DOI: 10.2298/CSIS111110021F

Implementation of EasyTime Formal Semantics using a LISA Compiler Generator

Iztok Fister Jr.[1], Marjan Mernik[1], Iztok Fister[1], Dejan Hrnčič[1]

University of Maribor
Faculty of electrical engineering and computer science
Smetanova 17
2000 Maribor
Slovenia
iztok.fister@guest.arnes.si,
marjan.mernik@uni-mb.si,
iztok.fister@uni-mb.si,
dejan.hrncic@uni-mb.si

Abstract. A manual measuring time tool in mass sporting competitions would not be imaginable nowadays, because many modern disciplines, such as IRONMAN, last a long-time and, therefore, demand additional reliability. Moreover, automatic timing-devices based on RFID technology, have become cheaper. However, these devices cannot operate as standalone because they need a computer measuring system that is capable of processing incoming events, encoding the results, assigning them to the correct competitor, sorting the results according to the achieved times, and then providing a printout of the results. This article presents the domain-specific language EasyTime, which enables the controlling of an agent by writing the events within a database. It focuses, in particular, on the implementation of EasyTime with a LISA tool that enables the automatic construction of compilers from language specifications, using Attribute Grammars.

Keywords: domain-specific language, compiler, code generator, measuring time.

1. Introduction

In the past, timekeepers measured the time manually. The time given by a timer was assigned to competitors based on their starting number, and these competitors were then placed in order according to their achieved results and category. Later, manual timers were replaced by timers with automatic time-registers capable of capturing and printing out registered times. However, assigning the times to competitors based on their starting numbers, was still done manually. This work could be avoided by using electronic-measuring technology which, in addition to registering the time, also enables the registering of competitors' starting numbers. An expansion of RFID (Radio Frequency Identification) technology has helped this measuring-technology to become less expensive ([4,

23]), and accessible to a wider-range of users (e.g., sports clubs, organizers of sporting competitions). Moreover, they were also able to compete with time-measuring monopolies at smaller competitions.

In addition to measuring technology, a flexible computer system is also needed to monitor the results. The proposed computer system enables the monitoring of different sporting competitions using a various number of measuring devices and measuring points, the online recording of events, the writing of results, as well as efficiency and security. This measuring device is dedicated to the registration of events and is triggered either automatically, when the competitor crosses the measuring point that acts as an electromagnetic antenna fields with an appropriate RFID tag, or manually, when an operator presses the suitable button on a personal computer that acts as a timer. The control point is the place where the organizers want to monitor the results. Until now, each control point has required its own measuring device. However, modern electronic-measuring devices now allow for the handling of multiple control points, simultaneously. Moreover, each registered event can have a different meaning, depending on the situation within which it is generated. Therefore, an event is handled by the measuring system according to those rules that are valid for the control point. As a result, the number of control points (and measuring devices) can be reduced by using more complex measurements. Fortunately, the rules controlling events can be described easily with the use of a domain-specific language (DSL) [11, 17]. When using this DSL, measurements at different sporting competitions can be accomplished by an easy pre-configuration of the rules.

A DSL is suited to an application domain and has certain advantages over general-purpose languages (GPL) within a specific domain [17]. The GPL is dedicated to writing software over a wider-range of application domains. General problems are usually solved using these languages. However, a programmer is necessary for changing the behavior of a program written in a GPL. On the other hand, the advantages of DSL are reflected in its greater expressive power in a particular domain and, hence, increased productivity [14] , ease of use (even for those domain experts who are not programmers), and easier verification and optimization [17]. This article presents a DSL called EasyTime, and its implementation. EasyTime is intended for controlling those agents responsible for recording events from the measuring devices, into a database. Therefore, the agents are crucial elements of the proposed measuring system. To the best of the author's knowledge there is no comparable DSL of time measuring for sport events, whilst some DSLs for performance measurement of computer systems [2, 21] as well as on general measurement systems do indeed already exist [13]. Finally, EasyTime has been successfully employed in practice, as well. For instance, it measured times at the World Championship for the double ultra triathlon in 2009 [9], and at a National Championship in the time-trials for bicycle in 2010 [9].

The structure of the remaining article is as follows; In the second section, those problems are illustrated that accompany time-measuring at sporting com-

petitions. Focus is directed primarily on triathlon competitions, because they contain three disciplines that need to be measured, and also because of their lengthy durations. The design of DSL EasyTime is briefly shown in section three. The implementation of the EasyTime compiler is described in the fourth section, whilst the fifth section explains the execution of the program written in EasyTime. Finally, the article is concluded with a short analysis of the work performed, and a look at future work. This paper extends a previous workshop paper [10] by providing general guidelines on how to transform formal language specifications using denotational semantics into attribute grammars. The concreteness of these guidelines is shown on EasyTime DSL.

2. Measuring Time in Sporting Competitions

In practice, the measuring time in sporting competitions can be performed manually (classically or with a computer timer) or automatically (with a measuring device). The computer timer is a program that usually runs on a workstation (personal computer) and measures in real-time. Thereby, the processor tact is exploited. The processor tact is the velocity with which the processor's instructions are interpreted. A computer timer enables the recording of events that are generated by the competitor crossing those measure points (MP) in line with the measuring device. In that case, however, the event is triggered by an operator pressing the appropriate button on the computer. The operator generates events in the form of $\langle \#, MP, TIME \rangle$, where $\#$ denotes the starting number of a competitor, MP is the measuring point, and $TIME$ is the number of seconds since 1.1.1970 at 0:0:0 (timestamp). One computer timer represents one measuring-point.

Today, the measuring device is usually based on RFID technology [6], where identification is performed using electromagnetic waves within a range of radio frequencies, and consists of the following elements:

- readers of RFID tags,
- primary memory,
- LCD monitor,
- numerical keyboard, and
- antenna fields.

More antenna fields can be connected on to the measuring device. One antenna field represents one measuring point. Each competitor generates an event by crossing the antenna field using passive RFID tags that include an identification number. This number is unique and differs from the starting number of the competitor. The event from the measuring device is represented in the form of $\langle \#, RFID, MP, TIME \rangle$, where the identification number of the RFID tag is added to the previously mentioned triplet.

The measuring devices and workstations running the computer timer can be connected to the local area network. Communication with devices is performed by a monitoring program, i.e. an agent, that runs on the database server. This

agent communicates with the measuring device via the TCP/IP sockets, and appropriate protocol. Usually, the measuring devices support a $Telnet$ protocol that is character-stream oriented and, therefore, easy to implement. The agent employs the file transfer protocol (ftp) to communicate with the computer timer.

2.1. Example: Measuring Time in Triathlons

Special conditions apply for triathlon competitions, where one competition consists of three disciplines. This article, therefore, devotes most of its attention to this problem.

