Computer Languages, Systems & Structures 37 (2011) 151–167

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Design and implementation of domain-specific language easytime

Iztok Fister Jr., Iztok Fister *, Marjan Mernik, Janez Brest

University of Maribor, Faculty of Electrical Engineering and Computer Science, Smetanova 17, 2000 Maribor, Slovenia

ARTICLE INFO

Article history:
Received 16 January 2011
Received in revised form
23 March 2011
Accepted 25 April 2011
Available online 10 May 2011

Keywords:
Domain-specific language
Timing system
RFID technology

ABSTRACT

Measuring time in mass sporting competitions is, typically, performed with a timing system that consists of a measuring technology and a computer system. The first is dedicated to tracking events that are triggered by competitors and registered by measuring devices (primarily based on RFID technology). The latter enables the processing of these events. In this paper, the processing of events is performed by an agent that is controlled by the domain-specific language, EasyTime. EasyTime improves the flexibility of the timing system because it supports the measuring of time in various sporting competitions, their quick adaptation to the demands of new sporting competitions and a reduction in the number of measuring devices. Essentially, we are focused on the development of a domain specific language. In practice, we made two case studies of using EasyTime by measuring time in two different sporting competitions. The use of EasyTime showed that it can be useful for sports clubs and competition organizers by aiding in the results of smaller sporting competitions, while in larger sporting competitions it could simplify the configuration of the timing system.

1. Introduction

Domain-specific languages (DSLs) are tailored to application domains and have definite advantages over general-purpose languages in a specific domain. These advantages are especially clear in their greater expressive power and, therefore, higher productivity, as well as their ease of use, easier verification, and optimization [4,5,11,12,16,19]. In this paper, we have developed the domain-specific language EasyTime for measuring time in sporting competitions.

In the past, measuring time in sporting competitions was performed by timekeepers that measured time manually. The times from a timer were assigned to the starting number of competitors and were then sorted according to the final results and categories. When RFID (Radio Frequency Identification) technology appeared, the cost of measuring technology decreased [3,28] and became accessible to a wider range of users (e.g., sporting clubs, organizers of sporting competitions, etc.). At the same time, these users began to compete with the established monopolies [30] by measuring time in smaller sporting competitions.

To automatically monitor results in sporting competitions, a timing system consisting of a measuring technology and a computer system is necessary. The measuring technology is capable of event tracking and is typically based on RFID technology. A competitor triggers an event with an RFID tag when they cross over an antenna field that is covered by a measuring device. This tag is normally strapped to the leg of the competitor. The computer system enables the processing of the results into a database, which is then sorted according to the final results and then printed. However, the main question remains how to connect the measuring technology and the computer system into the integrated timing system.

* Corresponding author. Tel.: +386 2 220 7401.
E-mail address: iztok.fister@uni-mb.si (I. Fister Jr.).

1477-8424/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cl.2011.04.001

For this functionality, we have proposed an agent that runs on a database server and enables a connection with the measuring device, obtaining events, assigning these to the results of the competitors and recording them into a database. Moreover, this agent is controlled by EasyTime, which can additionally improve the flexibility of the timing system. This flexibility means that the timing system can be used by measuring many kinds of sporting competitions, quick and simple configuration for any kinds of sporting competitions, and a reduction in the number of measuring devices needed.

The problem that we have solved is not new. Many specialized companies provide reliable and secure solutions for measuring time in sporting competitions. Unfortunately, these are usually very expensive, primarily because of costly measuring technology. Indeed, because of their monopoly position on the market, there are few papers that treat this topic in existing literature. However, many parts of these solutions have been published in the literature that deals with RFID technology [7,9].

Essentially, we focused on the development of the domain-specific language EasyTime. This development was divided into three phases: domain analysis – the definition of EasyTime concepts and terminology, DSL design – the definition of EasyTime language specifications (syntax and semantics), and DSL implementation – building the EasyTime compiler using the LISA tool [10,17,20]. LISA automatically generates a compiler from formal attribute grammar-based language specifications. While many DSLs have been developed in the past [19] only a few of them have been developed rigorously with a formal semantics approach. Notable examples are found in [2,18,29]. The aim of this paper is also to promote a disciplined engineering approach to the development of DSLs, where all phases are clearly identified and procedures formalized. Finally, the timing system based on EasyTime was tested in real competitions. Two case studies will be presented to illustrate how EasyTime behaves in the real-world. In the first case study, we dealt with a time trial for a bicycle competition, while for the second case study a triathlon competition was chosen, as it is one of the most challenging events to measure. In fact, tackling such a competition involves measuring three different disciplines, and stretches out over a significant period of time.

The structure of the rest of the paper is as follows: In Section 2, problems that arise in the measuring of sporting competitions are discussed. In line with this, we have focused on triathlon competitions, which consist of three disciplines and are, subsequently, very difficult to perform. In Section 3, the development of EasyTime is presented. In Section 4, practical experiences with EasyTime are presented. The paper is concluded with a short analysis of the performed work and suggestions for future work.

2. Measuring time in sporting competitions

In practice, time in sporting competitions can be measured *manually* (with a classical or computer timer) or *automatically* (with a measuring device). The computer timer is a computer program that was developed to meet the needs of manually measuring time. For measuring time, it exploits a processor tact. The processor tact is the speed, with which the processor executes computer instructions. The computer timer is capable of generating events—similar to a measuring device. However, in this case, the event is triggered by an operator pressing the suitable key on the keyboard, while the measuring device detects the event automatically. Generated events are in triplets $Ev = \langle \#, MP, TIME \rangle$, where # denotes the starting number of the competitor, *MP* is the measuring place where the event happened and *TIME* is the timestamp registered by the computer at the moment when the competitor crossed the measuring place and represents the number of seconds since 1.1.1970 at 0:0:0. However, the starting number of the competitor remains undefined at the time that the event is registered and needs to be entered at a later stage. For reliable event tracking, two operators are needed: the first for events generating and the second for manually recording the competitors' starting numbers according to the order in which they cross the measuring place. Events that consist of pairs $\langle Ev, \# \rangle$ need to be recorded to a database via file transfer protocol [8].

Measuring devices today are typically based on RFID technology [7] in which an identification is performed with an electromagnetic wave motion within the range of a radio frequency. This technology is not new and consists of the following elements:

- RFID tags that keep identification numbers and
- a reader of the RFID tags.

The main characteristics of this technology are

- no contact between tag and reader is needed for recognizing RFID tags,
- the identification number in tags can be modified.

Active and the passive RFID technologies are the most commonly used. For active RFID technology, no electrical power is needed. Furthermore, the tags can be read from large distances (more than 100 m) and they can keep more information than passive RFID technology. However, the main weakness of this technology is the use of batteries that need to be regularly charged. In this way, the technology is more expensive as well.

Passive RFID technology works by inducing the electrical power on the RFID tag. This causes the transmission of the identification number to the receiving antenna of the measuring device. In addition, the timestamp of the event registration is recorded by the measuring device. On the other hand, the passive RFID tag does not use its own power supply. The primary weakness of this technology is that the reading ability decreases in conjunction with the distance of the RFID tag from the reader.

The measuring device consists of an RFID tags reader, processor, primary storage, liquid crystal display (LCD) and a numerical keyboard. All of these elements are enclosed in a waterproof casing and secured from mechanical damage. More antenna fields can be connected to the measuring device. Furthermore, it can be connected to the local area network (LAN) with an ethernet adapter. The measuring device can be controlled via keyboard and LCD display. Typically, the main control functions that are available in the measuring device are

- read of real time,
- setup of real time,
- start of event registration,
- end of event registration,
- read of events online,
- read of events offline.

The measuring device connected to the LAN can also be controlled via a control program running on a workstation. This program accesses the device via TCP/IP sockets with a suitable protocol. Usually, the measuring device supports a Telnet protocol that is easy to implement. On the other hand, the commands can also be transferred to the device as a text stream.

In the timing system, each antenna field represents a *measuring place* (MP). In fact, the measuring place represents the special antenna in the mat. An event is triggered by the competitor crossing over the mat with a passive RFID tag. The event is a quadruplet $Ev2 = \langle \#, RFID, MP, TIME \rangle$, in which the previously mentioned triplet gets an additional RFID tag. Typically, the courses in sporting competitions are divided into multiple *control points* (CP), where the organizers need to track the measured time. This time can be *intermediate* or *final*. The location of the control points depends on the kind of competition and the configuration of the course in which the competition takes place.

