ISSN 1392 – 124X INFORMATION TECHNOLOGY AND CONTROL, 2013, Vol. 42, No. 1

Easytime++: A Case Study of Incremental Domain-Specific Language Development

Iztok Jr. Fister[1], Tomaž Kosar[2], Iztok Fister[3], Marjan Mernik[4]

University of Maribor, Faculty of Electrical Engineering and Computer Science
Smetanova 17, SI-2000 Maribor, Slovenia
e-mail: iztok.fister@guest.arnes.si[1], tomaz.kosar@uni-mb.si[2],
iztok.fister@uni-mb.si[3], marjan.mernik@uni-mb.si[4]

crossref http://dx.doi.org/10.5755/j01.itc.42.1.1968

Abstract. EasyTime is a domain-specific language (DSL) for measuring time during sports competitions. A distinguishing feature of DSLs is that they are much more amenable to change, and EasyTime is no exception in this regard. This paper introduces two new EasyTime features: classifications of competitors into categories, and the inclusion of competitions where the number of laps must be dynamically determined. It shows how such extensions can be incrementally added into the base-language reusing most of the language specifications. Two case studies are presented showing the suitability of this approach.

Keywords: Domain-specific languages; Language composition; Incremental language development; EasyTime.

1. Introduction

Domain-specific languages (DSLs) are languages tailored to specific application domain [1–4]. They offer substantial gains regarding expressiveness and ease of use compared with general-purpose languages (GPLs) within their domain of application [5–7]. However, DSLs are more amenable to changes [1, 8] since stakeholders' requirements frequently change. In order to design and implement DSLs more easily, we need to develop fully modular, extensible, and reusable language descriptions, whilst some of the descriptions could even be inferred from DSL programs [9, 10]. The language designer wants to include new language features incrementally as the programming language evolves. Ideally, a language designer would like to build a language simply by reusing different language definition modules (language components), such as modules for expressions, declarations, etc., as well as to extend previous language specifications. In the case of general software development the use of object-oriented techniques and concepts like encapsulation and inheritance, greatly improves incremental software development, whilst reusability is even further enhanced using aspect-oriented techniques [11]. The object-oriented, as well as the aspect-oriented techniques and concepts, have also been integrated into programming language specifications [12, 13] making new features more easily implemented. One of such tools, where object-oriented and aspect-oriented concepts have been incorporated, is the LISA tool [8, 14]. This paper shows how LISA is used within the incremental development of Easy-Time DSL, which has been developed recently for measuring time at different sports competitions (e.g., triathlon, cycling) [15, 16]. EasyTime DSL has already proved to be successful when used at real sport events (e.g., World Championship in the double ultra triathlon in 2009, National (Slovenian) Championship in the time-trials for cycles in 2010), so the requirements are changing quickly. Recent extensions to EasyTime have included the possibility of classifying competitors into different categories, where the number of laps is different for each category, and the inclusion of competitions where the number of laps can be dynamically determined during a competition (e.g., biathlon, where the number of extra laps depends on missed shots). The objective of this paper is to introduce EasyTime++ DSL, which supports these new extensions, as well as to show how such an extension can be incrementally developed using the introduced LISA tool.

The structure of this paper is as follows: in Section 2 an overview of the language composition is presented. Section 3 briefly introduces EasyTime DSL, whilst the core of this paper is Section 4, which describes how the extensions in EasyTime++ have been specified and implemented. Some examples are

presented in Section 5. The paper is concluded with Section 6, where a brief overview and word about future work is described.

2. Related work

Several kinds of language composition have been identified in the literature [8, 17–24]. In their recent paper [17], Erdweg et al., point out that language composition has obtained little attention, that it is still insufficiently understood, and that the terminology is confusing thus indicating that the research is inadequate, as yet. Erdweg et al. identified the language composeability not as a property of languages themselves, but as a property of language definition (e.g., how language specifications can be composed together). The following types of language composition have been distinguished in [17]: language extension (which subsumes also language restriction), language unification, self-extension, and extension composition. In language extension the specifications of base language B are extended with a new language specification fragment E, which typically makes little sense when regarded independently from the base language B. Hence, language B is a dominant language, which can be a DSL or a GPL, and serves as a base for other languages. Language extension, as a kind of language composition, is denoted as $B \lhd E$ indicating that the base language B has been extended with the language E. The LISA tool supports language extensions when single attribute grammar inheritance [8] is employed. As is shown in Section 4, EasyTime++ is a language extension over the base language Easy-Time (EasyTime / EasyTime++). In language unification the composition of language specifications is not based on the dominance of one language, but is based on equal terms. The dominance of one language over another does not exist and both language specifications are complete and standalone (note that in the case of language extension the language specifications for the extended part makes little sense alone). Language unification, as a kind of language composition, is denoted as $L_1 \uplus_g L_2$, describing the language composition of languages L_1 and L_2 using a glue code g. Since LISA supports multiple attribute grammar inheritance [8], language unification is easily achieved by inheriting both language specifications (from L_1 and L_2), where the glue code is specified as a new language specification fragment. In selfextension the language specifications do not change. The language itself is powerful enough for new extensions to be implemented using macros, function composition, and libraries that provide domain-specific constructs. This form of language composition is called 'pure language embedding' [3]. Functional languages are these languages particularly suitable for self-extension. Self-extension, as a kind of language composition, is denoted as $H \leftarrow E$ indicating that the host language H has been self-extended with the embedded language E. The last form of language composition is extension composition, which describes how language specifications also support the combination of various language compositions. That is showing how different compositions can work together. This kind of language composition can also be described as high-order language composition. Languageunification allows for such higher-order compositionper se (e.g., $L_1 \uplus_g (L_2 \uplus_h L_3)$). Whilst some other useful examples of higher-order language composition like $(B \lhd E_1) \lhd E_2$, and $B \lhd (L_1 \uplus_g L_2)$ can not always be easily achieved. Extension compositions involving language extension and language unification can also be easily achieved in the LISA tool.