The triathlon competition is performed as follows: first, the athletes swim, then they ride a bicycle and finally run. In practice, all these activities are performed consecutively. However, the transition times, i.e. the time that elapses when a competitor shifts from swimming to bicycling, and from bicycling to running, are added to the summary result. There are various types of triathlon competitions that differ according to the lengths of various courses. In order to make things easier, organizers often employ round courses (laps) of shorter lengths instead of one long course. Therefore, the difficulty of measuring time is increased because the time for each lap needs to be measured.

Measuring time in triathlon competitions can be divided into nine control points (Fig. 1). The control point (CP) is a location on the triathlon course, where the organizers need to check the measured time. This can be intermediate or final. When dealing with a double triathlon there are 7.6 km of swimming, 360 km of bicycling, and 84 km of running. Hence the swimming course of 380 meters consists of 20 laps, the bicycling course of 3.4 kilometers contains 105 laps, and the running course of 1.5 kilometers has 55 laps (Fig. 1).

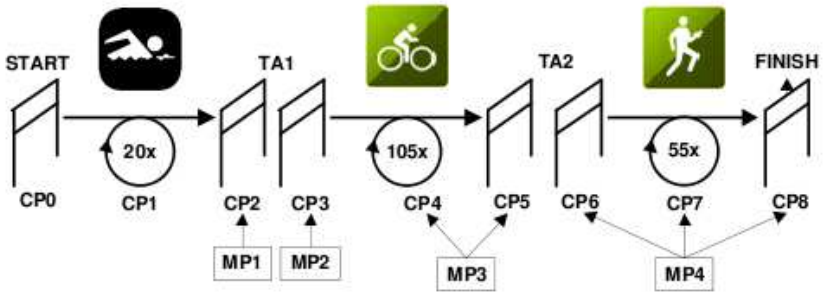

Fig. 1. Definition of control points in the triathlon

Therefore, the final result for each competitor in a triathlon competition (CP8) consists of five final results: the swimming time SWIM (CP2-CP0), the time for the first transition TA1 (CP3-CP2), the time spent bicycling BIKE (CP5-CP3), the time for the second transition TA2 (CP6-CP5), the time spent running RUN (CP8-CP6), and three intermediate results: the intermediate time for swimming

(CP1), the intermediate time for bicycling (CP4) and the intermediate time for running (CP7). However, the current time INTER_x and the number of remaining laps LAPS_x are measured by the intermediate results, where $x = \{1, 2, 3\}$ denotes the appropriate discipline (1=SWIM, 2=BIKE and 3=RUN).

The DSL EasyTime was developed in order to achieve this goal, and has been employed in practice by conducting measurements at the World Championship in the Double Triathlon in 2009. Note that the measurements were realized according to Fig. 1. The next sections presents the design, implementation, and operation of EasyTime.

3. The Design of the EasyTime Domain-Specific Language

Typically, the development of a DSL consists of the following phases [17]:

- a domain analysis,
- a definition of an abstract syntax,
- a definition of a concrete syntax,
- a definition of formal semantics, and
- an implementation of the DSL.

Domain analysis provides an analysis of the application domain, i.e. measuring time in sporting competitions. The results of this analysis define those concepts of EasyTime that are typically represented within a feature diagram [5, 25]. The feature diagram also describes dependencies between the concepts of DSL. Thus, each concept can be broken-down into features and sub-features. In the case of EasyTime, the concept $race$ consists of sub-features: $events$ (e.g., $swimming$, $bicycling$, and $running$), $control\ points$, $measuring\ time$, $transition\ area$, and $agents$. Each $control\ point$ is described by its $starting$ and $finish$ line and at least one lap. In addition, the feature $transition\ area$ can be introduced as the difference between the finish and start times. Both $updating\ time$ and $decrementing\ laps$ are sub-features of $measuring\ time$. However, an $agent$ is needed for the processing of events received from the measuring device. It can act either $automatically$ or $manually$. Note that during domain analysis not all the identified concepts are useful for solving actual problem. Hence, the identified concepts can be further classified into [16]:

- irrelevant concepts, those which are irrelevant to the actual problem;
- variable concepts, those which actually need to be described in the DSL program; and
- fixed concepts, those which can be built into the DSL execution environment.

Domain analysis identifies several variable and fixed concepts within the application domain that needs to be mapped into EasyTime syntax and semantics [17]. At first, the abstract syntax is defined (context-free grammar). Each variable concept obtained from the domain analysis is mapped to a non-terminal in the context-free grammar; additionally, some new non-terminal and

terminal symbols are defined. The translations of the EasyTime domain concepts to non-terminals are presented and explained in Table 1, whilst an abstract syntax is presented in Table 2. Note that, the concepts *Events* and *Transition* are irrelevant for solving actual problem and are not mapped into non-terminals' symbols (denoted as *none* in Table 1). Interestingly, a description of agents and measuring places cannot be found in other DSLs or GPLs. Whilst attribute declaration is similar to variable declaration in many other programming languages. However, note that there is the distinction that variables are actually database attributes allocated for every competitor. Some statements, such as assignment, conditional statement, and compound statement can be found in many other programming languages, whilst decrement attributes and update attributes are domain-specific constructs.

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

Application domain concepts	Non-terminal	Formal sem.	Description
Race	P	$\mathcal{CP}$	Description of agents; control points; measuring places.
Events (swimming, cycling, running)	none	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	none	Can be computed as difference between events final and starting times.
Control points (start, number of laps, finish)	D	$\mathcal{D}$	Description of attributes where start and finish time will be stored as well as remaining laps.
Measuring places (update time, decrement lap)	M	$\mathcal{CM}$	Measuring place id; agent id, which will control this measuring place; specific actions (presented with new non-terminal S) which will be performed at this measuring place (e.g., decrement lap).
Agents (automatic, manual)	A	$\mathcal{A}$	Agent id; agent type (automatic, manual); agent source (file, ip).

Table 2. The abstract syntax of EasyTime

$P \in$ **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**	
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	::= $n \mid x$

Although a language designer can proceed after domain analysis with informal or formal design patterns [17] the formal design step is preferred since it can identify problems before the DSL is actually implemented [27]. Moreover, formal specifications can be implemented automatically by language development systems, thus significantly reducing the implementation effort [17]. The meaning of the EasyTime language constructs is prescribed during the formal semantics phase. Each language construct, belonging to the syntax domain, is mapped into an appropriate semantic domain (Table 3) by semantic functions $\mathcal{CP}$, $\mathcal{A}$, $\mathcal{D}$, $\mathcal{CM}$, $\mathcal{CS}$, $\mathcal{CB}$, and $\mathcal{CA}$ (Table 4).

Table 3. Semantic domains

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

These semantic functions translate EasyTime constructs into the instructions of the simple virtual machine. The meaning of virtual machine instructions has been formally defined using operational semantics (Table 5) as the transition of configurations $< c,\ e,\ db,\ j >$, where c is a sequence of instructions, e is the evaluation stack to evaluate arithmetic and boolean expressions, db is the database, and j is the starting number of a competitor. More details of EasyTime syntax and semantics are presented in [9]. This article focuses on the implementation phase, as presented in the next section.

