns. The DSL programming is divided into multiple stages. When the user writes a DSL program in the early stage, the DSL compiler generates support code written in the host language from that user's DSL program. The domain-specific knowledge is encoded in that support code so that the IDE can provide domain-specific assistance for the later-stage DSL programming.

We have developed a PDL called MELTRANS with our design approach. MELTRANS is an embedded DSL hosted by Java for implementing a processor emulator with DBT. The emulator finally generated from a MELTRANS program is written in C++. We chose Java because it is a statically typed language, and its IDEs support rich programming assistance. Its environment-independent specification is also appropriate. For example, an integer value of type int is always 32-bit precision. The user of MELTRANS does not have to care about the execution environment to know the size of the integer.

A program in MELTRANS consists of six Java classes, each of which corresponds to a different concern about the description of a processor. There are dependencies among the concerns; some concerns contain domain-specific knowledge used when the user describes other concerns. The language runtime of MELTRANS utilizes these dependencies to provide domain-specific assistance by generating superclasses of the classes. Figure 3 shows the classes and concerns, as well as the relations among them. Each concern, except for the format concern, consists of a generated superclass and a user-defined subclass. The arrows are drawn from a concern to the superclass that is generated by the runtime of MELTRANS from the former. Each superclass serves as a canvas that provides

Listing 4: Class for the format concern for ARM

```
class ArmFormat {
  // instruction encodings
  String Inst105_A1_MUL =
    "31[cond]27[0b0000000]20[S]19[Rd]15[0b0000]11[Rm]7[0b1001]3[Rn]";
  String Inst016_A1_B =
    "31[cond]27[0b1010]23[imm24]";
    ...
};
```

```
String Inst105_A1_MUL =
    "31[cond]27[0b0000000]20[S]19[Rd]15[0b0000]11[Rm]7[0b1001]3[Rn]";
```

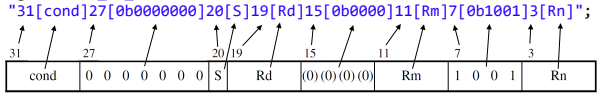

Figure 4: String literal for the MUL instruction (ARM)

auto-completion and limits the risk of error when the user defines the subclass for the concern.

The user defines classes in an order that is based on the dependencies among the concerns. The user starts by defining the format concern of a processor because it does not depend on any concern. For example, when the user develops a processor emulator for the ARM instruction set, the user starts by defining the class named ArmFormat. In MELTRANS, there is a naming convention in which all the class names start with the ISA name, such as ARM, and the concern name follows the ISA name. The definition of the format concern enables the language runtime of MELTRANS to generate superclasses for a concern, of which subclass the user then writes. The generated superclasses are named ArmSemanticsBase, ArmExclusionBase, and ArmBranchPredicateBase. In MELTRANS, the name of a superclass starts with the name of its subclass and ends with Base according to the convention. The domain-specific knowledge about the instruction format is encoded in the methods in the generated superclasses so that the IDE can exploit them for programming assistance. Thus, the user can expect domain-specific assistance by the IDE when writing subclasses ArmSemantics, ArmExclusion, and ArmBranchPredicate. After the user writes the class for the branch-predicate concern, named ArmBranchPredicate, MELTRANS generates a new Java class, ArmDelaySlotsBase, from that class. Then, the user writes ArmDelaySlots, and the same pattern follows.

3.1 Concerns

In this subsection, we present the six concerns in MELTRANS and demonstrate what domain-specific assistance the user can expect when writing each concern, except for the format concern.

3.1.1 Format. The language runtime of MELTRANS uses the domain-specific knowledge written in the class for the format concern to provide programming assistance for writing classes for other concerns. Because the class for the format concern is written first, domain-specific assistance is not available when the user writes this class. To ease the definition of the class with only general-purpose assistance, we designed a mini DSL in the string embedding style for specifying an instruction encoding.

The class for the format concern describes the bit encoding of every instruction. For each instruction, the class declares a field of String type with its initial value, representing the bit encoding written in the mini DSL. The name of this field is used as the identifier of that instruction. For example, the class for the ARM instruction set is shown in Listing 4, in which the Inst105_A1_MUL field represents the bit encoding of the MUL instruction. Inst105 denotes its section number in the ISA manual, and A1 indicates that the instruction is an ARM instruction and not a Thumb instruction.