In addition to LISA, which has been in existence since 1999, there are also other similar tools (e.g., Phobos [18], JastAdd [19], Silver [20], XMF [21], Tatoo [22], MontiCore [23], JAYCO [25], UUAG [26]) that enable various language compositions. Note, that the most well-known tools for syntax and the semantic specification of programming languages, Lex and Yacc [27], don't support language composition per se. For example, language extension is possible by manually changing base language specifications B by invasively adding the specification for extended language E. Hence, change is done in a non-disciplined manner, thus prohibiting further reuse of specifications. On the other hand, language composition can be done on top of Lex and Yacc (e.g., [28]). Here, it is desirable to briefly mention JastAdd [19] and Silver [20], since both are based on Attribute Grammars, as in the LISA case. JastAdd [19] is centered around object-oriented representation of the abstract syntax tree (AST). Non-terminals act as abstract super classes and productions act as specialized concrete subclasses that specify the syntactic structure, attributes, and semantic rules. All these elements can be inherited, specialized, and overridden within subclasses. The idea of aspect-orientation inJastAdd is to define each aspect of the language in a separate class and then weave them together at appropriate places. The JastAdd system is a class weaver: it reads all the JastAdd modules and weaves the fields and methods into the appropriate classes during the generation of the AST classes. Developers have the possibility of combining various language specificationsfollowing the separation of different language aspects amongst different classes. Silver [20] uses a concept called 'forwarding' to achieve modular language extensions, where the extension construct is translated into semantically equivalent constructs within the host language. Hence, forwarding only allows those new constructs that can be expressed as a combination of existing language constructs. Additional Silver features like: with-clause, auto-copying of inherited attributes, collection attributes, pattern matching, and type-safe polymorphic lists, allow for the host language to be extended in a more flexible manner, although still restrictive.

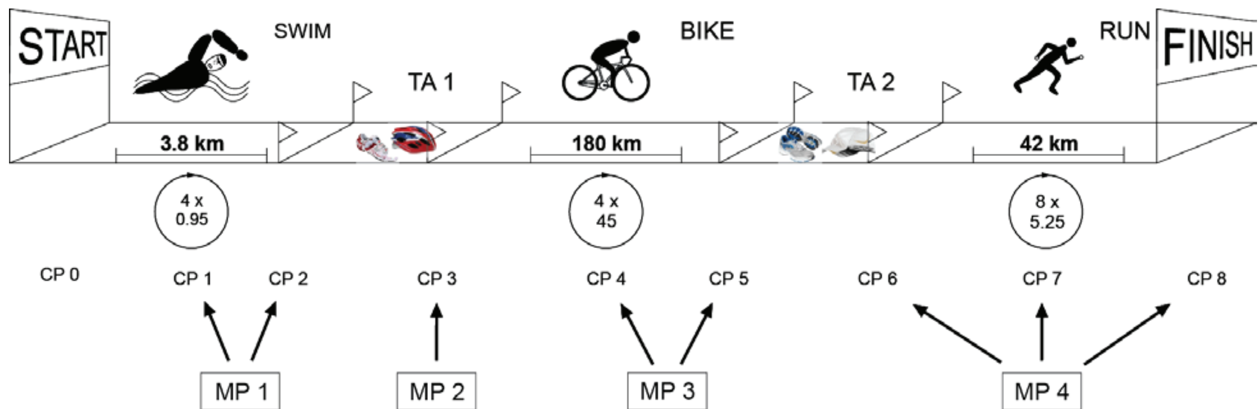

Figure 1. Measuring time in Ironman triathlon

3. EasyTime

EasyTime was developed for measuring time during Double ultra triathlon in 2009. At that time, the organizers of this competition were confronted with the problem of how to measure the times of competitors within three disciplines using a limited number of measuring devices. Besides this limitation, measures needed to be reliable and accurate, especially, because of its long duration. Although the measuring time for the triathlon was our first specific task, the goal was to develop a DSL for measuring time for any competition. A domain analysis was performed [15] using feature diagrams [29] with the aim of identifying common and variable concepts, their relations, and structure of particular concepts. In the case of EasyTime, the concept race consists of subconcepts: events (e.g., swimming, cycling, and running), control points (starting and finishing lines, the number of laps), the measuring time (updating time and decrementing laps), optional transition area (difference between the finish and start times), and agents (automatic or manual). In the next step, these concepts were mapped to the context-free grammar nonterminalsof EasyTime. Finally, its whole syntax and semantics were developed [15].