The sample program written in EasyTime that covers the measuring time in the double ultra triathlon is presented by Algorithm 1. In lines 1-2 two agents are defined. Agent no. 1 is manual and agent no. 2 is automatic. In lines 4-14 several variables, attributes in a database for each competitor, are defined and initialized appropriately. For example, from Figure 1 it can be seen that 20 laps are needed for the swimming course and $ROUND1$ is set to 20, 105 laps are needed for the bicycling course and $ROUND2$ is set to 105, and 55 laps are needed for the running course and $ROUND3$ is set to 55. Lines 16-19 define the first measuring place which is controlled by manual agent no. 1. At this measuring place the intermediate swimming time must be updated in the database ($upd\ SWIM$) and the number of laps must be decremented ($dec\ ROUND1$). Lines 20-22 define the second measuring place which is also controlled by manual agent no. 1. At this measuring place only transition time must be stored in the database ($upd\ TRANS1$). Lines 23-27 define the third measuring place which is controlled by automatic agent no. 2. At this measuring place we must update the intermediate result for bicycling ($upd\ INTER2$) and decrement the number of laps ($dec\ ROUND2$). If a competitor finished

Table 4. EasyTime formal semantics

$\mathcal{CP}$: **Pgm** → **Runners**	→ **Code** × **Integer** × **DataBase**	
$\mathcal{CP}[\![A\ D\ M]\!]r$	=	let $s = \mathcal{D}[\![D]\!]\emptyset$: db =create&insertDB(s, r) in $(\mathcal{CM}[\![M]\!](\mathcal{A}[\![A]\!]\emptyset), db)$
$\mathcal{A}$: **Adec** → **Agents**	→ **Agents**	
$\mathcal{A}[\![n$ **manual** $file]\!]ag$	=	$ag[n \rightarrow (manual, file)]$
$\mathcal{A}[\![n$ **auto** $ip]\!]ag$	=	$ag[n \rightarrow (auto, ip)]$
$\mathcal{A}[\![A_1; A_2]\!]ag$	=	$\mathcal{A}[\![A_2]\!](\mathcal{A}[\![A_1]\!]ag)$
$\mathcal{D}$: **Dec**→**State**	→ **State**	
$\mathcal{D}[\![$**var** $x := a]\!]s$	=	$s[x \rightarrow a]$
$\mathcal{D}[\![D_1, D_2]\!]s$	=	$\mathcal{D}[\![D_2]\!](\mathcal{D}[\![D_1]\!]s)$
$\mathcal{CM}$: **MeasPlace** → **Agents**	→ **Code** × **Integer**	
$\mathcal{CM}[\![$**mp**$[n_1]$ → **agnt**$[n_2]S]\!]ag$	=	(WAIT $i : \mathcal{CS}[\![S]\!](ag, n_2), n_1)$
$\mathcal{CM}[\![M_1; M_2]\!]ag$	=	$\mathcal{CM}[\![M_1]\!]ag : \mathcal{CM}[\![M_2]\!]ag$
$\mathcal{CS}$: **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 $y = \begin{cases} \text{accessfile}(ag(n) \downarrow 2) & \text{if } ag(n) \downarrow 1 = manual \\ \text{connect}(ag(n) \downarrow 2) & \text{if } ag(n) \downarrow 1 = automatic \end{cases}$
$\mathcal{CS}[\![x := a]\!](ag, n)$	=	$\mathcal{CA}[\![a]\!]$:STORE x
$\mathcal{CS}[\![(b) \rightarrow S]\!](ag, n)$	=	$\mathcal{CB}[\![b]\!]$:BRANCH$(\mathcal{CS}[\![S]\!](ag, n), NOOP)$
$\mathcal{CS}[\![S_1; S_2]\!](ag, n)$	=	$\mathcal{CS}[\![S_1]\!](ag, n) : \mathcal{CS}[\![S_2]\!](ag, n)$
$\mathcal{CB}$: **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
$\mathcal{CA}$: **Aexp**	→ **Code**	
$\mathcal{CA}[\![n]\!]$	=	PUSH n
$\mathcal{CA}[\![x]\!]$	=	FETCH x

all the requested 105 laps ($ROUND2 == 0$) then time spent on the bicycle must be stored in the database ($upd\ BIKE$). Lines 28-33 define the fourth measuring place which is also controlled by automatic agent no. 2. At this measuring place we must first check if a competitor has just started running ($ROUND3 == 55$). If this is the case, we must record the transition time between bicycling and running ($upd\ TRANS2$). At this measuring place we also must update the intermediate result for running ($upd\ INTER3$) and decremented number of laps ($dec\ ROUND3$). If a competitor finished all the requested 55 laps ($ROUND3 == 0$) then the final time must be stored in the database ($upd\ RUN$).

Algorithm 1 EasyTime program for measuring time in a triathlon competition as illustrated in Fig. 1

```
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:   (ROUND3 == 55) → upd TRANS2;
30:   (true) → upd INTER3;
31:   (true) → dec ROUND3;
32:   (ROUND3 == 0) → upd RUN;
33: }
```

4. Implementation of the Domain-Specific Language EasyTime

4.1. A LISA Compiler-Generator

One of the benefits of formal language specifications is the unique possibility for automatic language implementation. Although some compiler generators accept denotational semantics [22], the generated compilers are mostly inefficient. Although many compiler-generators based on attribute grammars [12, 20] exist today, we selected a LISA compiler-compiler that was developed at the University of Maribor in the late 1990s [18]. The LISA tool produces a highly efficient source code for: the scanner, parser, interpreter or compiler, in Java. The lexical and syntactical parts of the language specification in LISA supports various well-known formal methods, such as regular expressions and BNF [1]. LISA provides two kinds of user interfaces:

- a graphic user interface (GUI) (Fig. 2), and
- a Web-Service user interface.

The main features of LISA are as follows:

Table 5. The virtual machine specification

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

- since it is written in Java, LISA works on all Java platforms,
- a textual or a visual environment,
- an Integrated Development Environment (IDE), where users can specify, generate, compile and execute programs on the fly,
- visual presentations of different structures, such as finite-state-automata, BNF, a dependency graph, a syntax tree, etc.,
- modular and incremental language development [19].

LISA specifications are based on Attribute Grammar (AG) [20] as introduced by D.E. Knuth [12]. The attribute grammar is a triple $AG = \langle G, A, R\rangle$, where G denotes a context-free grammar, A a finite set of attributes, and R a finite set of semantic rules. In line with this, the LISA specifications (Table 6) include:

- lexical regular definitions (lexicon part in Table 6),
- attribute definitions (attributes part in Table 6),
- syntax rules (rule part before compute in Table 6),
- semantic rules, (rule part after compute in Table 6) and
- operations on semantic domains (method part in Table 6).