2.1. An example: measuring time in triathlon

Triathlon competitions demand special attention because they encompass three disciplines in one competition. In this paper, we will focus on this problem.

The triathlon is a relatively new sporting discipline as the first triathlon competition was performed in the USA in 1975. Today, it is an olympic discipline in which competitors first swim, then ride a bicycle, and in the end, run. All three disciplines occur consecutively and continuously. In the summary time, the times of both transitions are added, i.e. the time when the competitor goes from swimming to bicycling and, then, the transition time from bicycling to running. Professional and amateur competitors compete together in various age categories. Typically, there are many kinds of triathlon competitions, such as

- sprint (750 m swimming, 20 km bicycling and 5 km running),
- olympic or short (1500 m swimming, 40 km bicycling and 10 km running),
- half IRONMAN (1900 m swimming, 90 km bicycling and 21 km running),
- full IRONMAN (3800 m swimming, 180 km bicycling and 42 km running),
- ultra triathlon (n-times IRONMAN, where $n=2$, 3, 4, 5, 10 and 20, respectively).

The measuring time in the triathlon is divided into nine control points. In Fig. 1, the ultra double triathlon is illustrated (7.6 km of swimming, 360 km on a bicycle and 84 km of running), where the swimming course is divided into 20 laps, the bicycling course into 105 laps and the running course into 55 laps. The summary time of a competitor consists of five final times (the swimming time SWIM (CP2), the first transition time TA1 (CP3), the bicycling time BIKE (CP5), the second transition time TA2 (CP6) and the running time (CP8)) as well as three intermediate times (the intermediate swimming time (CP1), the intermediate bicycling time (CP4) and the intermediate running time (CP7)). By intermediate times, the number of laps ROUND_x as well as the achieved result INTER_x is measured. Here, $x=1$, 2, 3 denotes a particular discipline.

Suppose that for measuring time in a triathlon competition in Fig. 1 one measuring device with two measuring places (MP3 and MP4) is available and that the competition is performed at one location. In this case, the last crossing over MP3 can represent the CP5 time, the first crossing over MP4 the CP6 time, and the last crossing over MP4 the final result (CP8). The measuring places MP1 and MP2 can be measured manually. In line with this, the number of control points can be reduced by three if the control points are correctly located and the timing system is used. Therefore, 162 events per competitor (or 87.6%) can be measured with one measuring device. When we consider that the measuring technology for

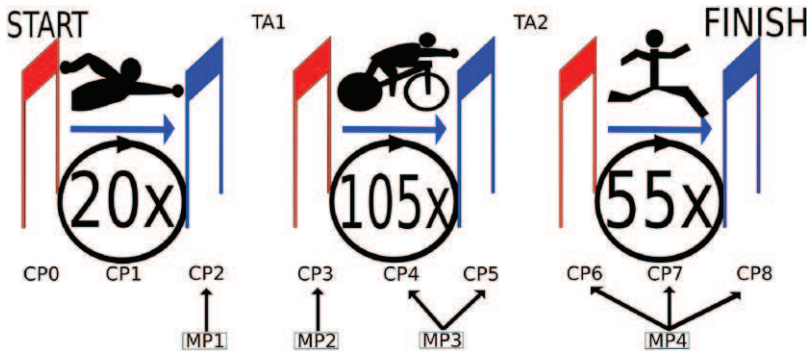

Fig. 1. Definition of control points in a triathlon competition.

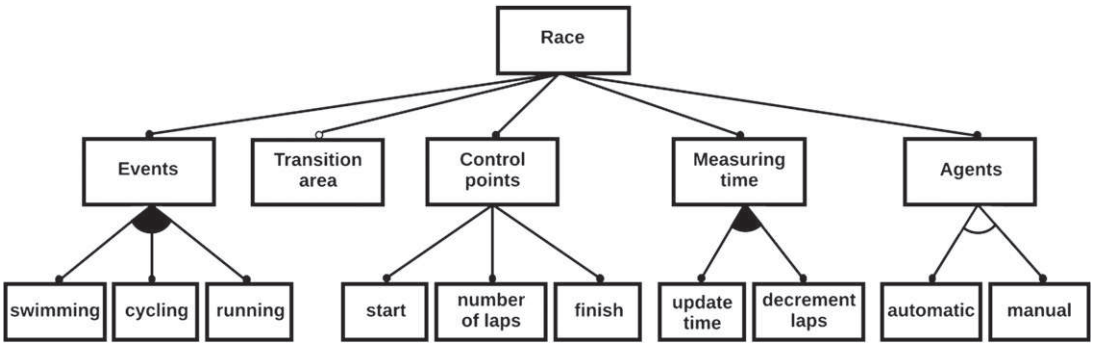

Fig. 2. The feature diagram of EasyTime.

swimming in lakes and seas remains expensive and that swimming is measured by referees manually, in this competition 98% of all events can be measured.

3. The domain-specific language EasyTime

3.1. Domain analysis

A prerequisite for the design of a DSL is the detailed analysis and structuring of the application domain [6]. The analysis of the application domain is provided by a *domain analysis*. The results of the domain analysis are obtained in a *feature model* [26]. A key element of the feature model is a *feature diagram* (FD) that, in a graphical way, describes the dependency between features. The FD is represented as a tree with nodes as rectangles and arcs connecting the nodes. Nodes determine the features, while arcs determine the relationships between them. The nodes can be *mandatory* or *optional*. The first is denoted by closed dots, whilst the latter is denoted by open dots. The sample FD of EasyTime is illustrated in Fig. 2.

From the FD in Fig. 2, it can be seen that the time measurements for the concept *Race* consists of *Events*, *Transition area*, *Control points*, *Measuring places* and *Agents*. All sub-features except for *Transition area* (as denoted by an open dot on arc) are mandatory because they are connected by the relation *all* (no semicircle joins the arcs from feature to its sub-features). The *Events* may either be *swimming*, *cycling* or *running* or any combination of these three sub-features as indicated by the closed semicircle denoting the relation *more-of*. The *Control points* consist of three mandatory sub-features: *start*, *number of laps* and *finish* that are connected by the relation *and*. With the *Measuring places* the time may be updated (*update time*) and/or laps may be decremented (*decrement laps*). Therefore, these sub-features are connected by the relation *more-of*. The *Agents* may be *automatic* or *manual*, but not both. The open semicircle indicates a *one-of* relation. Finally, the number of possible race instances is 84: $7(Events) \times 2(Transition\ area) \times 1(Control\ points) \times 3(Measuring\ places) \times 2(Agents)$.

The FDs reveal important concepts and their structures (in the sense of how the concept can be broken down into features and sub-features). In our case, a competition (or race) consists of events (swimming, cycling, running, etc.). Each event has a start and a finish line and at least one lap. This concept is embodied in the control point concept. Between events, transition areas are placed. At each control point, the time is measured and/or the lap is decremented by a measuring place. Measuring places are controlled by agents, which can be manual or automatic. The manual control of agents means that events are measured with manual timers, while the automatic control of these events is captured by measuring devices. In fact, events are assembled in files and batch processed by the manual agent. In line with automatic control, events are generated stochastically by the measuring devices and processed online. In both cases, the agent needs

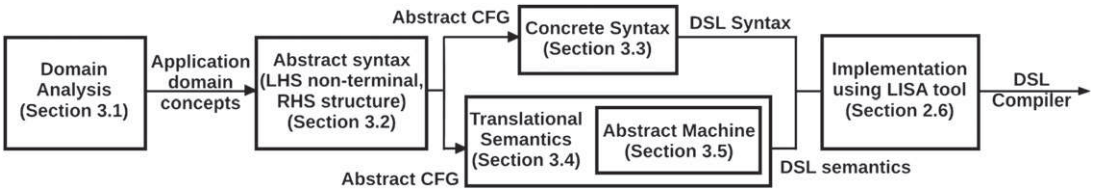

Fig. 3. An overview diagram of EasyTime development.

Table 1
Translation of the application domain concepts to a context-free grammar.

Application domain concepts	LHS non-terminal	RHS structure
Race	P	Description of agents; control points; measuring places
Events (swimming, cycling, running)	None	Measuring time is independent from the type of an event. However, good attribute's identifier in control points description will resemble the type of an event
Transition area times	None	Can be computed as difference between events final and starting times
Control points (start, number of laps, finish)	D	Description of attributes where start and finish time will be stored as well as remaining laps
Measuring places (update time, decrement lap)	M	Measuring place id; agent id, which will control this measuring place; specific actions which will be performed at this measuring place (e.g., decrement lap)
Agents (automatic, manual)	A	Agent id; agent type (automatic, manual); agent source (file, ip)

to wait either on the batch of events assembled into files or on the stochastically generated events obtained from the measuring device that is accessible through the local area network with an appropriate IP address.