In order to illustrate the power of EasyTime, lets describe the Ironman triathlon, as presented in Figure 1. This triathlon consists of: 3.8 km swim, 180 km cycling, and a 42 km run. These disciplines run one after another with two interruptions: In the first, those competitors who have finished with swimming prepare themselves for the cycling, whilst in the second, those competitors who have finished the cycling prepare themselves for running. Both interruptions occur within so-called transition areas. Their times spent within these areas are added to their swimming, cycling, and running times, in order to obtain the total times of specific competitors.

Typically, the organizers divide those courses on which they run particular disciplines into laps because of easier management. As can be seen in Figure 1, competitors need to accomplish 4 laps of swimming, 4 laps of cycling, and 8 laps of running. These laps represent another demand for the measuring time in such competitions because, besides the intermediate times of each lap, decrementing also needs to be performed. In order to reduce the number of measuring devices, a measuring point (MP in Figure 1) at which the intermediate time is measured and the number of laps is decremented, can be incorporated. In other words, when the number of laps is zero the last intermediatetime becomes the final time of a specific discipline.

This characteristic of the triathlon is put to profitableuse by EasyTime. In fact, EasyTime is a DSL that enables the organizers of sporting competitions to adapt measuring systems for various kinds of competitions, reduce the number of measuring devices, and achieve accuracy and reliability. The EasyTime program runs on a measuring system and employs a set of agents that control the measuring devices. For measuring time during Inronman, as illustrated in Figure 1, the EasyTime program presented in Program 1 is used.

At the start of Program 1, two agents are defined: The former describes a measuring device on which manual measuring time is performed on a portable computer by an operator, whilst the latter denotes a measuring device that automatically tracks an event caused when a competitor crossing the measuring place, based on RFID technology [30]. Typically, the automatic measuring place is implemented as a mat that acts as an antenna having two functions: Firstly, the antenna induces a passive tag that is born by competitor. Secondly, this induced tag acts as a transmitter that transmits its identification code to a measuring device. The transmitted code is detected by the measuring device as an event. This event is transmitted to the measuring system and recorded into a database by an agent.

After the agents definition in Program 1, a declaration of variables follows. For each measuring place, two variables are defined in general: an intermediate time INTERx and a laps counter

Program 1. EasyTime program for measuring time in Ironman

```
1: 1 manual "man.dat";     //definition of manual agent
2: 2 auto 192.168.225.100; //definition of automatic agent
3: // definition of variables
4: var ROUND1 ≔ 4;
5: var INTER1 ≔ 0;
6: var SWIM ≔ 0;
7: var TRANS1 ≔0;
8: var ROUND2 ≔ 4;
9: var INTER2 ≔0;
10: var BIKE ≔ 0;
11: var TRANS2 ≔0;
12: var ROUND3 ≔8;
13: var INTER3 ≔ 0;
14: var RUN ≔ 0;
15: // definition of measuring place 1
16: mp[1] → agnt[1] {
17: (true) → upd INTER1;
18: (true) → dec ROUND1;
19: (ROUND1 == 0) → upd SWIM;
20: }
21: // definition of measuring place 2
22: mp[2] → agnt[1] {
23: (true) → upd TRANS1;
24: }
25: // definition of measuring place 3
26: mp[3] → agnt[2] {
27: (true) → upd INTER2;
28: (true) → dec ROUND2;
29: (ROUND2 == 0) → upd BIKE;
30: }
31: // definition of measuring place 4
32: mp[4] → agnt[2] {
33: (true) → upd INTER3;
34: (ROUND3 == 8) → upd TRANS2;
35: (true) → dec ROUND3;
36: (ROUND3 == 0) → upd RUN;
37: }
```

ROUNDx. The final achievements of a competitor for specific disciplines are saved within variables SWIM, BIKE, and RUN.

The EasyTime program is completed by definitions of measuring places mp[i], where i represents its identification number that must be defined uniquely. The measuring place represents a physical device that is connected to a measuring device. The measuring device can support more measuring places simultaneously. Conversely to a measuring place, a control point (CP in Figure 1) represents an event from a logical point of view and denotes the specific location on the course, where the referees need to track the time information about competitors. As a matter of fact, the control points in EasyTime are directly mapped into variables. The identification numbers of each agent responsible for transmitting the events is assigned to each measuring place. For example, manual agent agnt[1] in line 16 of Program 1 controls the first measuring place.