Lexical specifications for EasyTime in LISA (Fig. 2) are similar to those used in other compiler-generators, and are obtained from EasyTime concrete syntax (Table 7). Note that in the rule part of LISA specifications the terminal symbols that are defined by regular expressions in the lexical part are denoted with symbol # (e.g., #Id, #Int). EasyTime concrete syntax is derived from EasyTime abstract syntax (Table 2). The process of transforming abstract syntax into concrete syntax is straightforward, and presented in [9]. Semantic rules are written in LISA as regular Java assignment statements and are attached to a particular syntax rule. Hence, the rule part in LISA (Table 6) specifies the BNF production as well as the attribute computations attached to this production. Since the theory about attribute grammars is a standard topic of compiler science, it is assumed that a reader has a basic knowledge about attribute grammars [12, 20].

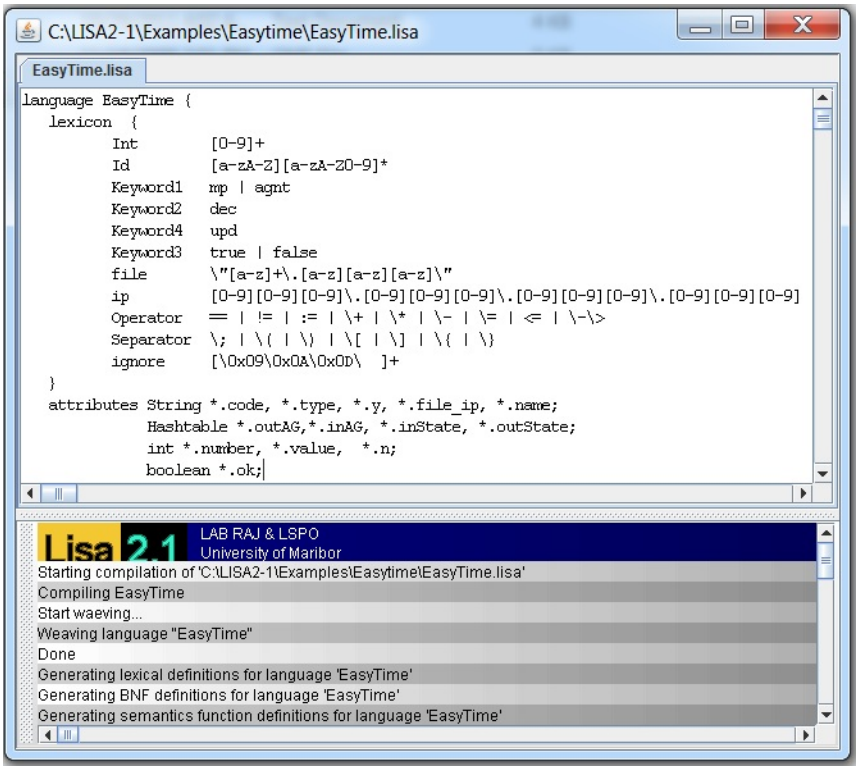

Fig. 2. LISA GUI

4.2. Translation scheme from denotational semantics to attribute grammars

The most difficult part of transforming formal EasyTime specifications into LISA specifications, consists of mapping denotational semantics into attribute grammars. This mapping can be described in a systematic manner, and can also be used for the implementation of other DSLs (e.g., [15]). It consist of the following steps similar to the translation scheme from natural semantics into attribute grammars [3]:

1. Identification of syntactic and semantic domains in each semantic function of denotational semantics. Identified syntactic domains must have their counterparts in non-terminals of concrete syntax. Identified semantic domains must be represented appropriately, with suitable data structures (types) in chosen programming language.
2. Identification of inherited and synthesized attributes for each non-terminal derived in step 1. Semantic argument, which is an input parameter in semantic function, is represented as inherited attribute, while an output parameter is represented as synthesized attribute. According to [12], the starting

Table 6. LISA specifications

```
language L1 [ extends L2, ..., LN ] {
 lexicon {
  [[P] overrides | [P] extends] R regular expr.
   ⋮
 }
 attributes type A1, ..., AM
  ⋮
 rule [[Y] extends | [Y] overrides] Z {
  X ::= X11 X12 ... X1p compute {
    semantic functions }
   ⋮
  |
   Xr1 Xr2 ... Xrt compute {
    semantic functions }
  ;
  }
  ⋮
 method [[N] overrides | [N] extends] M {
  operations on semantic domains
  }
 ⋮
}
```

non-terminal should not have inherited attributes. Whilst LISA automatically infers whether an attribute is inherited or synthesized [12], the type of attribute must be specified (Fig. 2).

3. For all identified attributes attached to a particular non-terminal's, semantic equations need to be developed that are in conformance to semantic equations from denotational semantics. In particular, semantic equations need to be written for each synthesized attribute of the left-hand side non-terminal and for each inherited attribute attached to non-terminals of the right-hand side. This rule is applied to every production of a concrete syntax. In this step the whole semantic equation is not yet written, only the existence of such an equation is identified.
4. In the productions of concrete syntax certain new non-terminals appear, which are consequences of transformation of abstract syntax into concrete syntax. These non-terminals also carry information that are needed for computations. In this step such non-terminals are identified and attached attributes are classified into inherited and synthesized.
5. Finalizing semantics for all identified semantic equations. These semantic equations need to be in conformance to denotational semantics, and require careful examination of semantic functions of denotational semantics (e.g., $\mathcal{CP}$, $\mathcal{A}$, $\mathcal{D}$, $\mathcal{CM}$, $\mathcal{CS}$, $\mathcal{CB}$, and $\mathcal{CA}$ from Table 4). This step is most demanding.
6. In code generation, certain additional tests are usually performed, which are sometimes non-described in formal semantics, in order to be on a proper abstraction level. For example, only declared variables can be used in ex-

Table 7. 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_PLACES | 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

pressions and commands of a language under development. Such additional tests require that new attributes are defined to carry the results of tests, as well as existing attributes being propagated to appropriate constructs (e.g., expressions, commands). An attribute grammar is finalized during this step.

Note that the presented guidelines are general and not restricted to a particular class of attribute grammars [12, 20] (e.g., S-attributed, L-attributed, ordered attribute grammar, absolutely non-circular attribute grammar). Actually, the class of obtained attribute grammar can be identified only after the translation has been completely performed.

4.3. Translation scheme from EasyTime formal semantics to LISA

When applying the aforementioned rules to EasyTime, the following results are obtained after each step.