Listing 5: Class for the semantics concern for ARM

```
class ArmSemantics extends ArmSemanticsBase {

  @Override
  void Inst105_A1_MUL(int cond, int S, int Rd, int Rm, int Rn) {
    if (isConditionPassed(cond)) {
      int result = reg[Rn] * reg[Rm];
      reg[Rd] = result;
      if (S == 1) {
        cpsr.N = extract(result, 31);
        cpsr.Z = (result == 0) ? 1 : 0;
      }
    }
  }

  @Override
  void Inst016_A1_B(int cond, int imm24) {
    if (isConditionPassed(cond)) {
      int imm32 = signExtend(imm24 << 2, 25);
      setNextPc(getPc() + 8 + imm32);
    }
  }
      ...
}
```

The user can easily write the string literal in our mini DSL by almost copying the description of the instruction in the ISA manual. For example, Figure 4 shows the string literal for the MUL instruction and its format description found in the ISA manual for ARM. The string literal is read as follows. The bits from the 31st to (but excluding) the 27th are used for the instruction field named cond, bits from the 27th up to the 20th must be 0000000 (0b is the prefix for binary numbers), the 20th bit is instruction field S, and so on.

3.1.2 Semantics. The class for the semantics concern must declare the methods specifying the behavior of every instruction. The name of the method for each instruction must be identical to the name of the corresponding field in the class for the format concern. For example, Listing 5 shows the class for the semantics concern for ARM. In Listing 5, the method name for MUL is Inst105_A1_MUL, whereas that for B is Inst016_A1_B. These names are found in the ArmFormat class that the user wrote for the format concern. The parameters to the methods are the instruction fields of each instruction. They are also defined in the class for the format concern. For example, the parameters to the Inst105_A1_MUL method are cond, S, Rd, Rm, and Rn, and they are defined in the string literal given to the Inst105_A1_MUL field in the ArmFormat class.

The bodies of those methods can be written by almost copying the corresponding description in the ISA manual. For example, the body of Inst105_A1_MUL is fairly identical to the description of the behavior of MUL in the ISA manual shown in Listing 1. It multiplies

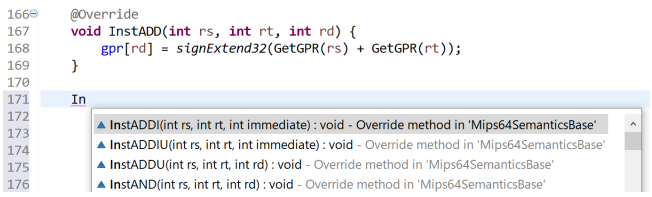

Figure 5: Auto-completion of instruction names

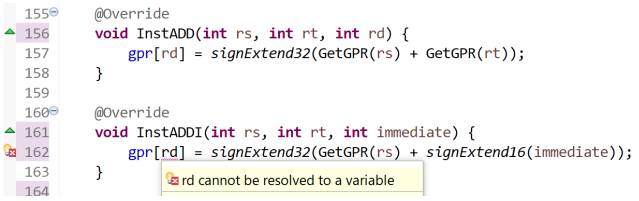

Figure 6: Instruction field rd is not available for ADDI

the Rn register by the Rm register and stores the result into the Rd register. Then it updates the current program status register.

The user can expect domain-specific programming assistance by the IDE when writing the class for the semantics concern. First, all of the methods that the user must write when specifying the instruction's behavior are already declared in the superclass generated by MELTRANS, although their bodies are empty. For example, superclass ArmSemanticsBase declares the Inst105_A1_MUL and Inst016_A1_B methods. The ArmSemantics class, which the user must write, overrides them. Thus, when the user types the first few letters of a method name, the IDE shows the candidate list of methods for auto-completion, as illustrated in Figure 5. The user can simply choose one of them to obtain a skeleton of a method declaration. In Listing 5, lines 3, 4, and 13 can be automatically completed when the user writes the Inst105_A1_MUL method.

Second, the instruction fields are encoded into the parameters to the method specifying the instruction's behavior. This also improves the domain-specific assistance by the IDE. For example, the IDE reports an error when the user writes the name of an instruction field unavailable for the instruction that is being written by the user. Figure 6 shows an error message that is reported when the user attempts to store a value into the rd register (the register specified by instruction field rd), which is not available for the ADDI instruction in MIPS64. This is a typical mistake unless the user carefully reads the ISA manual because most instructions in MIPS64, such as ADD, store the arithmetic results into the rd register. ADDI exceptionally stores the result into the rt register. In MELTRANS, this mistake is detected as an error of using an undeclared parameter or variable. The domain-specific knowledge regarding which instruction fields are available is encoded into the method parameters so that the domain-specific assistance will be provided as normal programming assistance in the host language.