Before recording the event into a database, a sequence of statements in curly brackets are interpreted on an abstract machine (AM). These statements are in forms of (**predicate**) → **operation**, where **operation** denotes a sequence of instructions that are executed when the value of **predicate** returns value $true$. Typically, two instructions are employed in EasyTime++: **upd** and **dec**. The former update the value of variable in the database, whilst the latter decrements its value.

Although, DSLs can be implemented in vastly possible ways [1], an appropriate implementation when the end-users are not also the programmers is, a compiler/interpreter approach [31]. Hence, EasyTime was implemented using a compiler generator tool called LISA [8, 14]. The LISA specifications include lexical, syntax and semantic specifications. Whilst classical regular expressions and BNF are used for the first two specifications, the third specifications are based on Attribute Grammars [32]. One of the distinguishing features of a LISA compiler generator is that specifications (lexical, syntax, and semantics) can be easily reused and extended. An overall view of LISA specifications is given in Listing 1.

Listing 1. Overall view of LISA specifications

language L_1 [extends $L_2, \ldots, L_N$] {
lexicon {
[[P] **overrides** | [P] **extends**] R regular expr.
⋮
}
attributes type A1; :::;AM
⋮
rule [[Y] **extends** j [Y] **overrides**] Z {
X ::= X_{11} X_{12} … X_{1p} **compute** {
semantic functions }
⋮
|
X_{r1} X_{r2} … X_{rt} compute {
semantic functions }
;
}
⋮
method [[N] **overrides** j [N] **extends**] M {
operations on semantic domains
}
⋮
}

4. EasyTime++

EasyTime's formal description was introduced in [15], whilst the mapping of EasyTime's denotational semantics into attribute grammars, as well as its implementation, are presented in [16]. Due to requested extensions of EasyTime the language has evolved into EasyTime++. This section describes the formal specifications that were necessary for the change. Due to the space constraints, we are unable to include complete specifications. Interested readers are further referred to [15, 16].

The first small change was done within the semantic domain **Runners**, which represents a database of competitors (Listing 2). The additional components are now **Gender** and **Category**. Along with Id, RFID, $LastName$, and $FirstName$, the

$Gender$ and $Category$ regarding competitors have added to the semantic domain **Runners**. The second, and the most important change within the semantic domains is how the **State**, which is the mapping from variables to values, has been modelled (Listing 2). Within EasyTime, the **State** was a simple mapping: **State=Var → Integer**, however in EasyTime++ an initial value of an attribute depends on categories, and a variable, called 'dynamicvar', can also be initialized during a run-time. Hence, the State is now modelled as: **State**=**Var→((Category→Integer)**×**Truth-Value**).

Listing 2. Semantic domains in EasyTime++

Category={0, 1, 2, 3 ...}
Gender={$female$, $male$}
Runners=($Id \times RFID \times LastName \times FirstName \times Gender \times Category)^*$
State=Var→((Category→Integer) ×**Truth-Value**)

Let us describe the **State** using a simple excerpt from EasyTime++ declarations. Three variables were declared within an EasyTime++ program (Program 2). The first variable, $ROUND1$ specifies that all competitors need to complete 50 laps, hence in a database of competitors the attribute $ROUND1$ is set at 50 for all competitors. The second variable, $ROUND2$, specifies that, in a case where a competitor belongs to $category = 1$, he/she needs to complete 20 laps, whilst a competitor within $category = 2$ only needs 10 laps. In the database of competitors, the attribute $ROUND2$ is initialized according to the category. For all competitors in the first category this attribute will be initialized to 20, and for all competitors in the second category to 10. The third variable, $PENALTY$, is a dynamic variable and its initial value for each competitor will be set during the run-time.

Program 2. Excerpt from EasyTime++ declarations

```
1: var ROUND1 := 50;
2: var ROUND2 := {  (category==1)→20,
3:                  (category==2) →10 };
4: dynamicvar PENALTY; // definition of dynamic var.
```

The **State** in EasyTime++ is mapping which maps variable names (e.g., $ROUND1$, $ROUND2$, $PENALTY$) into two components. The first component is itself a mapping from **Category** to **Integer** (e.g., 12–>20, 2–>10), whilst the second component indicates whether a variable is dynamic or not. To cope with this new model for variables in EasyTime++, the following LISA methods are needed (note that the mapping from **Category** to **Integer** can be implemented using a hashtable [16], see Program 3).