Step 1:

The following non-terminals from Table 7 represent syntactic domains (Table 2): PROGRAM ∈ **Pgm**, MES_PLACES ∈ **MeasPlace**, DECS ∈ **Dec**, AGENTS ∈ **Adec**, STMTS ∈ **Stm**, etc. Semantic domains (Table 3) such as **Integer**, **TruthValue**, **Code** have direct counterparts with Java types: int, boolean, and String. While semantic domains which are functions (e.g., **State**, **Agents**) can be modeled with Java Hashtable type. For example, from Figure 2 we can notice that attribute $inState$, which represents function **State**, is of type Hashtable. Using methods such as $put()$, $get()$, and $containsKey()$ we can respectively insert a new variable, obtain a variable's value, and check if the variable is declared. Other semantic domains (e.g., cartesian product) can be modeled easily with a Java rich type system. Hence, in LISA the type of attributes regarding an attribute grammar can be any valid pre-defined or user-defined Java type. An example of auxiliary operations on semantic domains (e.g., Hashtable), is presented in [10].

Step 2:

From $\mathcal{CP}$: **Pgm** → **Runners** → **Code** × **Integer** × **DataBase** (Table 4) it can be concluded that to non-terminal PROGRAM one inherited (representing a parameter of type **Runners**) and three synthesized attributes (representing parameters of **Code**, **Integer**, and **DataBase**) need to be attached. However, the starting non-terminal should not have inherited attributes [12, 20]. From the definition of semantic function $\mathcal{CP}$ (Table 4) it can be noticed that the input parameter of type **Runners** are only needed to create a database. Hence, both parameters (of type **Runners** and **DataBase**) can be omitted from LISA specifications, and its functionality can be externally implemented. Moreover, it was decided to represent both the generated code and the identification number of the virtual machine, where the code is going to be executed, as a string **"**(Code, Integer)**"**. Hence, only one synthesized attribute, PROGRAM.code, is attached to starting non-terminal PROGRAM.

From $\mathcal{A}$: **Adec** → **Agents** → **Agents** (Table 4) it can be concluded that one inherited and one synthesized attribute need to be attached to non-terminal AGENTS. For this purpose AGENTS.inAG is an inherited attribute, and AGENTS.outAG a synthesized attribute. Both attributes are of type Hashtable since semantic domain **Agents** is a function, which can be modeled as a Hashtable.

From $\mathcal{D}$: **Dec** → **State** → **State** (Table 4) it can be concluded that one inherited and one synthesized attributes need to be attached to non-terminal DECS. For this purpose DECS.inState is inherited attribute, and DECS.outState a synthesized attribute. Both attributes are of type Hashtable since semantic domain **State** is a function, which can be modeled as a Hashtable.

From $\mathcal{CM}$: **MeasPlace** → **Agents** → **Code** × **Integer** (Table 4) it can be concluded that one inherited and two synthesized attributes need to be attached to non-terminal MES_PLACES. Again, it was decided to represent both, a generated code and the identification number of virtual machine, as a string. For this purpose MES_PLACES.inAG is an inherited attribute and MES_PLACES.code is a synthesized attribute.

From $\mathcal{CS}$: **Stm** → **Agents** × **Integer** → **Code** (Table 4) it can be concluded that two inherited and one synthesized attribute need to be attached to non-terminal STMTS. For this purpose STMTS.inAG and STMTS.n are inherited attributes of type Hashtable and int, respectively. The attribute STMTS.code is a synthesized attribute of type String. The attributes, inherited and synthesized, attached to the appropriate non-terminals are collated in Table 8.

Table 8. Attributes of non-terminals representing syntactic domains from EasyTime formal semantics

X	Inherited(X)	Synthesized(X)
PROGRAM		code
AGENTS	inAG	outAG
DECS	inState	outState
MES_PLACES	inAG	code
STMTS	inAG, n	code

Step 3:

In this step semantic equations are given for each synthesized attribute of the left-hand side non-terminal, and for each inherited attribute for the right-hand side non-terminal. This procedure is applied to each production in the context-free grammar (Table 7). The LISA specification fragment as illustrated in Table 9 indicates, which semantic equations need to be developed. Let us explain the process for the first production. Since the non-terminal PROGRAM, left-hand side non-terminal, has only one synthesized attribute $code$ (Table 8) only one semantic equation must be defined (PROGRAM.code = ...;). Other non-terminals (AGENTS, DECS, MES_PLACES) in the first production are on the right hand side and hence only inherited attributes attached to those non-terminals must be defined (AGENTS.inAG = ...; DECS.inState = ...; MES_-PLACES.inAG = ..;). Note that the order of these semantic equations is irrelevant [12, 20].

Table 9. Semantic equations under development that are obtained after Step 3

```
PROGRAM     ::= AGENTS DECS MES_PLACES compute {
                    AGENTS.inAG = ...;
                    DECS.inState = ...;
                    MES_PLACES.inAG = ...;
                    PROGRAM.code = ...; };

AGENTS      ::= AGENTS AGENT compute {
                    AGENTS[1].inAG = ...;
                    AGENTS[0].outAG = ...; };

DECS        ::= DECS DEC compute {
                    DECS[1].inState = ...;
                    DECS[0].outState = ...; };

MES_PLACES  ::= MES_PLACE MES_PLACES compute {
                    MES_PLACES[1].inAG = ...;
                    MES_PLACES[0].code = ...; };

STMTS       ::= STMT STMTS compute {
                    STMTS[1].n = ...;
                    STMTS[1].inAG = ...;
                    STMTS[0].code = ...; };
```

Step 4:

From step 3, it can be identified the following non-terminals, which appears in concrete syntax (Table 7) and were unidentified in steps 1 - 3: AGENT, DEC, MES_PLACE, and STMT (Table 10). If the structure of these non-terminals is simple (e.g., AGENT, DEC) then attributes attached to these non-terminals carried only synthesized attributes representing mostly lexical values (Table 11). Semantic equations can be derived immediately for those attributes. On the other hand, some non-terminals might be complex (e.g., MES_PLACE, STMT) and inherited attributes attached to these non-terminals are also needed. The attributes might be similar to those attributes attached to other non-terminals in productions, where new non-terminals appear (Table 8). Moreover, semantic equations may no longer be simple (Table 11). For example, attributes attached to non-terminals MES_PLACE and STMT (Table 10) are the same as those attached to non-terminals STMTS and MES_PLACES, respectively (Table 8). However, due to the semantics of the update statement (Table 4) another attribute y is attached to the non-terminal STMT (Table 10).

Table 10. Attributes for additional non-terminals

X	Inherited(X)	Synthesized(X)
AGENT		number, type, file_ip
DEC		name, value
MES_PLACE	inAG	code
STMT	inAG, n	code, y

Step 5:

The reasoning of this step is only explained for semantic functions $\mathcal{A}$ and $\mathcal{CM}$ (Table 4), which are translated into attributes for non-terminals AGENTS, AGENT, MES_PLACES, and MES_PLACE (Tables 8 and 10). For other semantic functions the reasoning is similar. The semantic equation $\mathcal{A}[\![A_1; A_2]\!]ag$ = $\mathcal{A}[\![A_2]\!]$ ($\mathcal{A}[\![A_1]\!]ag$) (Table 4) constructs $ag \in Agents$, which is a function from an integer, denoting an agent, into an agent's type (manual or auto), and an agent's ip or agent's file. This function is described in LISA as presented in Table 12. From Table 12 it can be noticed how the attribute $outAG$, which represents the $ag \in Agents$, is constructed simply by the calling method $insert()$. The method $insert()$ will insert a new agent with a particular number, type, and file_ip into the Hashtable. Note also, how the missing equations from Step 3 have been developed. The net effect is that we are constructing a list, more precisely a hash table, of agents where we are recording the agent's number ($AGENT.number$), the agents's type ($AGENT.type$), and the agent's ip or file ($AGENT.file\_ip$) (see Step 4). The complete LISA specifications for semantic function $\mathcal{A}$, is shown in Algorithm 2.