This design of MELTRANS also helps the IDE to detect another typical mistake. Although the result of ADD in MIPS64 is stored in the rd register, the user might misinterpret that ADD is a double-operand instruction and store the result into the rt register (the

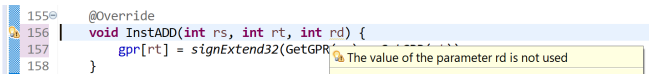

Figure 7: Result of ADD is stored in an incorrect register

second operand). The IDE may detect this mistake as an error because the third method parameter, rd, is never used in the method body of InstADD (Figure 7). When the instruction is executed, all of the instruction fields should be used. MELTRANS exploits this fact and thus passes them through the method parameters to the method implementing the instruction's behavior, such as InstADD.

3.1.3 Exclusion Predicate. The class for the exclusion-predicate concern describes how to disambiguate the instructions that share the same opcode. This is necessary for certain ISAs such as ARM and RH850.

Suppose that we implement an emulator for the RH850 instruction set. The class for the format concern would include the following field declarations for the DIVH and RIE instructions:

```
String Inst028_DIVH = "15[r]10[0b000010]4[R]";
String Inst086_RIE = "15[0b0000000001000000]";
```

When both the instruction fields, r and R, in DIVH are 00000, we cannot distinguish DIVH and RIE because their bit patterns are identical. For disambiguation, the ISA manual for RH850 specifies that instruction field r or R in the DIVH instruction must not be zero. We call this the exclusion predicate of the DIVH instruction. In the class for the exclusion-predicate concern for RH850, the exclusion predicate of the DIVH is written as follows:

```
@Override
boolean Inst028_DIVH(int r, int R) {
    return r == 0 | R == 0;
}
```

The Inst028_DIVH method returns true when the given instruction field, r or R, is zero; hence, it is not valid. If it returns true, then the current instruction word is not that of DIVH, but that of RIE.

The user can expect domain-specific programming assistance when writing this class. This user-defined class must inherit from the superclass generated from the class for the format concern. The superclass declares all of the methods that the user may declare in its subclass. These methods return false. Therefore, the user can expect domain-specific assistance similar to the assistance that can be expected for the semantics concern.

3.1.4 Branch Predicate. The class for the branch-predicate concern describes which instructions are branch instructions. It declares a method for every instruction, and the method returns true if the instruction is a branch.

For example, the B instruction in ARM is a branch instruction. The user can specify this by defining a method in the class for the branch-predicate concern as follows:

```
@Override
boolean Inst016_A1_B() {
    return true;
}
```

The return value indicates that the B instruction is a branch instruction.

For some instructions, its instruction fields determine whether the instruction is a branch. For example, the ADD instruction in ARM is a branch instruction when instruction field Rd (destination register) is the program counter. Otherwise, ADD is not a branch instruction. To support this, the class for the branch-predicate concern can declare a method taking all the instruction fields. Hence, the branch predicate for the ADD is written as follows:

```
@Override
boolean Inst005_A1_ADD(int cond, int S, int Rn, int Rd, int imm12) {
    return Rd == 15;
}
```

The expression in the return statement indicates that the ADD instruction is a branch instruction when Rd is the program counter (register 15).

The user can expect domain-specific programming assistance when writing the class for the branch-predicate concern. This class must inherit from the superclass generated by the language runtime of MELTRANS from the class written for the format concern. The generated class declares two methods for every instruction. One takes no parameter while the other takes all the instruction fields as parameters. Because the methods in the superclass return false, the subclass does not need to declare the methods for instructions that are always non-branch instructions. When the subclass needs to declare the method, the user can use auto-completion by the IDE to select one method for each branch instruction.

3.1.5 Delay Slots. The emulators for some ISAs must consider delay slots for each branch instruction. In MELTRANS, the user can describe the delay slots by writing a class for the delay-slots concern. This class declares a method for the branch instruction with delay slots; the method returns the number of delay slots.

For example, the BFS instruction in the SH instruction set is a branch instruction with a delay slot. The user can specify this by writing a method in the class for the delay-slots concern as follows:

```
@Override
int InstBFS() {
    return 1;
}
```

The return value indicates that the BFS instruction has one delay slot.