Program 3. Implementation of EasyTime++ State in LISA

```
1: method M_Var
2:   class Var {
3:     String name;
4:     Hashtable values;
5:     boolean isDynamic;
6:     Var (String name, Hashtable values,
7:       boolean isDynamic) {
8:         this.name = name;
9:         this.values = values;
10:        this.isDynamic = isDynamic;
11:    }
12:    // Java methods are omitted
13:    …
14:  }
15: }
16:
17: method VarEnvironment {
18:  import java.util.*;
19:  public Hashtable put (Hashtable env, Var aVar) {
20:    env = (Hashtable)env.clone();
21:    env.put(aVar.getName(), aVar);
22:    return env;
23:  } // java method
24: } // Lisa method
```

Since all changes in EasyTime++ are done in a declaration part the semantic function D (for full description of EasyTime semantic functions please see [15, 16]), which describes the meanings of the declarations needs to be changed accordingly (Listing 3).

Semantic function $\mathcal{D}$ maps the syntactic construct **Dec**, representing the declarations, into its meaning **State→State**, which is a mapping from **State** to **State**. Note, how the first component of **State** is defined in a case where the categories are unspecified (first equation in Listing 3), and in a case of dynamic variables (third equation in Listing 3). In the first equation, it is stated that variable x is mapped to value a regardless of category. The mapping function $\lambda category.a$ is a constant function. The second equation states that variable x is mapped to different values (e.g., a_1, a_2) according to different categories (e.g., cat_1, cat_2), whilst in the third equation, the variable x is mapped to undefined value ⊥ regardless of category. In the case of dynamic variables the second component of (**Category → Integer**) ×**Truth-Value** is true, otherwise it is false.

Listing 3. Meaning of declarations in EasyTime++

$\mathcal{D}$: **Dec → State**	→	**State**
$\mathcal{D}$ ⟦**var**x := a⟧s	=	$s[x \to ((\lambda category.a) \times false)]$
$\mathcal{D}$⟦**var**$x := \{cat_1 \to a_1, cat_2 \to a_2\}$⟧$s$	=	$s[x \to (\{cat_1 \to a_1, cat_2 \to a_2\} \times false)]$
$\mathcal{D}$⟦**dynamicvar** x⟧s	=	$s[x \to ((\lambda category.\bot) \times true)]$
$\mathcal{D}[\![\mathcal{D}_1; \mathcal{D}_2]\!]s$	=	$\mathcal{D}[\![\mathcal{D}_2]\!](\mathcal{D}[\![\mathcal{D}_1]\!]s)$

The aforementioned changes in formal specifications of EasyTime++ also require changes in the implementation part. Note that changes are required in the lexical part (new keywords category and dynamicvar, new separator), syntax part (new syntax rules for declarations), as well as in the semantic part (new semantics for declarations). All the other parts of EasyTime (e.g., agents, measuring places, statements) [16] are intact and hence can be completely reused. Since EasyTime is implemented in LISA, which supports attribute grammar inheritance [8], and where lexical, syntax and semantic specifications can be inherited, it was natural to extend Easy-Time specifications written in LISA for implementing EasyTime++, thus achieving incremental language development. Program 4 shows the LISA specification of EasyTime++. Note, how all EasyTime specifications have been reused ('language EasyTime++ extends EasyTime'). In the inherited specifications it was necessary to override rule Dec, which contained syntactic and semantic specifications for declarations, add some new grammar productions and their semantics (rule $Categories$), as well as add new atribute $varvalues$ of type $Hashtable$, which were attached to the non-terminal $CTGRS$, and extend regular definitions for $Separator$ and $Keyword$. Overall less than 70 lines of LISA specifications have been newly written to obtain the complete compiler for EasyTime++. Note that this is an example of language extension where language specifications' fragment (Program 4) alone does not make any sense and can not exist without base-language specifications (for complete EasyTime specifications in LISA see [16]). Hence, this kind of language composition can be denoted as EasyTime ⊲ EasyTime++.

Program 4. LISA specification of EasyTime++

```
1: language EasyTime++ extends EasyTime {
2: lexicon {
3:   extends Separator,
4:   extends Keyword category | dynamicvar
5: }
6: attributes Hashtable*.varvalues;
7:   rule extends Start
8:     compute { }
9: }
10: rule overrides Dec {
11:   DEC::=var#Id:=#Int; compute {
12:     //category is not specified; isDynamic=false
13:     DEC.outState = put(DEC.inState,
14:      new Var(#Id.value(),
15:      put(new Hashtable(), "0",
16:      Integer.valueOf(#Int.value()).intValue()),false));
17:   };
18:   DEC ::= dynamicvar #Id ; compute {
19:     // category can not be specified; isDynamic=true
20:     DEC.outState = put(DEC.inState,
21:       new Var(Id.value(), null, true));
22:   };
23:   DEC ::= var #Id := { CTGRS } ; compute {
24:     // categories are specified and can't be dynamic
25:     DEC.outState = put(DEC.inState,
26:       new Var(#Id.value(), CTGRS.varvalues, false));
27:   };
28: }
29: rule Categories {
30:   CTGRS ::= ( category == #Int ) -> #Int , CTGRS
31:     compute {
32:       CTGRS[0].varvalues = put(CTGRS[1].varvalues,
33:        #Int[0].value(),
34:        Integer.valueOf(#Int[1].value()).intValue());
35:     };
36:     CTGRS ::= ( category == #Int ) -> #Int compute {
37:       CTGRS.varvalues = put(new Hashtable(),
38:        #Int[0].value(),
39:        Integer.valueOf(#Int[1].value()).intValue());
40:     };
41: }
42: ...
43:     // LISA methods
44: }
```

5. Examples

In order to test EasyTime++ DSL two casestudies were performed:

- cyclo-cross Grand-prix, and
- biathlon.