The reasoning for the semantic function $\mathcal{CM}$ is done in a similar manner. The semantic equation $\mathcal{CM}[\![M_1; M_2]\!]ag$ = $\mathcal{CM}[\![M_1]\!]ag$: $\mathcal{CM}[\![M_2]\!]ag$ (Table 4) translates the first construct M_1 into code before performing the translation of the second construct M_2. This function is described in LISA, as repre-

Table 11. Semantic equations for additional non-terminals

```
AGENT       ::= #Int auto #ip   compute {
                    AGENT.number = Integer.valueOf(#Int[0].value()).intValue();
                    AGENT.type = "auto";
                    AGENT.file_ip = #ip.value(); };

DEC         ::= var #Id  #Int   compute {
                    DEC.name = #Id.value();
                    DEC.value = Integer.valueOf(#Int.value()).intValue(); };

MES_PLACES  ::= MES_PLACE MES_PLACES compute {
                    MES_PLACE.inAG = ...; };

MES_PLACE   ::= mp [ #Int ] -> agnt [ #Int ] { STMTS } compute {
                    MES_PLACE.code= ...; };

STMTS       ::= STMT STMTS compute {
                    STMT.n = ...;
                    STMT.inAG = ...; };

STMT        ::= upd #Id   compute {
                    STMT.y = ...;
                    STMT.code = ...; };
```

Table 12. Semantic equation for AGENTS

```
AGENTS ::= AGENTS AGENT compute {
               AGENTS[1].inAG = AGENTS[0].inAG;
               AGENTS[0].outAG = insert(AGENTS[1].outAG,
               new Agent(AGENT.number, AGENT.type, AGENT.file_ip));
               }
           | epsilon compute {
               AGENTS.outAG = AGENTS.inAG;
               };
```

sented in Table 13, with the following meaning: The code for the first construct MES_PLACE is simply concatenated with the code from the second construct $MES\_PLACES[1]$.

The semantic equation $\mathcal{CM}[\![\textbf{mp}[n_1] \rightarrow \textbf{agnt}[n_2]S]\!]ag$ = $(WAIT\ i : \mathcal{CS}[\![S]\!]$ $(ag, n_2), n_1)$ (Table 4) is described in LISA, as presented in Table 14.

However, in this step the undefined semantic equations from steps 3 and 4 also need to be developed (Table 15). For example, a list of agents ($inAG$) needs to be propagated.

Step 6:

Easytime also uses variables in statements, and additional checks must be performed if only declared variables appear in expressions and statements. For this reason an additional attribute ok of type boolean has been introduced into the specifications. Moreover, to be able to check if a variable is declared, it

Algorithm 2 Translation of Agents into LISA specifications

```
1: rule Agents {
2:    AGENTS ::= AGENTS AGENT compute {
3:       AGENTS[1].inAG = AGENTS[0].inAG;
4:       AGENTS[0].outAG = insert(AGENTS[1].outAG,
5:        new Agent(AGENT.number, AGENT.type, AGENT.file_ip));
6:    }
7:    | epsilon compute {
8:       AGENTS.outAG = AGENTS.inAG;
9:    };
10: }
11: rule AGENT {
12:    AGENT ::= #Int manual #file   compute {
13:       AGENT.number = Integer.valueOf(#Int[0].value()).intValue();
14:       AGENT.type = "manual";
15:       AGENT.file_ip = #file.value();
16:    };
17:    AGENT ::= #Int auto #ip   compute {
18:       AGENT.number = Integer.valueOf(#Int[0].value()).intValue();
19:       AGENT.type = "auto";
20:       AGENT.file_ip = #ip.value();
21:    };
22: }
```

Table 13. Semantic equation for MES_PLACES

```
MES_PLACES ::= MES_PLACE MES_PLACES compute {
                    MES_PLACES[0].code = MES_PLACE.code +
                    "\ n" + MES_PLACES[1].code; };
MES_PLACES ::= MES_PLACE compute {
                    MES_PLACES.code = MES_PLACE.code };
```

is necessary to propagate attribute $inState$ into the measuring places, statements, and expressions. The complete LISA specifications for MES_PLACE are shown in Algorithm 3 also using attributes ok and $inState$.

Semantic equations for other production are obtained in a similar manner. Let us conclude this example by finalizing semantic equations for the starting production (see also Table 9). The initial hash table for agents ($AGENTS.inAG$) and declarations ($DECS.inState$) are empty (Table 16). Agents and declarations are constructed after visiting the subtrees represented by the non-terminals $AGENTS$ and $DECS$, and stored into attributes $AGENTS.outAG$ and $DECS.outState$, that are passed to the subtree represented by the non-terminal MES_PLACES. If all the syntactic constraints are satisfied ($MES\_PLACES.ok == true$), then the generated code is equal to a code produced by the subtree represented by the non-terminal MES_PLACES.

Table 14. Semantic equation for MES_PLACE

```
MES_PLACE ::= mp [ #Int ] - > agnt [ #Int ] { STMTS } compute {
                    MES_PLACE.code= "(WAIT i " + STMTS.code +
                    ", " + #Int[0].value() + ")"; };
```

Table 15. Developing undefined semantic equations for MES_PLACES

```
MES_PLACES ::= MES_PLACE MES_PLACES compute {
                    MES_PLACE.inAG = MES_PLACES[0].inAG;
                    MES_PLACES[1].inAG = MES_PLACES[0].inAG;
                    ... };
MES_PLACES ::= MES_PLACE compute {
                    MES_PLACE.inAG = MES_PLACES.inAG;
                    ... };
```

Table 16. Semantic equations for the starting production

```
PROGRAM ::= AGENTS DECS MES_PLACES compute {
                    AGENTS.inAG = new Hashtable();
                    DECS.inState = new Hashtable();
                    MES_PLACES.inAG = AGENTS.outAG;
                    MES_PLACES.inState = DECS.outState;
                    PROGRAM.code = MES_PLACES.ok ? "\ n" +
                    MES_PLACES.code + "\ n" : "ERROR"; };
```

5. Operation

Local organizers of sporting competitions were faced with two possibilities before developing EasyTime:

- to rent a specialized company to measure time,
- to measure time manually.