On the other hand, FD also reveals commonalities (common features which always exist in a system) and variabilities (optional features which may or may not exist in a system). The latter are very important in the next phase of DSL development, since the list of variations indicates precisely what information is required to specify an instance of a system; this information must be directly specified in or be derivable from DSL programs, while commonalities should be built into a DSL execution environment through a set of common operations and primitives (e.g., types) of a language. From FD the variation points can be easily identified (optional, one-of and more-of features). Overall, during domain analysis, the following information is usually gathered: terminology, concepts, and common and variable properties of concepts and their interdependencies. Although this is extremely useful information, further steps in DSL development are not at all obvious. In the next step, the design phase of DSL development syntax and semantics are defined formally or informally. In an informal design, the specification is usually in some form of natural language, often containing a set of illustrative DSL programs, while a formal design generally consists of a specification written in some of the available formal definition methods (e.g., regular expressions and grammars for syntax specifications, and attribute grammars, denotational semantics, operational semantics, and abstract state machines for semantic specifications). Hence, designing a language involves defining the constructs in the language and providing the semantics, preferably formal, to the language.

An overview diagram that links specific steps to sections is presented in Fig. 3.

3.2. The abstract syntax

During the domain analysis we identified several concepts in the application domain that needed to be mapped into DSL syntax and semantics. Here, we can notice correspondence between concepts/features in an application domain and non-terminals in a context-free grammar (CFG). First, at a higher abstraction level, non-terminal symbols are used to describe different concepts in the programming language (e.g., an expression or a declaration in a general-purpose programming language, or an agent in our EasyTime DSL). On the other hand, at a more concrete level, non-terminal and terminal symbols are used to describe the structure of a concept (e.g., an expression consists of two operands separated by an operator symbol, or an agent specification in EasyTime consists of the agent's number identification, the type of the agent and the source of the events). Therefore, both the concepts and the relations between them, belonging to the specific problem domain, can be captured in a context-free grammar. Table 1 represents the mapping between application domain concepts and non-terminals in context-free grammars, which appears on the left hand side (LHS) of CFG production. The non-terminals' structure which appears on the right-hand side (RHS) of CFG productions is also shown in Table 1.

The abstract syntax of EasyTime, which is based on Table 1, is presented in Table 2, whereas the syntactic domains of variables are presented in Table 3. An EasyTime program P consists of the agents declaration A, attribute declarations D, and specification of measuring places M. The agent declaration A specifies the agent's number identification n; the type of the agent (manual or auto) and the source of the events (file or IP address). An EasyTime program might have many agent declarations. The declaration D specifies the attributes of a database that will be created as well as the initial values of

Table 2
The abstract syntax of EasyTime.

$P ::= A\ D\ M$
$A ::= n$ **manual** *file* $\mid n$ **auto** $ip \mid A_1; A_2$
$D ::=$ **var** $x := a \mid D_1; D_2$
$M ::=$ **mp**$[n_1] \rightarrow$ **agnt**$[n_2]\ S \mid M_1; M_2$
$S ::=$ **dec** $x \mid$ **upd** $x \mid x := a \mid (b) \rightarrow S \mid S_1; S_2$
$b ::=$ **true** $\mid$ **false** $\mid a_1 = a_2 \mid a_1 != a_2$
$a ::= n \mid x$

Table 3
Syntactic domains.

$P \in$ **Pgm**	$A \in$ **Adec**
$D \in$ **Dec**	$M \in$ **MeasPlace**
$S \in$ **Stm**	$b \in$ **Bexp**
$a \in$ **Aexp**	$n \in$ **Num**
$x \in$ **Var**	*file* $\in$ **FileSpec**
$ip \in$ **IpAddress**	

those attributes for each runner. Note that runners are not specified in an EasyTime program. However, to generate a code and a database, the runners will be provided during the compilation process. The measuring places M specify the identification numbers of the measuring place n_1, the agent's identification number n_2, and the statements S which are going to be executed at measuring place n_1 and will be under the control of agent n_2. An EasyTime program might specify many measuring places. Among the statements S we can identify simple statements such as the decrement attribute x (**dec** x), update attribute x (**upd** x), and assignment statement ($x := a$), as well as the conditional statement ($(b) \rightarrow S$) and compound statement ($S_1; S_2$). The arithmetic expression a can be a number (n) or an attribute (x). The boolean expression b can be literals **true** and **false**, or a compound expression using either the operator equal ($=$) or the operator not equal ($!=$).

3.3. The concrete syntax

After the abstract syntax is defined, the next step is to define the meaning of language constructs. In other words, the language semantics. In parallel, a language designer often experiments with various forms of concrete syntaxes to see how various constructs might look. For example, the agent's description, one manual and another automatic, might be described using concrete syntax such as (Algorithm 1):

Algorithm 1. EasyTime agents description.

```
1:  1 manual "abc.res";
2:  2 auto 192.168.225.100;
```

The EasyTime program consists of a description of agents, attributes where the results of control points are stored or laps decremented, and a description of measuring points. After the agent's description, the attributes are defined. These descriptions using concrete syntax might be as follows (Algorithm 2):

Algorithm 2. EasyTime attributes description.

```
1:  var ROUND2 := 105;
2:  var INTER2 := 0;
3:  var BIKE := 0;
```

A measuring place is marked with an identification number, the agent's id, which controls this measuring place, and the actions that will be executed at this measuring place. Again, a concrete example of such a description, where measuring place 3 is controlled by agent 2, is shown in Algorithm 3. Note that each time a competitor crosses this measuring place the following actions are executed: updating the attribute INTER2, decrementing a lap represented by the attribute ROUND2, and updating the attribute BIKE (final time of bicycling) if the attribute ROUND2 is zero.

Algorithm 3. EasyTime measuring place description.

```
1:  mp[3]→agnt[2]{
2:     (true)→upd INTER2;
3:     (true)→dec ROUND2;
4:     (ROUND2 = = 0)→upd BIKE;
5:  }
```

When a language designer is satisfied with the look and feel of the language's syntax, and possible additional constraints from domain experts or language end-users are fulfilled, the concrete syntax can be finalized. This process can be executed in parallel with defining language semantics. In Table 4, the EasyTime concrete syntax is given. In Algorithm 4, a complete example of the EasyTime program for measuring time in a triathlon is also presented.

Algorithm 4. EasyTime program for measuring time in a triathlon.

```
1:  1 manual "abc.res";
2:  2 auto 192.168.225.100;
3:
4:  var ROUND1 := 20;
5:  var INTER1 := 0;
6:  var SWIM := 0;
7:  var TRANS1 := 0;
8:  var ROUND2 := 105;
9:  var INTER2 := 0;
10: var BIKE := 0;
11: var TRANS2 := 0;
12: var ROUND3 := 55;
13: var INTER3 := 0;
14: var RUN := 0;
15:
16: mp[1]→agnt[1]{
17:   (true)→upd SWIM;
18:   (true)→dec ROUND1;
19: }
20: mp[2]→agnt[1]{
21:   (true)→upd TRANS1;
22: }
23: mp[3]→agnt[2]{
24:   (true)→upd INTER2;
25:   (true)→dec ROUND2;
26:   (ROUND2 = = 0)→upd BIKE;
27: }
28: mp[4]→agnt[2]{
29:   (true)→upd INTER3;
30:   (ROUND3 = = 55)→upd TRANS2;
31:   (true)→dec ROUND3;
32:   (ROUND3 = = 0)→upd RUN;
33: }
```

3.4. Formal semantics

The translation of an EasyTime program into a code that is going to be executed on several abstract machines (AM), described in section 3.5, is given through semantic translation functions (e.g., $\mathcal{CP}$,$\mathcal{CM}$). Those semantic translation functions employ several semantic domains, which are presented in Table 5. Among classical semantic domains [21] such as sets **Integer**, **Truth-Value**, and the function **State**, we are also using mathematical entities that represent agents (**Agents**), runners (**Runners**), and a database (**DataBase**). The function **Agents** will map the agent's identification number to the agent's type (manual or auto) and the agent's source (file or IP address). **Runners** is a database that contains the runner's data (identification number, RFID, last and first name). Note that the attributes of this database are fixed. On the other hand, **DataBase** is a database where the runner's results will be stored. The structure of this database is determined by the EasyTime program.