The user can also expect domain-specific programming assistance when writing a class for the delay-slots concern. The user-defined class inherits from the superclass generated by MELTRANS from the class for the branch-predicate concern. The superclass declares a method for every branch instruction. Because the method in the superclass returns 0, its subclass can declare only the methods for the branch instructions with more than zero delay slots.

Note that the superclass declares only the methods for branch instructions and not all instructions. Therefore, the method list for auto-completion is more accurate. Even if the user declares a method for a non-branch instruction, the IDE will report a warning message because the method does not override any method in the superclass (Figure 8). To determine which instruction is a branch one, MELTRANS investigates the bodies of the methods in the class for the branch-predicate concern. If the method may return true, MELTRANS considers the corresponding instruction as a branch one.

```
87    @Override
88    int InstBEQ() {
89        return 1;
90    }
91
92    @Override
93    int InstADD() {
94
95    }
```

The method InstADD() of type Mips64DelaySlots must override or implement a supertype method

Figure 8: No overridden method in the superclass because the method is not for a branch instruction

Algorithm 1 Translation of a basic block

Input: memory model memory, start address of BB addr, strategy object isaStrategy
Output: llvm IR corresponding to BB

```
1: repeat
2:     iword ← fetchInstruction(memory, addr)
3:     (inst, fields) ← isaStrategy.decode(iword)
4:     inst.generateIr(fields)                          ▷ semantics
5:     addr ← addr + inst.length                        ▷ format
6: until inst.isBranch(fields)                          ▷ branch predicate
7: if inst.delaySlots > 0 then                          ▷ delay slots
8:     generateDelaySlotsIr(memory, addr, inst)         ▷ delay slots and likely predicate
```

3.1.6 Likely Predicate. The class for the branch-likely concern declares a method for every branch-likely instruction. The method must return true.

Some ISAs have branch-likely instructions. They skip the execution of the following instructions in the delay slots when the branch is not taken. From the class for the delay-slots concern, MELTRANS generates the superclass of the class for the likely-predicate concern. The superclass declares a method for every branch instruction with delay slots. This method returns false because MELTRANS assumes that all branch instructions are not branch-likely instructions by default. Hence, the subclass needs to declare only a method for branch-likely instructions. The generated superclass enables domain-specific assistance similar to the assistance for the delay-slots concern.

3.2 Generated Processor Emulator

To explain how the six concerns contribute to the DBT, we show our ISA-independent skeleton of DBT in Algorithm 1. The name of the concern denotes that the operation in its source line depends on that concern. Algorithm 1 dynamically translates the guest instructions in a basic block (BB) into the LLVM [14] intermediate representation (IR). Once the LLVM IR is available, the generated emulator uses the LLVM JIT engine to generate the host native code. Algorithm 1 is based on the strategy pattern [7] win which ISA-dependent parts are implemented in the strategy objects. The language runtime of MELTRANS generates ISA-dependent parts from the six concerns written in MELTRANS.

Algorithm 1 takes the memory model memory, the start address of a BB addr, and a strategy object isaStrategy as parameters, and it generates the LLVM IR that emulates the behavior of the instructions in the BB. In lines 1–6, the emulator translates a guest BB into the LLVM IR. For each iteration in lines 2–5, a single guest instruction is translated into the LLVM IR instructions, which are appended to the resulting LLVM IR. In line 3, the decode method is called with arguments isaStrategy and iword to decode instruction iword in the ISA-specific way. The decode method returns the

identified instruction and its instruction fields as inst and fields, respectively. The implementation of decode can be generated from the format and exclusion-predicate concerns using the algorithm proposed in [18]. In line 4, the emulator generates the LLVM IR code according to the identified instruction and its instruction fields. The LLVM IR for the instruction is generated by the AST of the method for the instruction in the semantic concern. We use the deep reification approach proposed in [4] to obtain ASTs from a program written in MELTRANS. In line 5, the emulator increments address addr such that it points to the next instruction. The length of the instruction is retrieved from the format concern. In line 6, the emulator checks whether the decoded instruction is a branch instruction. If it is a branch instruction, the emulator terminates the iteration; otherwise, the emulator goes back to line 2. In lines 7–8, the emulator translates the instructions in the delay slots immediately after the branch, if any. The number of delay slots is obtained from the delay-slots concern.