The former possibility is expensive, whilst the latter can be very unreliable. However, both objectives (i.e. inexpensiveness and reliability), can be fulfilled by EasyTime. On the other hand, producers of measuring devices usually deliver their units with software for the collecting of events into a database. Then these events need to be post-processed (batch processed) to get the final results of the competitors. Although this batch-processing can be executed whenever the organizer desires, each real-time application requests online processing. Fortunately, EasyTime enables both kinds of event processing.

In order to use the source program written in EasyTime by the measuring system, it needs to be compiled. Note that the code generation [1] of a program in EasyTime is performed only if the parsing is finished successfully. Otherwise the compiler prints out an error message and stops. For each of measuring places individually, the code is automatically generated by strictly following the rules, as defined in Section 3. An example of the generated code from the Algorithm 1 for the controlling of measurements, as illustrated by Fig. 1, is presented in Table 17. Note that the generated code is saved into a database. The meaning of the particular instructions of virtual machine (e.g., WAIT, FETCH, STORE), is explained in Table 5.

As a matter of fact, the generated code is dedicated to the control of an agent by writing the events received from the measuring devices, into the database. Normally, the program code is loaded from the database only once. That

Algorithm 3 Translation of MES_PLACE into LISA specifications

```
rule Mes_places {
   MES_PLACES ::= MES_PLACE MES_PLACES compute {
      MES_PLACE.inAG = MES_PLACES[0].inAG;
      MES_PLACES[1].inAG = MES_PLACES[0].inAG;
      MES_PLACE.inState = MES_PLACES[0].inState;
      MES_PLACES[1].inState = MES_PLACES[0].inState;
      MES_PLACES[0].ok = MES_PLACE.ok && MES_PLACES[1].ok;
      MES_PLACES[0].code = MES_PLACE.code + "\n" + MES_PLACES[1].code;
   };
MES_PLACES ::= MES_PLACE compute {
      MES_PLACE.inAG = MES_PLACES.inAG;
      MES_PLACE.inState = MES_PLACES.inState;
      MES_PLACES.ok = MES_PLACE.ok;
      MES_PLACES.code = MES_PLACE.code;
   };
}
rule MES_PLACE {
   MES_PLACE ::= mp \[ #Int \] \ - \ > agnt \[ #Int \] \{ STMTS \} compute {
      STMTS.inAG = MES_PLACE.inAG;
      STMTS.inState = MES_PLACE.inState;
      STMTS.n = Integer.valueOf(#Int[1].value()).intValue();
      MES_PLACE.ok = STMTS.ok;
      MES_PLACE.code = "(WAIT i " + STMTS.code + ", " + #Int[0].value() + ")";
   };
}
```

is, only an interpretation of the code could have any impact on the performance of a measuring system. Because this interpretation is not time consuming, it cannot degrade the performance of the system. On the other hand, the precision of measuring time is handled by the measuring device and is not changed by the processing of events. In fact, the events can be processed as follows:

- batch: manual mode of processing, and
- online: automatic mode of processing.

The agent reads and writes the events that are collected in a text file, when the first mode of processing is assumed. Typically, events captured by a computer timer are processed in this mode. Here, the agent looks for an existence of the event text file that is configured in the agent statement. If it exists, the batch processing is started. When the processing is finished, the text file is archived and then deleted. The online processing is event oriented, i.e. each event generated by the measuring device is processed in time. In both modes of processing, the agent works with the program PGM, the runner table RUNNERS, and the results table DATABASE, as can be seen in Fig. 3. An initialization of the virtual machine is performed when the agent starts. The initialization consists of loading the program code from PGM. That is, the code is loaded only once. At the same time, the variables are initialized on starting values.

In order to ensure the reliability of Easytime in practice, competitors are not allowed to go directly from swimming to running, because the course is complex and the competitor must to go through both transition areas. In the case that a competitor skips over the next discipline, the referees disqualify

Table 17. Translated code for the EasyTime program in Algorithm 1

```
(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)
```

him/her immediately. Actually, EasyTime is only of assistance to referees. All misuses of the triathlons rules do not have any impact on its operation.

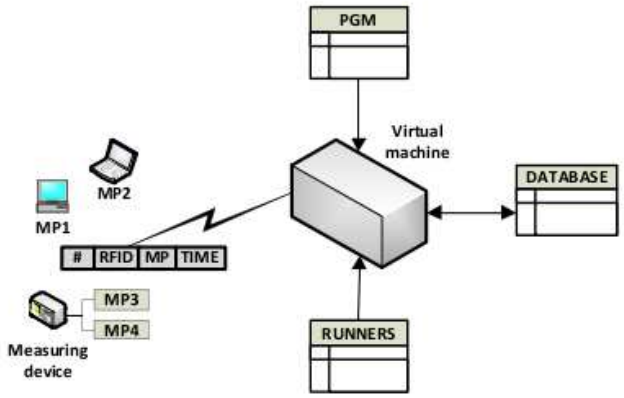

Fig. 3. Executable environment of a program in EasyTime

After the development of EasyTime another demand has arisen - drafting detection in triathlons. This problem is especially expressive in cycling, where competitors wishing to improve their results ride their cycles within close-knit groups. In this way, competitors achieve a higher speed and save energy for later efforts. Typically, within such groups of competitors the hardest work is performed by the leading competitor because he needs to overcome on air resistance. At the same time, other competitors may take a rest. Actually, the drafting violation arises when one competitor rides behind the other closer than 7 me-

ters for more than 20 seconds. Interestingly, this phenomenon is only pursued during long-distance triathlons, whilst drafting is allowed over short-distances. Any competitor who violates this drafting rule is punished by the referees with 5 minutes of elimination from the cycling race. The referees observe the race from motorcycles and determine the drafting violations according to their feelings. In this sense only, this assessment is very subjective. On the other hand, the referees can control one competitor a time. Consequently, an automatic system is needed for detecting drafting violations during triathlons. A drafting detection system is proposed in order to track this violation. This system is based on smart-phones because these incorporate the following features: information access via wireless networks and GPS navigation. Smart-phones need to be borne by competitors on their bicycles (Fig. 4). These determine information about competitor current GPS positions and transmit these over wireless modems to a web-service. From the positions of all competitors the web-service calculates whether a particular competitor is violating the drafting rule. In addition, these violations can be tackled by the referees on motorcycles using smart-phones.