Table 4
The concrete syntax of EasyTime.

PROGRAM ::= AGENTS DECS MES_PLACES
AGENTS ::= AGENTS AGENT | ε
AGENT ::= #Int auto #ip; |#Int manual #file;
DECS ::= DECS DEC | ε
DEC ::= var #Id := #Int;
MES_PLACES ::= MES_PLACE MES_PLACE | MES_PLACE
MES_PLACE ::= mp[#Int]− > agnt[#Int]{STMTS}
STMTS ::= STMT STMTS | STMT
STMT ::= dec #Id; | upd #Id; | #Id := EXPR; | (LEXPR)− > STMT
LEXPR ::= true | false | EXPR = = EXPR | EXPR! = EXPR
EXPR ::= #Int | #Id

Table 5
Semantic domains.

Integer = {...−3,−2,−1,0,1,2,3...}	$n \in$ **Integer**
Truth-Value = {*true*,*false*}	
State = **Var** → **Integer**	$s \in$ **State**
AType = {*manual*,*auto*}	
Agents = **Integer** → **AType** × (*FileSpec* ∪ *IpAddress*)	$ag \in$ **Agents**
Runners = (*Id* × *RFID* × *LastName* × *FirstName*)*	$r \in$ **Runners**
DataBase = ($Id \times Var_1 \times Var_2 \times \cdots \times Var_n$)*	$db \in$ **DataBase**

Table 6
Translation of the program.

$\mathcal{CP}$: **Pgm** → **Runners** → **Code** × **Integer** × **DataBase**
$\mathcal{CP}[\![A\ D\ M]\!]r =$ let $s = \mathcal{D}[\![D]\!]\emptyset$:
$db =$ create &insertDB(s,r)
in $(\mathcal{CM}[\![M]\!](\mathcal{A}[\![A]\!]\emptyset), db)$

Table 7
Translation of measuring places.

$\mathcal{CM}$: **MeasPlace** → **Agents** → **Code** × **Integer**
$\mathcal{CM}[\![\mathbf{mp}[n_1] \rightarrow \mathbf{agnt}[n_2]S]\!]ag = (\text{WAIT } i : \mathcal{CS}[\![S]\!](ag, n_2), n_1)$
$\mathcal{CM}[\![M_1; M_2]\!]ag = \mathcal{CM}[\![M_1]\!]ag : \mathcal{CM}[\![M_2]\!]ag$

Table 8
Meaning of declarations.

$\mathcal{D}$: **Dec** → **State** → **State**
$\mathcal{D}[\![\mathbf{var}\ x := a]\!]s = s[x \rightarrow a]$
$\mathcal{D}[\![D_1, D_2]\!]s = \mathcal{D}[\![D_2]\!](\mathcal{D}[\![D_1]\!]s)$

The translation of an EasyTime program into an AM code is specified in Table 6. The translation function $\mathcal{CP}$ takes two inputs: $p \in$ **Pgm** and $r \in$ **Runners**. The result is a triplet: a code $c \in$ **Code** that is going to be executed on a particular AM with the identification number $n \in$ **Integer**, as well as the database $db \in$ **DataBase**, where the runner's results will be stored. The code and the AM's identification number are obtained by applying the translation function $\mathcal{CM}$ on the measuring places *M*. Meanwhile, the database *db* is obtained from attributes specified in the declaration part *D* and from $r \in$ **Runners**.

The translation of measuring places $M \in$ **MeasPlace** produces code and the AM's identification number (Table 7). There are two different syntactic constructs for *M*: the definition of measuring place n_1 and an agent n_2 that controls this measuring place with the corresponding statements *S*, as well as a sequence of measuring places $M_1; M_2$. In the former case, the translation function $\mathcal{CM}$ first generates the WAIT instruction and then call the translation function $\mathcal{CS}$ over statements *S*. The instruction WAIT *i* waits for an event by some of the competitors *i*. When the event is registered on the measuring place it is sent to the appropriate AM *j*. Note that the received event is a triplet $\langle i, RFID, TIME \rangle$. The parameter *i* identifies the competitor by the starting number but it can be zero when the event is registered on the measuring device. In that case, the correct starting number of the competitors is obtained by looking up of the database via the *RFID* parameter. Furthermore, the translation function $\mathcal{CM}$ is applied over M_1 and M_2.

Declarations are not directly translated into code. The meaning of the declaration is updating the **State**, which is a function from attributes to values (Table 8). We simply store attributes and their initial values into this semantic entity. These are needed to create and initialize a database for storing a runner's results.

Similarly, an agent's declaration is also not translated into code. The purpose of the agent's declaration is updating the **Agents**, which is a mapping from the agent's identification number into the agent's type and the agent's source (file or IP address). The meaning of the agent's declaration **Adec** is given in Table 9.

The translation of the statements **Stm** into AM code is specified in Table 10. Since the update statement (**upd** *x*) also requires information about the type of agent *n*, the translation function $\mathcal{CS}$ takes as its input the statements **Stm** as well as **Agents** and the agent's identification number. The update statement (**upd** *x*) is translated into the instructions FETCH *y* : STORE *x*, where *y* is the current runner's time obtained either from a file (agent's type is *manual*) or IP address (agent's type is *automatic*). Informally, the instruction FETCH *y* accesses the current runner's time, and the instruction STORE *x* stores this value into attribute *x*.

Table 9
Meaning of agents.

$\mathcal{A}$: **Adec**→**Agents**→**Agents**
$\mathcal{A}[\![n$ **manual** *file* $]\!]ag = ag[n \rightarrow (manual,file)]$
$\mathcal{A}[\![n$ **auto***ip* $]\!]ag = ag[\![n \rightarrow (auto,ip)]$
$\mathcal{A}[\![A_1,A_2]\!]ag = \mathcal{A}[\![A_2]\!](\mathcal{A}[\![A_1]\!]ag)$

Table 10
Translation of statements.

$\mathcal{CS}$: **Stm**→**Agents** × **Integer**→**Code**
$\mathcal{CS}[\![$ **dec** $x]\!](ag,n) =$ FETCH x : DEC : STORE x
$\mathcal{CS}[\![$ **upd** $x]\!](ag,n) =$ FETCH y : STORE x where$y = \begin{cases} \text{accessfile}(ag(n)\downarrow 2) & \text{if } ag(n)\downarrow 1 = manual \\ \text{connect}(ag(n)\downarrow 2) & \text{if } ag(n)\downarrow 1 = automatic \end{cases}$
$\mathcal{CS}[\![x \coloneqq a]\!](ag,n) = \mathcal{CA}[\![a]\!]$: STORE x
$\mathcal{CS}[\![(b) \rightarrow S]\!](ag,n) = \mathcal{CB}[\![b]\!]$: BRANCH($\mathcal{CS}[\![S]\!](ag,n)$,*NOOP*)
$\mathcal{CS}[\![S_1; S_2]\!](ag,n) = \mathcal{CS}[\![S_1]\!](ag,n) : \mathcal{CS}[\![S_2]\!](ag,n)$

Table 11
Translation of boolean expressions.

$\mathcal{CB}$: **Bexp**→**Code**
$\mathcal{CB}[\![$ **true** $]\!] =$ TRUE
$\mathcal{CB}[\![$ **false** $]\!] =$ FALSE
$\mathcal{CB}[\![a_1 = a_2]\!] = \mathcal{CA}[\![a_2]\!] : \mathcal{CA}[\![a_1]\!]$: EQ
$\mathcal{CB}[\![a_1 != a_2]\!] = \mathcal{CA}[\![a_2]\!] : \mathcal{CA}[\![a_1]\!]$: NEQ

Table 12
Translation of arithmetic expressions.

$\mathcal{CA}$: **Aexp**→**Code**
$\mathcal{CA}[\![n]\!] =$ PUSH n
$\mathcal{CA}[\![x]\!] =$ FETCH x

Other translations of statements are straightforward. The assignment statement $x \coloneqq a$ is translated into a sequence of the translation function $\mathcal{CA}[\![a]\!]$ followed by the instruction STORE x. The predicate statement $(b) \rightarrow S$ is translated into the translation function $\mathcal{CB}[\![b]\!]$ followed by the instruction BRANCH enabling branching between two instruction sets based on a logical condition. The translation is followed by a call of the translation function $\mathcal{CS}[\![S]\!]$ that is executed if the Boolean expression returns the value *true* and the instruction NOOP (no operation) that is executed in another case. In the end, the compound statement $S_1; S_2$ is translated into the sequence of translation functions $\mathcal{CS}[\![S_1]\!]$ and $\mathcal{CS}[\![S_2]\!]$.