4 EXPERIMENTAL RESULTS

To validate our design, we implemented several processor emulators with MELTRANS and conducted experiments using them.

4.1 Amount of Code to Be Written

To determine whether the domain-specific assistance is achieved without increasing the amount of code to be written by the user, we compared MELTRANS with Pydgin in terms of the code metrics of their programs for ARM. We used Eclipse as an IDE in this experiment.

The results are summarized in Table 1. For example, the first row in Table 1 shows that the format concern implements 173 instructions, and its code accounts for 5.87% of the entire code; the lines of code (LOC) excluding comments and the blank lines is 180, LOC per instruction is 1.04, and LOC automatically completed by the IDE is zero and accounts for 0% of the format concern. The format concern includes one method, and its cyclomatic complexity [8] is 1.0 on average.

The total LOC per instruction for ARM in MELTRANS is 17.74. This result is comparable with the LOC per instruction of 16.18 for ARM in Pydgin. MELTRANS requires an additional 1–2 LOC per instruction compared with Pydgin for ARM. This difference appears to be caused by the difference between the host languages. In general, Java is more verbose than Python.

In MELTRANS, 39.49% of the code could be automatically completed by the IDE. Although auto-completion may be available in each line of the program, we counted only lines for the method templates completed by the IDE, which consisted of method signatures with the @Override annotation and a pair of opening and closing braces. If we remove the automatically completed lines from the total amount of code, the remaining LOC becomes 1,857, and the LOC per instruction becomes 10.73. The result shows that our domain-specific assistance can effectively reduce the amount of code to be written by the user.

4.2 Performance

To determine whether the generated processor emulators run at practical speeds, we compared the emulator for ARM generated by

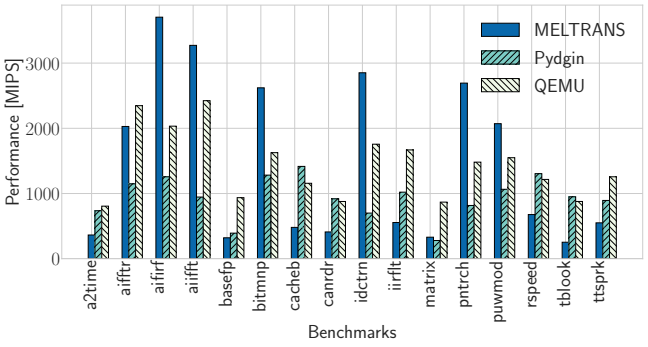

Figure 9: Performance of the emulators (ARM)

MELTRANS with state-of-the-art QEMU and the emulator generated by Pydgin in terms of their simulation speed. We used benchmark programs from the EEMBC [22] Autobench benchmark suite, which is one of the de-facto industrial standard benchmarks for comparing embedded processors. We performed all of the measurements presented in this paper on a Linux-based desktop machine with a 64-bit Core i7 7700T at 2.9 GHz, disabling Turbo Boost, and a 16-GB main memory. We built gcc 6.1.0 and used it with the -O2 flag to cross-compile the benchmark programs. We also used gcc 7.5.0, which is the default compiler in the host operating system, to build the processor emulators. The generated emulators used the JIT engine of LLVM 10.0 to translate LLVM IR into the host AMD64 instructions. Our emulator uses a superblock as a translation unit. A superblock consists of all BBs that are traceable through direct branch instructions, except call instructions when the emulator needs translation.

Figure 9 depicts the results. Our emulator achieved 1,449 MIPS on average. A minimum of 253 MIPS was observed when the emulator ran tblook, and the maximum was 3,704 MIPS when it ran aifirf. It appeared that the emulator runs fast when highly executed superblocks contain loops. In such a case, the superblock was well optimized by LLVM.

Our emulator outperformed the other two emulators in seven of the 16 programs. On average, QEMU and Pydgin executed the programs in 1,430 and 945 MIPS, respectively. Because all of these emulators use different translation strategies, the performance tendencies in the benchmark programs appeared to be different among the emulators.

4.3 Generality

To determine whether MELTRANS is general enough to generate emulators for multiple ISAs, we implemented MIPS64, RH850, SH, RISC-V, and RX, which are widely used in industry in addition to ARM, and compared the code metrics of the programs. Six concerns were used to write these programs. The results are listed in Table 2. For example, the first row in Table 2 shows that the program for ARM consists of concerns 1–4 and implements 173 instructions, its LOC is 3,069, the LOC per instruction is 17.74, and the IDE automatically completes 39.49% of the LOC. Each number of a concern corresponds to the number shown in Figure 3. The table shows that the IDE can automatically fill 24.17%–49.02% of the LOC.