The former was experienced in practice, whilst the latter could be taken as proof of concept. In the rest of this section, both case-studies are discussed in detail.

5.1. Case-study 1: Cyclo-cross Grand-Prix

This case-study tested the introduction of categories in EasyTime++. Cyclo-cross is a relatively new sport that typically takes place in winter and is dedicated to cycle road-riders who are preparing for the new season. Races usually consist of several laps of a short course featuring pavements, wooded trails, grass, steep hills, and obstacles.

In this case-study, one lap of 2.5 km was used (Figure 2). According to the number of laps, the competitors were divided into three categories, as follows:

- 4 laps: junior men and women up to 15 years old (U-15),
- 6 laps: junior men and women up to 19 years old (U-19), and
- 9 laps: absolute categories (U-23, Elite, Masters).

In order to make the competition more interesting, the organizers allowed all the competitors onto the course simultaneously. Only one measuring device with two measuring places was needed for measuring this competition because the course passed at one location. Here, the intermediate times of laps were measured and, thereby, decremented the laps' counters of specific competitors. When the laps counter reached zero the finish time of the competitor was reported. However, how many laps to go depended on the category to which the specific competitor belonged.

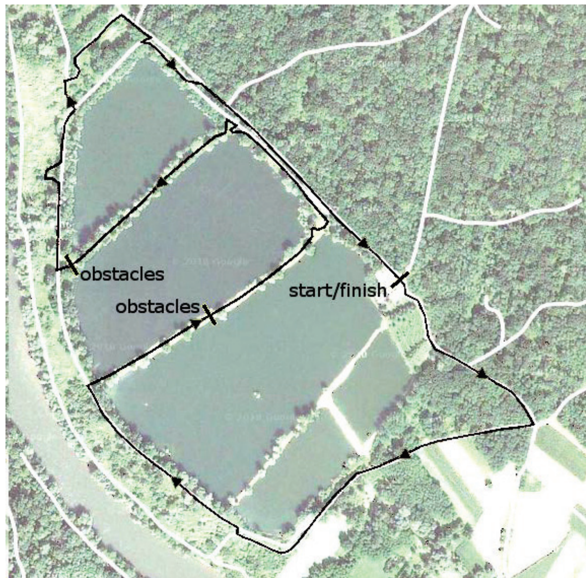

Figure 2. Track layout of the cyclo-cross competition

The EasyTime++ program for measuring time in this competition can be seen by Program 5. Note that here both measuring places, i.e., mats, were laid so that the whole length of the finish line was captured. In line with this, a competitor can cross either of both mats. As a result, the programs for both measuring devices are the same, and work in parallel.

Program 5. EasyTime++ program for measuring time in cyclo-cross competition

```
1: 2 auto 192.168.225.100; // definition of agent
2: var BIKE ≔ 0;
3: var ROUND1 ≔{(category ==1) → 4,
4:    (category == 2) → 6, (category ==3) → 9 };
5: // definition of measuring place 1
6: mp[1] → agnt[2] {
7:    (true) → dec ROUND1;
8:    (ROUND1 == 0) → upd BIKE;
9: }
10: // definition of measuring place 2
11: mp[2] → agnt[2] {
12:    (true) → dec ROUND1;
13:    (ROUND1 == 0) → upd BIKE;
14: }
```

In summary, measuring time in cyclo-cross performed well with EasyTime++. Although the organizers prepared three different lengths of courses, six different lists of results were obtained according to gender. Fortunately, in our case the gender could be handled by a database system, whilst the EasyTime++ program was unaware of it.

5.2. Case-study 2: Biathlon

A biathlon was the second case-study for EasyTime++. Biathlon refers specifically to the winter sport that combines cross-country skiing and rifle shooting. As can be seen from Figure 3, competitors start with skiing. Skiing is interrupted by rifle shooting. Typically, biathlon consists of 4 laps of skiing. The shooting appears close to the end of a lap. Two positions for competitors are allowed when shooting, i.e., prone and standing. Interestingly, the number of missed shoots is penalized by the additional number of penalty laps. Note that the time spent within the penalty laps are added to the total time of the competitor. The time of the penalty lap is typically taken to be between 20-30 seconds.