The translation of boolean expressions into AM code is specified in Table 11. The Boolean expressions *true* and *false* are straightforward because these are translated to the corresponding instructions TRUE and FALSE, while the Boolean expressions equal $a_1 = a_2$ and not equal $a_1 != a_2$ are translated to the sequence of corresponding translation functions $\mathcal{CA}[\![a_1]\!]$ and $\mathcal{CA}[\![a_2]\!]$ followed by the suitable logical instruction EQ and NEQ.

Finally, the arithmetical expressions are translated into AM code as illustrated in Table 12, where the constant n is translated into the instruction PUSH n, which pushes the value n onto a stack, and the attribute x is translated to the instruction FETCH x, which accesses the attribute x.

An example of generated code is presented in Table 13, where the source code of the EasyTime program in Algorithm 4 is compiled. The informal interpretation of the generated code for measuring place 3 is as follows: The program begins with an instruction WAIT i that waits for the event of some competitors i. When the event is registered at the measuring place 3 the appropriate AM $j=3$ is woken from the wait state and the event is received via the instruction FETCH *connect*(*ip*). Then, the timestamp of the event, which is put onto the stack, is stored into the database attribute INTER2 by instruction STORE INTER2. The decrement of lap counter ROUND2 is performed by a sequence of three instructions, i.e. FETCH ROUND2, DEC and STORE ROUND2. That is, the lap counter ROUND2 is put onto the stack, then decremented and at last, stored back into the database attribute. Next, the interpretation of the conditional statement follows. It consists of predicate testing followed by an operation. The predicate testing represents the sequence of instructions PUSH 0, FETCH ROUND2 and EQ followed by the instruction BRANCH. In other words, a constant 0 and value of attribute ROUND2 is put onto the stack. Then, the logical instruction EQ is executed that enters the value *true* or *false* depending on the result of the

Table 13
Translated code for EasyTime program in Algorithm 4.

```
(WAIT i FETCH accessfile("abc.res") STORE SWIM
FETCH ROUND1 DEC STORE ROUND1, 1)

(WAIT i FETCH accessfile("abc.res") STORE TRANS1, 2)

(WAIT i FETCH connect(192.168.225.100) STORE INTER2
FETCH ROUND2 DEC STORE ROUND2
PUSH 0 FETCH ROUND2 EQ BRANCH(FETCH connect(192.168.225.100) STORE BIKE, NOOP), 3)

(WAIT i FETCH connect(192.168.225.100) STORE INTER3
PUSH 55 FETCH ROUND3 EQ BRANCH(FETCH connect(192.168.225.100) STORE TRANS2, NOOP)
FETCH ROUND3 DEC STORE ROUND3
PUSH 0 FETCH ROUND3 EQ BRANCH(FETCH connect(192.168.225.100) STORE RUN, NOOP), 4)
```

logical operation. If the value of the logical operation on the top of stack is *true* the following sequence of instructions FETCH *connect*(*ip*), STORE BIKE is executed. Otherwise the instruction NOOP is executed. The net effect of these instructions is that the attribute BIKE will be updated only when ROUND2 becomes zero.

However, an informal description of instructions (e.g., FETCH, STORE, BRANCH, NOOP) does not allow formal reasoning and might complicate the implementation of the abstract machine. Therefore, a formal description of the abstract machine is provided in the next subsection.

3.5. The abstract machine

The abstract machine (AM) [21] is configured in the form of $\langle c,e,db,i \rangle$, where

- c is a sequence of instructions to be executed, i.e. a code segment,
- e is the evaluation stack to evaluate arithmetic and boolean expressions (formally, $\mathbf{Stack} = (\mathbf{Int} \cup \mathbf{Bool})^*$, where **Int** denotes integers and $\mathbf{Bool} = \{true,false\}$ denotes boolean values),
- db is the database (formally, $db \in \mathbf{DataBase}$, where $\mathbf{DataBase} = (Id \times Var_1 \times \cdots \times Var_n)^*$),
- i is the starting number of a competitor.

Therefore, the configuration is described as $\langle c,e,db,i \rangle \in \mathbf{Code} \times \mathbf{Stack} \times \mathbf{DataBase} \times \mathbf{Int}$. The configuration of AM is similar to those described in [21], except with the 3rd (db) and 4th component (i) of AM. The database db is a collection of rows, each containing the number of a runner (identification number i) and the results of this runner at various control points. The results are stored in the database's attributes Var_1 to Var_n. The attributes Var_1 to Var_n as well as the database db are created after the compilation of the EasyTime program, which also specifies the attributes of the database db. A sequence of instructions is always executed in the environment where the values of the attributes are stored in the database db and the 4th component (i) represents the runner's starting number. The AM instruction set is:

$c ::= inst : c \mid \varepsilon$
$inst ::=$ PUSH n | TRUE | FALSE
| EQ | NEQ | DEC
| WAIT i | FETCH x | FETCH *connect*(*ip*) | FETCH *accessfile*(*fn*)
| STORE x | NOOP | BRANCH(c,c)

The abstract machine specification is given by operational semantics (Table 14). The meaning of most AM instructions is straightforward. Here, we would like to emphasize the meaning of the instructions WAIT i, FETCH x, FETCH *connect*(*ip*), FETCH *accessfile*(*fn*), and STORE x. The instruction WAIT i puts AM into a waiting state until a runner's starting number i is not signaled and stored into the 4th component of AM. The instruction FETCH x under the current evaluation stack e, the database db, and the runner's identification number j pushes a new value onto the evaluation stack e. A new value is the value of the attribute x stored in the database db for the current runner j. The instruction FETCH *connect*(*ip*) receives an event from the corresponding measuring point via TCP/IP socket (i.e. IP address, protocol TCP or IP and port), while the instruction FETCH *accessfile*(*fn*) gets an event via normal operating systems file operations (*open*, *read*, *close*). In both cases, the corresponding timestamp of an event is set onto the stack. The instruction FETCH x does not change the database db and the runner j. On the other hand, the instruction STORE x changes the database db. It removes the value z from the top of the evaluation stack e and updates the attribute x with the value z for the current runner j.

In summary, the instructions can be divided into arithmetical instructions (DEC, PUSH), logical instructions (TRUE, FALSE, EQ, NEQ), input/output (I/O) instructions (WAIT, FETCH, STORE) and control instructions (BRANCH, NOOP). The arithmetical–logical instructions operate on the stack segment. The I/O instructions enable the AM to communicate with the I/O devices (measuring devices and computer timers) and the database. The control instructions are dedicated to

Table 14
The abstract machine specification.

$\langle \text{PUSH } n : c,e,db,j \rangle \rhd \langle c,n : e,db,j \rangle$	
$\langle \text{TRUE} : c,e,db,j \rangle \rhd \langle c,true : e,db,j \rangle$	
$\langle \text{FALSE} : c,e,db,j \rangle \rhd \langle c,false : e,db,j \rangle$	
$\langle \text{EQ} : c,z_1 : z_2 : e,db,j \rangle \rhd \langle c,(z_1 == z_2) : e,db,j \rangle$	if $z_1,z_2 \in$ **Int**
$\langle \text{NEQ} : c,z_1 : z_2 : e,db,j \rangle \rhd \langle c,(z_1 != z_2) : e,db,j \rangle$	if $z_1,z_2 \in$ **Int**
$\langle \text{DEC} : c,z : e,db,j \rangle \rhd \langle c,(z-1) : e,db,j \rangle$	if $z \in$ **Int**
$\langle \text{WAIT } i : c,e,db,j \rangle \rhd \langle c,e,db,i \rangle$	
$\langle \text{FETCH } x : c,e,db,j \rangle \rhd \langle c,\text{select } x \text{ from } db \text{ where } Id = j : e,db,j \rangle$	
$\langle \text{FETCH } accessfile(fn) : c,e,db,j \rangle \rhd \langle c,time : e,db,j \rangle$	
$\langle \text{FETCH } connect(ip) : c,e,db,j \rangle \rhd \langle c,time : e,db,j \rangle$	
$\langle \text{STORE } x : c,z : e,db,j \rangle \rhd \langle c,e,\text{update } db \text{ set } x = z \text{ where } Id = j,j \rangle$	if $z \in$ **Int**
$\langle \text{NOOP} : c,e,db,j \rangle \rhd \langle c,e,db,j \rangle$	
$\langle \text{BRANCH}(c_1,c_2) : c,t : e,db,j \rangle \rhd \begin{cases} \langle c_1 : c,e,db,j \rangle \\ \langle c_2 : c,e,db,j \rangle \end{cases}$	if $t = true$ otherwise

Table 15
Translation of the statement $(true) \rightarrow S$.

$$\begin{aligned} \mathcal{CS}[\![true \rightarrow S]\!](ag,n) &= \mathcal{CB}[\![true]\!] : \text{BRANCH}(\mathcal{CS}[\![S]\!](ag,n),\text{NOOP}) \\ &= \text{TRUE} : \text{BRANCH}(\mathcal{CS}[\![S]\!](ag,n),\text{NOOP}) \end{aligned}$$

Table 16
Proving the correctness of the optimization step in EasyTime.