Table 1: Comparison of code metrics among PDLs (ARM)

PDL	Concern	Instructions	LOC	%	LOC per instruction	Completed LOC	Completed LOC (%)	Methods	Complexity
MELTRANS	Format	173	180	5.87	1.04	0	0.00	1	1.00
	Semantics	173	1,953	63.64	11.29	519	26.57	191	2.65
	Exclusion predicate	173	695	22.65	4.02	519	74.68	173	1.00
	Branch predicate	173	235	7.66	1.36	174	74.04	58	1.00
	All	173	3,069	100.00	17.74	1212	39.49	423	1.75
Pydgin	-	62	1003	100.00	16.18	0	0.00	116	3.1

Table 2: Comparison of code metrics among ISAs

ISA	Concerns	Instructions	LOC	LOC per instruction	Completed LOC (%)
ARM	1, 2, 3, 4	173	3,069	17.74	39.49
MIPS64	1, 2, 3, 4, 5, 6	250	2,185	8.74	49.02
RH850	1, 2, 3, 4	223	3,389	15.20	24.17
SH	1, 2, 4, 5	154	1,334	8.66	40.48
RISC-V	1, 2, 4	54	397	7.35	46.85
RX	1, 2, 4	473	3,619	7.65	44.43

Although the percentage of the code automatically completed by the IDE depends on the complexity of the ISA, all six concerns are sufficient to describe these ISAs. Because other commercial ISAs such as PowerPC and TriCore are similar to these ISAs, it appears that MELTRANS is general enough to describe many practical ISAs.

5 RELATED WORK

Programming Assistance in Embedded DSLs. One of the advantages of embedded DSLs is its low implementation cost. However, domain-specific programming assistance in embedded DSLs is poor. Researchers have tackled this problem, and their solutions appear to be able to be combined with our approach.

Dinkelaker [5] proposed the Eclipse plug-in called TigersEye, which enables the use of the domain's established syntax in programs in an embedded DSL. Nosal et al. [17] proposed techniques for customizing host IDEs for embedded DSLs, including the prevention of inexpert editing, code completion, and error reporting.

Embedded DLSs for Emulator Generation. Several embedded DSL approaches have been proposed for generating processor emulators; however, they do not provide domain-specific assistance. Pydgin [15] uses a Python-based embedded DSL for generating processor emulators. It uses PyPy's [2] meta-tracing JIT compiler for DBT. The resulting processor simulator runs the guest program with tracing JIT compilation. ArchC [1] is a SystemC [20]-based PDL. SystemC is also an embedded DSL hosted by C++ for system simulation. A processor description in ArchC can be compiled as a C++ program and runs as an interpretive processor emulator. Wagstaff et al. [24] proposed a method to generate processor emulators with DBT using the description written in ArchC.

Engel et al. [6] and Okuda et al. [19] proposed frameworks in C/C++ to generate static binary translators and dynamic binary translators, respectively. In these systems, the user can develop a processor emulator with binary translation as if an interpretive one is developed.

6 CONCLUSION

This paper presented our design approach for developing an embedded DSL with domain-specific programming assistance. The proposed approach divides DSL programming into multiple stages, and the language processor of that DSL generates a program from the program written by the user in an earlier stage. The generated program exploits the inheritance mechanism to provide domain-specific assistance to the user. To demonstrate our approach, we explain the design of our PDL named MELTRANS, which is an embedded DSL hosted by Java. We implemented several emulators in MELTRANS and experimentally confirmed that our domain-specific programming assistance effectively reduces the amount of code that needs to be written by the user.

REFERENCES

[1] Rodolfo Azevedo, Sandro Rigo, Marcus Bartholomeu, Guido Araujo, Cristiano Araujo, and Edna Barros. 2005. The ArchC Architecture Description Language and Tools. *Int. J. Parallel Program.* 33, 5 (Oct. 2005), 453–484.

[2] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09)*. ACM, 18–25.

[3] Barrett R. Bryant, Jeff Gray, and Marjan Mernik. 2010. Domain-Specific Software Engineering. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER '10)*. ACM, 65–68.