The EasyTime++ program for measuring time in biathlon can be seen by Program 6. Three measuring devices are needed to cover this competition. Each measuring device realizes one measuring place. Moreover, each measuring place also represents a control point. In contrast to Ironman, in a biathlon time spent in penalty loops is of no interest in the preferred race. Here, only the total time of competitor is important.

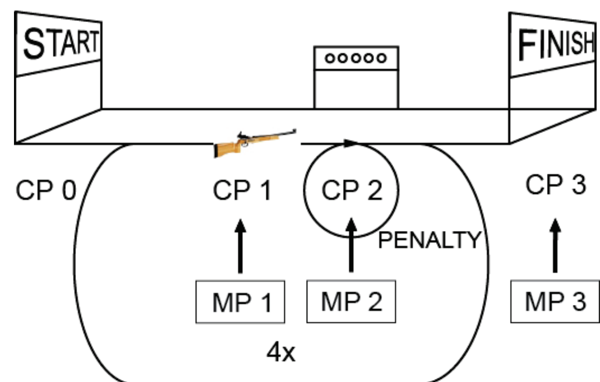

Figure 3. Measuring time in biathlon competitions

Program 6. EasyTime++ program for measuring time in biathlon competition

```
1: 1 auto 192.168.225.110; // definition of agent 1
2: 2 auto 192.168.225.100; // definition of agent 2
3: var ROUND ≔ 4;
4: var RUN ≔ 0;
5: dynamicvar PENALTY; // definition of dynam.variable
6: // definition of measuring place 1
7: mp[1] → agnt[1] {
8:    (true) → upd PENALTY;
9: }
10: // definition of measuring place 2
11: mp[2] → agnt[2] {
12:    (true) → dec PENALTY;
13: }
14: // definition of measuring place 3
15: mp[3] → agnt[2] {
16:    (true) → dec ROUND;
17:    (ROUND == 0) → upd RUN;
18: }
```

In summary, the first device represents the special measuring device for counting hits. The agent assigned to this device puts the number of missed hits into the database variable PENALTY, dynamically. Note that this device is treated in EasyTime like an ordinary measuring device. The second measuring device is dealt with by counting the penalty laps, whilst the third device measures the final time.

6. Conclusions

Easy language composition is still an open-issue within programming language research. In particular, a new young field of software language engineering is of interest regarding engineering principles when constructing new languages, whether general-purpose or domain-specific. A language designer would like to build a new language simply by composing different components and/or extending previous components. This paper has presented EasyTime++ DSL as a language extension of EasyTime, where the base language specifications written in the LISA compiler generator have been extended with new features, thus enabling the introduction of categories into competitions, and those new competitions where the number of laps is dynamically determined. The implemented multiple attribute grammar inheritance in LISA enables easy language composition since lexical, syntax, and semantic specifications can be reused and extended. In such a manner, an incremental language development using LISA has been demonstrated. The suitability of EasyTime++ was shown in two case studies: cyclo-cross Grand-prix and a biathlon. More extensive experimental work, which would include other types of language compositions and more DSLs, is also planned in the future.

References

[1] **M. Mernik, J. Heering, A. M. Sloane.** When and how to develop domain-specific languages. IN: *ACM Computing Surveys*, 2005, Vol. 37, No. 4, pp. 316-344.

[2] **A. van Deursen, P. Klint, J. Visser.** Domain-specific languages: an annotated bibliography. In: *ACM SIGPLAN Notices*, 2000, Vol. 35, No. 6, pp. 26–36.

[3] **P. Hudak.** Building domain-specific embedded languages. In: *ACM Computing Surveys*, 1996, 28(4es).

[4] **M. Fowler.** Domain Specific Languages. In: *Addison-Wesley Professional*, 2010.

[5] **M. J. Varanda Pereira, M. Mernik, D. da Cruz, P. R. Henriques.** Program comprehension for domainspecific languages. In: *Computer Science and Information Systems*, 2008, Vol. 5, No. 2, pp. 1–17.

[6] **T. Kosar, P. R. Henriques, M. J. Varanda Pereira, M. Črepinšek, M. Mernik, N. Oliveira, D. da Cruz,** Comparing General-Purpose and Domain-Specific Languages: An Empirical Study. In: *Computer Science and Information Systems*, 2010, Vol. 7, No. 2, pp. 247-264.

[7] **T. Kosar, M. Mernik, J. C. Carver.** Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. In: *Empiricalsoftware engineering*, 2012, Vol. 17, No. 3, pp. 276–304.

[8] **M. Mernik, V. Žumer**. Incremental programming language development. In: *Computer Languages, Systems and Structures*, 2005, Vol. 31, No. 1, pp. 1-16.

[9] **D. Hrnčič, M. Mernik, B. R. Bryant.** Embedding DSLs into GPLs: A Grammatical Inference Approach. In: *Information Technology and Control*, 2011, Vol. 40, No. 4, pp. 307-315.

[10] **D. Hrnčič, M. Mernik, B. R. Bryant, F. Javed.** A memetic grammar inference algorithm for language learning. In: *Applied Soft Computing*, 2012, Vol. 12, No. 3, pp. 1006-1020.