$\langle \text{TRUE} : \text{BRANCH}(S_1,S_2) : c,e,db,j \rangle \rhd$ (by true rule)
$\langle \text{BRANCH}(S_1,S_2) : c,\text{TRUE} : e,db,j \rangle \rhd$ (by branch rule)
$\langle S_1 : c,e,db,j \rangle$

controlling the program flow. This set also includes the branch instruction BRANCH and instruction NOOP that increments the instruction counter only.

The benefits of formal semantics in the design phase not only reflect on the easier implementation of a DSL, but formal semantics is also very helpful in proving various algebraic properties and validating various optimization steps. For example, by using formal semantics we can show that the meaning of the statement $(true) \rightarrow S$ is equivalent to the statement S. In other words, we would like to prove that $(true) \rightarrow S \equiv S$. We can show this property by showing that the generated code has the same semantics. The generated code for the statement $(true) \rightarrow S$ is TRUE : BRANCH($\mathcal{CS}[\![S]\!](ag,n)$,NOOP) (Table 15), while the generated code for the statement S is $\mathcal{CS}[\![S]\!](ag,n)$. In order to show that the meaning of both generated codes is the same, we need to prove that TRUE : BRANCH(S_1,S_2) $\equiv S_1$ (the statement S_2 is not important here and can be any sequence of instructions (e.g., NOOP)). Proving this property is easy once we have defined the operational semantics for AM (Table 16).

Thus, we show that the instructions TRUE : BRANCH(S_1,S_2) yield the same configuration as the instruction S_1, but requires two additional transitions. This is the formal proof, so that we can safely and more efficiently translate the statement $(true) \rightarrow S$ into the statement S and thereby optimize the generated code, which will be executed faster on AM.

3.6. Implementation

Various implementation techniques to implement a DSL exist, such as preprocessing, embedding, compiler/interpreter, compiler generator, extensible compiler/interpreter, commercial off-the-shelf, and hybrid approaches [19]. A study by Kosar et al. [15] revealed that not only implementation effort must be taken into account when choosing a suitable implementation technique, but even more important is the effort needed for an end-user to rapidly write correct programs using the produced DSL. If only DSL implementation effort is taken into consideration, then the most efficient implementation technique is embedding. However, the embedding approach might have significant penalties when end-user effort is taken into account (e.g., DSL program size, closeness to original notation, debugging and error reporting). To minimize end-user effort building a DSL compiler [1] is most often a good solution, but at the same time the most costly from an implementation point of view. However, the implementation effort can be greatly reduced, but not as much as with embedding, especially if compiler generators (e.g., LISA [20], ANTLR [23], Silver [32], YAJCO [24]) are used. For this reason, the EasyTime compiler was automatically generated using the compiler-generator tool LISA, which has

proven itself useful in many other DSL projects [10,25,31]. The following LISA features are particularly useful in implementing a DSL:

- integrated development environment where users can specify, generate, compile-on-the-fly, and execute programs in a newly specified language,
- visual presentation of different structures, such as finite state automata, Backus Naur Form (BNF), syntax tree, semantic tree, dependency graph,
- animation of lexical, syntax and semantic analyzers, and
- incremental language development where the language implementer is able to add new features (syntax constructs and/or semantics) to the language in a simple manner by extending lexical, syntax and semantic specifications [20].

LISA specifications are based on attribute grammars [14,22] and consist of

- lexical regular definitions,
- attribute definitions,
- rules which are generalized syntax rules that encapsulate semantic rules, and
- operations on semantic domains.

Lexical specifications for EasyTime are straightforward. The notation used in LISA is similar to those used in other compiler generators.

Algorithm 5. Lexical specifications for EasyTime in LISA.

```
1:  lexicon
2:  {
3:    Int          [0-9]+
4:    Id           [a-zA-Z][a-zA-Z0-9]*
5:    Keywords     mp | agnt | dec | upd | true | false
6:    file         \" [a − z]+\.[a − z][a − z][a − z]\"
7:    ip           [0 − 9][0 − 9][0 − 9]\.[0 − 9][0 − 9][0 − 9]\.[0 − 9][0 − 9][0 − 9]\.[0 − 9][0 − 9][0 − 9]
8:    Operator     = = | != | := | \+ | \* | \- | \= | < = | \-\>
9:    Separator    \; | \( | \) | \[ | \] | \{ | \}
10:   ignore       [\0x09\0x0A\0x0D\]+
11: }
```

While LISA automatically infers whether an attribute is inherited or synthesized [14,22], the type of an attribute must be specified (Algorithm 6). Most of the attributes are derived from semantic specifications (see Section 3.4). For example, the attribute *code* represents generated code using translation functions, the attribute *outAG* is the synthesized attribute and *inAG* the inherited attribute representing agents (*ag* from semantic specifications). Similarly, the attributes *inState* and *outState* represent the state *s* from semantic specifications.

Algorithm 6. Attributes in LISA.

```
1:  attributes String *.code;
2:       BufferedWriter PROGRAM.file;
3:       Hashtable *.outAG,*.inAG;
4:       Hashtable *.inState, *.outState;
5:       int *.number, *.value;
6:       String *.type, *.y;
7:       String *.file_ip, *.name;
8:       int *.n;
9:       boolean *.ok;
```

The most interesting part of LISA specifications consists of generalized syntax rules that also encapsulate semantic rules. EasyTime rules strictly follow semantic specifications from Section 3.4. Readers are invited to compare the specifications in Algorithm 7 with Table 7. During the conversion from the abstract syntax to the concrete syntax (Section 3.3) the production in the abstract syntax $M_1; M_2$ denoting a sequence of measuring places is translated to the following production in the concrete syntax: *MES_PLACES* ::= *MES_PLACE MES_PLACES* | *MES_PLACE*. The translation function $\mathcal{CM}[\![M_1]\!]ag : \mathcal{CM}[\![M_2]\!]ag$ translates into code the first construct M_1 before the translation of the second construct M_2 is performed. This functions is described in LISA as *MES_PLACES*[0].*code* = *MES_PLACE.code* + " \n" + *MES_PLACES*[1].*code* with the following meaning: The code for the first construct *MES_PLACE* is simply concatenated with the code from the second construct *MES_PLACES*[1]. While the abstract syntax for the definition of the measuring place **mp**[n_1]→**agnt**[n_2]*S* is translated to the following production in the concrete syntax: *MES_PLACE* ::= *mp*[#*Int*]− > *agnt*[#*Int*]{*STMTS*}. The translation function (*WAIT i* : $\mathcal{CS}[\![S]\!](ag,n_2),n_1$) is described in LISA as *MES_PLACE.code* = " *WAIT i* " + *STMTS.code* + " ," + *Int*[0].*value*() + ")" .

Algorithm 7. Semantic rules in LISA.

```
1:  attributes String *.code;
2:  rule Mes_places {
3:    MES_PLACES ::=MES_PLACE MES_PLACES compute {
4:      ... // some rules are omitted
5:      MES_PLACES[0].code = MES_PLACE.code + " \n" + MES_PLACES[1].code;
6:    };
7:    MES_PLACES ::=MES_PLACE compute {
8:      ... // some rules are omitted
9:      MES_PLACES.code = MES_PLACE.code;
10:   };
11: }
12:
13: rule Mes_place {
14:   MES_PLACE ::=mp [#Int ] -> agnt [#Int ] { STMTS } compute {
15:     ... // some rules are omitted
16: MES_PLACE.code = " WAIT i" +STMTS.code+" , " +Int[0].value()+" )" ;
17:   };
18: }
```

Overall, building a DSL compiler using various compiler-generator tools drastically reduce implementation efforts [15], while the maintainability of DSL implementation is also improved [13].