[4] Shigeru Chiba, YungYu Zhuang, and Maximilian Scherr. 2016. Deeply Reifying Running Code for Constructing a Domain-Specific Language. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16)*. ACM, Article 1, 12 pages.

[5] Tom Dinkelaker, Michael Eichberg, and Mira Mezini. 2011. Incremental Concrete Syntax for Embedded Languages. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC '11)*. ACM, 1309–1316.

[6] Frank Engel, Johannes Nührenberg, and Gerhard P. Fettweis. 2000. A Generic Tool Set for Application Specific Processor Architectures. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign (CODES '00)*. ACM, 126–130.

[7] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. 1993. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP '93)*. Springer-Verlag, 406–431.

[8] Geoffrey K. Gill and Chris F. Kemerer. 1991. Cyclomatic complexity density and software maintenance productivity. *IEEE transactions on software engineering* 17, 12 (1991), 1284–1288.

[9] George Hadjiyiannis, Pietro Russo, and Srinivas Devadas. 1999. A Methodology for Accurate Performance Evaluation in Architecture Exploration. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference (DAC '99)*. ACM, 927–932.

[10] Mark R. Hartoog, James A. Rowson, Prakash D. Reddy, Soumya Desai, Douglas D. Dunlop, Edwin A. Harcourt, and Neeti Khullar. 1997. Generation of Software Tools from Processor Descriptions for Hardware/Software Codesign. In *Proceedings of the 34th Annual Design Automation Conference (DAC '97)*. ACM, 303–306.

[11] ARM Holdings. 2014. ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition. *Arm Holdings* (2014).

[12] Rola Kassem, MikaëL Briday, Jean-Luc BéChennec, Guillaume Savaton, and Yvon Trinquet. 2012. Harmless, a hardware architecture description language dedicated to real-time embedded system simulation. *J. Syst. Archit.* 58, 8 (Sept. 2012), 318–337.

[13] Marco Kaufmann, Matthias Häsing, Thomas Preußer, and Rainer Spallek. 2011. The Java Virtual Machine in Retargetable, High-Performance Instruction Set Simulation. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11)*. ACM, 21–30.

[14] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization (CGO '04)*. IEEE, 75–86.

[15] Derek Lockhart, Berkin Ilbeyi, and Christopher Batten. 2015. Pydgin: generating fast instruction set simulators from simple architecture descriptions with meta-tracing JIT compilers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 256–267.

[16] Prabhat Mishra and Nikil Dutt. 2008. *Processor Description Languages*. Morgan Kaufmann Publishers Inc.

[17] Milan Nosál, Jaroslav Porubän, and Matú Sulír. 2017. Customizing host IDE for non-programming users of pure embedded DSLs: A case study. *Computer Languages, Systems & Structures* 49 (2017), 101–118.

[18] Katsumi Okuda and Haruhiko Takeyama. 2016. Decision Tree Generation for mDecoding Irregular Instructions. In *Proceedings of the 2016 Conference on Design, Automation & Test in Europe (DATE '16)*. EDA Consortium, 1592–1597.

[19] Katsumi Okuda, Minoru Yoshida, Haruhiko Takeyama, and Minoru Nakamura. 2017. Automated generation of dynamic binary translators for instruction set simulation. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 214–219.

[20] Preeti Ranjan Panda. 2001. SystemC: A Modeling Platform Supporting Multiple Design Abstractions. In *Proceedings of the 14th International Symposium on Systems Synthesis (ISSS '01)*. ACM, 75–80.

[21] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. 1999. LISA Machine Description Language for Cycle-accurate Models of Programmable DSP Architectures. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference (DAC '99)*. ACM, 933–938.

[22] Jason A. Poovey, Thomas M. Conte, Markus Levy, and Shay Gal-On. 2009. A Benchmark Characterization of the EEMBC Benchmark Suite. *IEEE Micro* 29, 5 (Sept. 2009), 18–29.

[23] Mehrdad Reshadi, Nikil Dutt, and Prabhat Mishra. 2006. A retargetable framework for instruction-set architecture simulation. *ACM Trans. Embed. Comput. Syst.* 5, 2 (May 2006), 431–452.

[24] Harry Wagstaff, Miles Gould, Björn Franke, and Nigel Topham. 2013. Early Partial Evaluation in a JIT-Compiled, Retargetable Instruction Set Simulator Generated from a High-Level Architecture Description. In *Proceedings of the 50th Annual Design Automation Conference (DAC '13)*. ACM, Article 21, 6 pages.