[11] **G. Kiczales.** Aspect-oriented programming. In: *ACM Computing Surveys*, 1996, Vol. 28, No. 4, Article No. 154.

[12] **P. Klint, T. van der Storm, J. J. Vinju.** Term rewriting meets aspect-oriented programming. In: *Processes, Terms and Cycles: Steps on the Road to Infinity,* Springer-Verlag, 2005, pp. 88–105.

[13] **M. Mernik, D. Rebernak.** Aspect-Oriented Attribute Grammars. In: *Electronics and Electrical Engineering*, 2011, Vol. 10, No. 116, pp. 99–104.

[14] **P. R. Henriques, M. J. Varanda Pereira, M. Lenič, M. Mernik, J. Gray, H. Wu.** Automatic generation of language-based tools using LISA. In: *IEE Proceedings-Software Engineering*, 2005, Vol. 152, No. 2, pp. 54-69.

[15] **I. Jr. Fister, I. Fister, M. Mernik, J. Brest.** Design and implementation of domain-specific language Easytime. In: *Computer Languages, Systems and Structures*, 2011, Vol. 37, No. 4, pp. 276-304.

[16] **I. Jr. Fister, M. Mernik, I. Fister, D. Hrnčič.** Implementation of EasyTime Formal Semantics using a LISA Compiler Generator. In: *Computer Science and Information Systems*, 2012, Vol. 9, No. 3, pp. 1019-1044.

[17] **S. Erdweg, P. G. Giarrusso, T. Rendel.** Language Composition Untangled. In: *Proceedings of Workshop on Language Descriptions, Tools and Applications (LDTA'12).* Available at http://www.informatik.unimarburg.de/~seba/publications/languagecomposition.pdf, 2012.

[18] **A. Granicz, J. Hickey.** Phobos: A front-end approach to extensible compilers. In: *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS36)*, 2003.

[19] **G. Hedin, E. Magnusson.** JastAdd: an aspectoriented compiler construction system. In: *Science of Computer Programming*, 2003, Vol. 47, No. 1, pp. 37–58.

[20] **E. Van Wyk, D. Bodin, J. Gao, L. Krishnan.** Silver: An extensible attribute grammar system. In: *Science of Computer Programming*, 2010, Vol. 75, No. 1, pp. 39-54.

[21] **T. Clark, P. Sammut, J. Willans.** Superlanguages: Developing languages and applications with XMF. Published online at: *hhtp://bit.ly/HiTOKp*, 2008.

[22] **J. Cervelle, R. Forax, G. Roussel.** A simple implementation of grammar libraries. In: *Computer Science and Information Systems*, 2007, Vol. 4, No. 2, pp. 65–77.

[23] **H. Krahn, B. Rumpe, S. Voelkel.** MontiCore: Modular development of textual domain specific languages. In: *Proceedings of the 30th International Conference on Software Engineering (SLE 2008)*, 2008, pp. 925–926.

[24] **A. Stonis, D. Rubliauskas, J. Blonskis.** Mapping Syntax Extensions. In: *Information Technology and Control*, 2002, Vol. 24, No. 3, pp. 35-48.

[25] **J. Porubän, M. Forgáč, M. Sabo, M. Bìhálek.** Annotation Based Parser Generator. In: *Computer Science and Information Systems*, 2010, Vol. 7, No. 2, pp. 291-307.

[26] **M. Viera, S. D. Swierstra, A. Middelkoop.** UUAG Meets AspectAG: How to make Attribute

Grammars First-Class. In: *Proceedings of Workshop on Language Descriptions, Tools and Applications (LDTA'12)*. Available at http://www.cs.uu.nl/research/techreps/repo/CS-2011/2011-029.pdf, 2012.

[27] **D. Brown, J. Levine, T. Mason.** Lex & yacc. *O'Reilly Media, 2nd Edition*, 1992.

[28] **X. Wu, B. R. Bryant, J. Gray, M. Mernik.** Component-based LR parsing. In: *Computer Languages, Systems and Structures*, 2010, Vol. 36, No. 1, pp. 16-33.

[29] **V. Štuikys, R. Damaševicius.** Measuring Complexity of Domain Models Represented by Feature Diagrams. In: *Information Technology and Control*, 2009, Vol. 38, No. 3, pp. 179-187.

[30] **K. Finkenzeller.** RFID Handbook. *John Wiley & Sons*, 2010.

[31] **T. Kosar, P. E. Martínez López, P. A. Barrientos, M. Mernik.** A Preliminary Study on Various Implementation Approaches of Domain-Specific Language. In: *Information and Software Technology*, 2008, Vol. 50, No. 5, pp. 390-405.

[32] **J. Paakki.** Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. ACM Computing Surveys, 1995, Vol. 27, No. 2, pp. 196-255.

Received June 2012.