4. Practical experiences

The goal of this section is to acquaint the reader with the practical experiences that were obtained by using EasyTime. We have therefore selected two case studies of EasyTime applications:

- A National Championship in time trial bicycle (Slovenia 2010) and
- A World Championship in an ultra double triathlon (Slovenia 2009).

In the rest of this section, these applications are presented in detail and a short analysis of the work conducted is given at the end of the section.

4.1. National Championships in a time trial for bicycle

Measuring time in a National Championship in a time trial for bicycle was not as complicated as measuring time in a triathlon, but still has other specific qualities that need to be carefully noted. At first, a competitor must overcome the time trial course on a bicycle alone. The bicycle is designed especially for this discipline. Then, if two competitors ride bicycles one after the other more times than the rules of the Cycling Federation allow, then the last competitor is disqualified. This occurrence is also known by the name of *drafting*. Finally, the starting time of each competitor is determined in advance. As a result, when the competitor does not come to the start at a predetermined time, they are also disqualified.

Competitors in these competitions are classified into categories according to their age and sex. According to the regulations of the Cycling Federation, competitors are divided into four age groups, as follows:

- age under 17 (U-17),
- age under 19 (U-19),
- age under 23 (U-23),
- age over 23 (Elite).

Typically, each category is characterized by the length of the course, i.e. the older the age category, the longer the course. On the other hand, the length of courses for female competitors is shorter than the lengths of those for men in the same age group. Normally, women classified in the category U-19 take part in courses of the same length as men in the U-17 category. Likewise, the female category U-23 is equivalent to the men's category U-19, etc. The female category U-17 is usually associated with the female category U-19.

In the bicycle time trial, the round courses divided into laps are used for organizational and efficiency reasons. Typically, organizers of time trial bicycle competitions adapt the length of course to the youngest category, i.e. male U-17. The competitors of this category need to accomplish one lap. Then, the number of laps is increased for increasing age categories. In the end, professional competitors, who appear in the male Elite category must overcome four laps.

The measuring time in National Championships in time trial bicycles is illustrated by Fig. 4, where organizers for the time trial course assess the 5 km long raw section of a two-lane highway and close it for all traffic for the duration of the competition. An advantage of such a set up is that organizers can minimize the number of referees on the course because

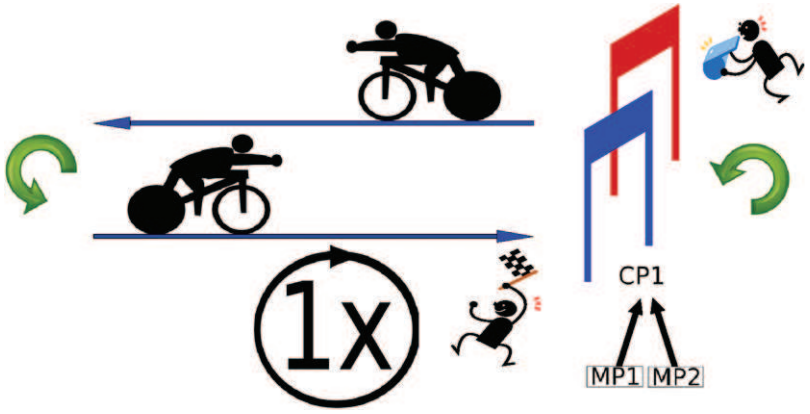

Fig. 4. National Championship in time trial (bicycle).

these can be placed in the turning round only. Note that the turning round is exactly half of the course distance away from the start.

Suppose that for the measurements in Fig. 4 one measuring device with two measuring places is available. Indeed, this device is placed on a finishing point that captures one lane of the highway, i.e. right side. The other lane is occupied by the starting point. In fact, this location is used as the second turning point for those competitors that must finish more than one lap. However, the first turning round is controlled by the referees. Furthermore, both measuring places are connected together and, in fact, constitute one measuring place. In other words, each competitor crosses over two antenna mats because of the higher speed involved with bicycling, which means that the probability of registration could be insecure. In this way, the reliability of event registration is assured.

To manipulate these measurements, the source program in EasyTime (illustrated in Algorithm 8) was developed.

Algorithm 8. EasyTime program for measuring time in a time trial for bicycle.

```
1:   1 auto 192.168.225.100;
2:
3:   var ROUND1 := 2;
4:   var INTER1 := 0;
5:   var INTER2 := 0;
6:   var INTER3 := 0;
7:   var BIKE := 0;
8:
9:   mp[1]→agnt[1]{
10:    dec ROUND1;
11:    (ROUND1 = = 3)→upd INTER3;
12:    (ROUND1 = = 2)→upd INTER2;
13:    (ROUND1 = = 1)→upd INTER1;
14:    (ROUND1 = = 0)→upd BIKE;
15:  }
16:  mp[2]→agnt[1]{
17:    dec ROUND1;
18:    (ROUND1 = = 3)→upd INTER3;
19:    (ROUND1 = = 2)→upd INTER2;
20:    (ROUND1 = = 1)→upd INTER1;
21:    (ROUND1 = = 0)→upd BIKE;
22:  }
```

As can be seen in Algorithm 8, measuring time in time trial competitions requires only one agent. It transfers events from both measuring places. In addition to one finishing time, organizers also wish to have the intermediate times of each competitor. Therefore, we defined five attributes: the lap counter ROUND1, the intermediate times INTER1, INTER2, INTER3, and the finish time BIKE. However, the presence of intermediate times depends on the category of the competitors. For example, if the competitor belongs to male U-17, female U-17 and U-19, they do not have any intermediate times. On the other hand, the Elite competitor has three intermediate times. Indeed, the registering of intermediate times is controlled by the attribute ROUND1 that determines the number of laps to overcome. If the attribute is initialized to ROUND1 = 1, then only the finishing time will be measured. That is, this attribute needs to be set to a correct value before a new category can be started. Fortunately, some break time is taken between categories, in which the EasyTime program can be adjusted.

Table 17
Translated code for the EasyTime program in Algorithm 8.

```
(WAIT i FETCH connect(192.168.225.100) FETCH ROUND1 DEC STORE ROUND1
PUSH 3 FETCH ROUND1 EQ BRANCH(FETCH connect(192.168.225.100) STORE INTER3, NOOP)
PUSH 2 FETCH ROUND1 EQ BRANCH(FETCH connect(192.168.225.100) STORE INTER2, NOOP)
PUSH 1 FETCH ROUND1 EQ BRANCH(FETCH connect(192.168.225.100) STORE INTER1, NOOP)
PUSH 0 FETCH ROUND1 EQ BRANCH(FETCH connect(192.168.225.100) STORE BIKE, NOOP), 1)

(WAIT i FETCH connect(192.168.225.100) FETCH ROUND1 DEC STORE ROUND1
PUSH 3 FETCH ROUND1 EQ BRANCH(FETCH connect(192.168.225.100) STORE INTER3, NOOP)
PUSH 2 FETCH ROUND1 EQ BRANCH(FETCH connect(192.168.225.100) STORE INTER2, NOOP)
PUSH 1 FETCH ROUND1 EQ BRANCH(FETCH connect(192.168.225.100) STORE INTER1, NOOP)
PUSH 0 FETCH ROUND1 EQ BRANCH(FETCH connect(192.168.225.100) STORE BIKE, NOOP), 2)
```

This program is also interesting from the process point of view. In other words, both processes representing measuring places run simultaneously and execute the same piece of program code. Note that the event cannot be generated by the measuring device twice, i.e. only the first registration is taken into account.

The generated code of the EasyTime source program in Algorithm 8 is presented in Table 17.

4.2. World Championship in ultra double triathlon

Everything about measuring time in triathlons was learned at the World Championships in the ultra double triathlon in 2009. The rules of this competition are presented in Section 2. An architecture of the timing system for measuring time in that competition, i.e. the positioning of measuring devices and corresponding measuring places, and determining of control points, is illustrated in Section 2.1 and the source program in EasyTime is illustrated in Algorithm 4. Remember that the measurements were performed using one measuring device with two measuring places, and two computer timers. Two agents took care of the connection between measurement technology and the computer system. The first automatically retrieved events from two antenna mats, while the second retrieved events from two computer timers manually. Agents ran in parallel as threads. However, each agent executed its own program code—in contrast to the bicycle time trial. The main characteristic of this triathlon was its long duration, as the best competitors needed almost 21 h to finish all three disciplines.

4.3. Discussion

In this section, an analysis of the conducted work is discussed from the aspect of EasyTime usage. Therefore, two case studies were taken into account. The first case study demonstrated that the measuring device, on which the timing technology is based, works perfectly, i.e. all events were successfully registered. However, what if the measuring device breaks down? In that case, only the manual tracking of results can solve the problem. In this respect, the timing technology needs a redundancy.

From an EasyTime point of view, the following weakness can be noted: for each category a recompiling of the EasyTime program is needed. As a result, a restart of the agents must be done. However, this can be regarded as a minor issue. On the other hand, it shows the flexibility of EasyTime.

In the second case study, the main weakness was the measuring technology. Although the manual measuring of time showed very good results, this solution demanded the activation of an additional number of people. On the other hand, the measuring devices automated the measuring process but carry with them a purchasing cost. The measuring device with two antenna mats constituted the weakest link of the chain in this case study. The first antenna mat was used for bicycling, while the second for running. With bicycling, the antenna mat can be unreliable. Here, three problems were detected:

- wrong installation of the RFID tag by a competitor,
- a poorly marked antenna mat that could therefore be missed,
- too high speed of bicycles over the mat.

All of the above-mentioned problems have been solved with parallel manual events tracking. Because the number of competitors on the track was small (less than 50) this was an acceptable solution. Moreover, the timing system represents a support tool for referees and not their substitution. That is, parallel manual measurements still remain at most competitions. However, the most difficult problem for these kinds of competitions is the long duration.

From an EasyTime point of view, we did not discover any deficiencies. Fortunately, in triathlon competitions the same lengths of courses were used irrespective of category. From the measuring system point of view, three topics need to be discussed:

- error recovery,
- system failures,
- security measures.

In the case of hardware malfunctioning, two error recovery methods are provided: redundancy of measuring devices that work in parallel and the manual measuring of results. However, the first solution is expensive because of additional hardware that needs to be provided and is therefore not acceptable for small organizers, while the second demands a lot of peoples. If the system breaks down, the registering of data is continued undisturbed but the online results tracking is suspended. When the system is recovered, the data can be transferred from the measuring device and processed in batch. However, the loss of the measuring system can be avoided with redundant system working standby. The competitors cannot tamper with the system because RFID tags are strapped to the leg of the competitor under the control of referees. If the tag is lost by the competitor it can be replaced by another in time. Note that each tag is verified before being used by the competitor.

In summary, EasyTime allows domain-users to program the timing system alone. That is, there is no need for specialized programmers any more. In line with this, our goal of giving domain-users an efficient and easy-to-use tool for measuring time in sports competitions, was fulfilled.

5. Conclusion

The flexibility of the timing system is a crucial objective in the development of universal software for measuring time in sporting competitions. In this sense, we proposed the domain-specific language EasyTime. We were focused on the design of a domain-specific language that consists of three phases: an identification of the EasyTime concepts, the formal definition of language specifications and, finally, the implementation of a compiler. Although many ways for this implementation exist, we decided on the LISA tool, which is able to automatically generate the compiler from formal attribute grammar-based language specifications. In this way, the creator of the domain-specific language is free of implementation tasks that can be difficult and time consuming. Instead of implementation, the creator can focus on the definition of language specifications that demands a lot of specific domain knowledge.

In summary, this approach satisfies our demands about the flexibility of the timing system because it enables the quick adaptation of a timing system to the new requests of different sporting competitions. It supports the accompanying of results in various sporting competitions and reduces the number of necessary measuring devices. With EasyTime, a modification of the timing system is simplified because it demands only the changing of a program's source code, its compiling and restarting of the agent. The case studies applying EasyTime to the real-world have shown that for measuring time in small sporting competitions, the organizers do not need to employ specialized and expensive companies any more. At larger competitions, EasyTime could reduce the heavy configuration task of the timing system.

In future work, EasyTime could be replaced by the domain-specific modeling language (DSML) [27] that could additionally simplify the programming of the timing system.

References

[1] Aho AV, Lam MS, Sethi R, Ullman JD. Compilers: principles, techniques, and tools with gradiance. Upper Saddle River, NY, USA: Prentice Hall PTR; 2007.
[2] Cardelli L, Davies R. Service combinators for web computing. IEEE Transactions on Software Engineering 1999;25:309–16.
[3] Champion Chip. Championchip2010, 〈http://www.championchip.com〉.
[4] van Deursen A, Klint P. Little languages: little maintenance. Journal of Software Maintenance 1998;10(2):75–92.
[5] van Deursen A, Klint P, Visser J. Domain-specific languages: an annotated bibliography. ACM Sigplan Notices 2000;35(6):26–36.
[6] van Deursen A, Klint P. Domain-specific language design requires feature descriptions. Journal of Computing and Information Technology 2002;10: 1–17.
[7] Finkenzeller K. RFID handbook. Chichester, UK: John Wiley; 2010.
[8] Forouzan B. TCP/IP protocol suite. New York, NY, USA: McGraw-Hill; 2009.
[9] Glover B, Bhatt H. RFID essentials. Sebastopol, USA: O'Reilly Media, Inc.; 2006.
[10] Henriques P, Varanda Pereira MJ, Mernik M, Lenič M, Gray J, Wu H. Automatic generation of language-based tools using LISA. IEEE - Proceedings Software Engineering 2005;152(2):54–69.
[11] Hudak P. Modular domain specific languages and tools. In: Proceedings of fifth international conference on software reuse; 1998. p. 134–42.
[12] Kieburtz RB, McKinney L, Bell JM, Hook J, Kotov A, Lewis J, et al. A software engineering experiment in software component generation. In: Proceedings of the 18th international conference on software engineering; 1996. p. 542–53.
[13] Klint P, van der Storm T, Vinju J. On the impact of DSL tools on the maintainability of language implementations. In: Proceedings of the tenth workshop on language descriptions, tools and applications, LDTA '10; 2010. p. 10:1–9.
[14] Knuth D. Semantics of context-free languages. Mathematical Systems Theory 1968;2(2):127–45.
[15] Kosar T, Martínez Lopez PE, Barrientos PA, Mernik M. A preliminary study on various implementation approaches of domain-specific language. Information and Software Technology 2008;50(5):390–405.
[16] Kosar T, Oliveira N, Mernik M, Varanda Pereira MJ, Črepinšek M, da Cruz D, et al. Comparing general-purpose and domain-specific languages: an empirical study. Computer Science and Information Systems 2010;7(2):247–64.
[17] LISA. LISA2010, 〈http://labraj.uni-mb.si//lisa〉.
[18] Mauw S, Wiersma W, Willemse T. Language-driven system design. International Journal of Software Engineering and Knowledge Engineering 2004;6(14):625–64.
[19] Mernik M, Heering J, Sloane A. When and how to develop domain-specific languages. ACM Computing Surveys 2005;37(4):316–44.
[20] Mernik M, Žumer V. Incremental programming language development. Computer Languages, Systems and Structures 2005;31(1):1–16.
[21] Nielson HR, Nielson F. Semantics with applications: a formal introduction. New York, NY, USA: John Wiley & Sons; 1992.
[22] Paakki J. Attribute grammar paradigms—a high-level methodology in language implementation. ACM Computing Surveys 1995;27(2):196–255.
[23] Parr T. The definitive ANTLR reference: building domain-specific languages. Pragmatic Bookshelf; 2007.
[24] Porubän J, Forgáč M, Sabo M, Behálek M. Annotation based parser generator. Computer Science and Information Systems 2010;7(2):291–307.

[25] Rebernak D, Mernik M, Wu H, Gray J. Domain-specific aspect languages for modularizing crosscutting concerns in grammars. IET Software 2009;3(3):184–200.
[26] Schobbens P-Y, Heymans P, Trigaux J-C, Bontemps Y. Generic semantics of feature diagrams. Computer Networks 2007;51:456–79.
[27] Sprinkle J, Mernik M, Tolvanen J-P, Spinellis D. What kinds of nails need a domain-specific hammer? IEEE Software 2009;26(4):15–8.
[28] RFID Time System. RFIDTechnology2010, 〈http://www.rfidtiming.com〉.
[29] Thibault S, Marlet R, Consel C. Domain-specific languages: from design to implementation—application to video device drivers generation. IEEE Transactions on Software Engineering 1999;25(3):363–77.
[30] Timing Ljubljana. Timing2010, 〈http://www.timingljubljana.si〉.
[31] Wu H, Gray JG, Mernik M. Grammar-driven generation of domain-specific language debuggers. Software Practice and Experience 2008;38(10): 1073–103.
[32] van Wyk E, Bodin D, Gao J, Krishnan L. Silver: an extensible attribute grammar system. Science of Computer Programming 2010;75(1–2):39–54.