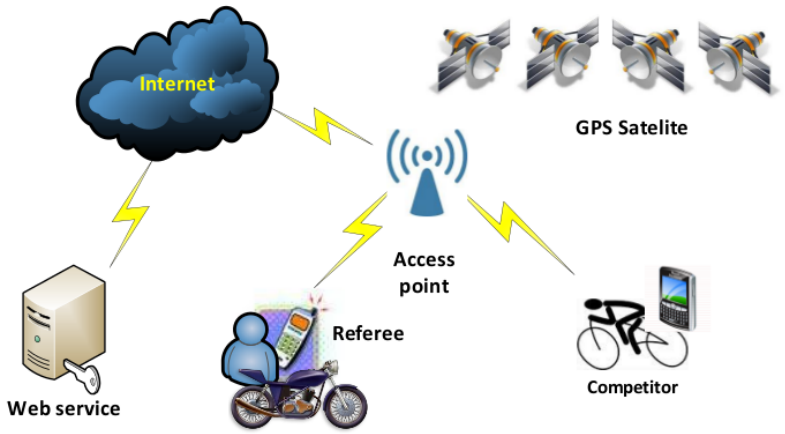

Fig. 4. Proposed system for drafting detection in triathlons

Normally, the organizers of triathlons demand the integration of EasyTime within the system for drafting violation. At a glance, this integration can be performed at the computer-system level, i.e., the mobile agent is added to the existing EasyTime agents. This mobile agent acts as a web-service and runs on an application server. Like EasyTime, it uses its own database. Each record in this database represents a competitor's current GPS position that can be defined as tuple $\langle \#, x, y, z, t, l \rangle$, where $\#$ denotes the competitor's starting number, x, y, z his current position within the coordinate system UTM, t the registration time in the mobile device, and l the calculated path-length. This length l is ob-

tained by projecting the current position of the $\#$-th competitor on the line that connects the points gained by tracking the cycling course with a precise GPS device, at each second. This has an impact on the competitor's current position, from which the distance is calculated to the competitor in front of him. At the moment, both systems run on the same server separately. However, further development of a wireless technology and pervasive computing [29] indicates that EasyTime should have the ability to run on an application server as well.

Interestingly, the measuring time in biathlons represents another great challenge for EasyTime. Here, competitors ski on cross-country skiis and stop at certain places to shoot at targets with rifles carried by them. In order to measure time during biathlons, EasyTime needs to be modified slightly. In line with this, two measuring devices are need, and a special measuring device for counting hits. The first measuring device is dedicated to measuring the four laps of skiing, whilst the second is applied for counting the penalty laps. Each missed shot attracts one additional penalty lap. The measuring device for counting hits is described in EasyTime as a new agent. This agent is responsible for setting the number of additional penalty laps to be measured using the second measuring device. In contrast to the static initialization of the laps counter in EasyTime, a new request is demanded, i.e, a dynamic initialization of this laps counter needs to be implemented.

EasyTime could also be extended and used in some other application domains. For example, EasyTime could be employed as an electric shepherd for tracking livestock (cows, sheep, etc.) in the mountains. In this case, each animal would be labeled with a RFID tag that is controlled by crossing the measuring place twice a day. First, in the morning, when the animals go from their stalls and, second, in the evening, when they return to their stalls. Each crossing of the measuring place by the animal decrements a counter of herd-size for one. Essentially, the EasyTime tracking system reports an error, when the counter is not decreased to zero within a specified time interval. In order for this tracking system to work properly, the herd-size counter has to be initialized twice a day (for example, at 12:00 am and 12:00 pm). Additionally, EasyTime could be used in the clothing industry for tracking cloth through the production. Clothing production consists of the following phases: preparing, sewing, ironing, adjusting, quality-control and packing [7, 8]. The particular cloth origins during the preparation stage, where the parts of cutting patterns are collected into bundles, labeled with the RFID tags, and delivered for sewing. This transition of the bundle into the sewing room presents a starting point for the EasyTime tracking system. The other control points are, as follows: transition from sewing room into ironing, transition from ironing into adjusting, transition from adjusting into quality-control, and transition from quality-control into packing room that represents the finishing point of the cloth production. Note that these transitions act similarly to those transition areas in Ironman competitions. Usually, the cloth does not traverse through the production in any one-way because quality-control can return it to any of the past production phases. In this case, EasyTime could be used for tracking errors during clothing production.

6. Conclusion

The flexibility of the measuring system is a crucial objective in the development of universal software for measuring time in sporting competitions. Therefore, the domain-specific language EasyTime was formally designed, which enables the quick adaptation of a measuring system to the new requests of different sporting competitions. Preparing the measuring system for a new sporting competition with EasyTime requires the following: changing a program's source code that controls the processing of an agent, compiling a source code and restarting the agent. Using EasyTime in the real-world has shown that when measuring times in small sporting competitions, the organizers do not need to employ specialized and expensive companies any more. On the other hand, EasyTime can reduce the heavy configuration tasks of a measuring system for larger competitions, as well. In this paper, we explained how the formal semantics of EasyTime are mapped into LISA specifications from which a compiler is automatically generated. Despite the fact that mapping is not difficult, it is not trivial either, as some additional rules must be defined for attribute propagation. Moreover, we need to take care of error reporting (e.g., multiple declarations of variables). In future work, EasyTime could be replaced by the domain-specific modeling language (DSML) [24, 26, 28] that could additionally simplify the programming of a measuring system.

References

1. A.V. Aho and J.D. Ullman. *The theory of parsing, translation, and compiling.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1972.
2. P. Arpaia, L. Fiscarelli, G. La Commara, and C. Petrone. A model-driven domain-specific scripting language for measurement-system frameworks. *IEEE Transactions on Instrumentation and Measurement*, 60(12):3756–3766, 2011.
3. I. Attali and D. Parigot. Integrating natural semantics and attribute grammars: the minotaur system. Technical Report 2339, INRIA, 1994.
4. Championchip website, 2010.
5. A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10:1–17, 2002.
6. K. Finkenzeller. *RFID Handbook*. John Wiley & Sons, Chichester, UK, 2010.
7. I. Fister, M. Mernik, and B. Filipič. Optimization of markers in clothing industry. *Engineering Application of Artificial Intelligence*, 21(4):669–678, 2008.
8. I. Fister, M. Mernik, and B. Filipič. A hybrid self-adaptive evolutionary algorithm for marker optimization in the clothing industry. *Applied Soft Computing*, 10(2):409–422, 2010.
9. I. Jr. Fister, I. Fister, M. Mernik, and J. Brest. Design and implementation of domain-specific language Easytime. *Computer Languages, Systems & Structures*, 37(4):276–304, 2011.
10. I. Jr. Fister, M. Mernik, I. Fister, and D. Hrnčič. Implementation of the domain-specific language easy time using a LISA compiler generator. In *Proceedings of the Federated Conference on Computer Science and Information Systems*, pages 809–816, Szczecin, Poland, 2011.

11. P. Hudak. Building domain-specific embedded languages. *ACM computing surveys*, 28, 1996.
12. D. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.
13. T. Kos, T. Kosar, and M. Mernik. Development of data acquisition systems by using a domain-specific modeling language. *Computers in industry*, 63(3):181–192, 2012.
14. T. Kosar, M. Mernik, and J.C. Carver. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical software engineering*, 17(3):276–304, 2012.
15. I. Lukovič, M.J. Varanda Pereira, N. Oliveira, D. da Cruz, and P.R. Henriques. A DSL for PIM specifications: Design and attribute grammar based implementation. *Computer Science and Information Systems*, 8(2):379–403, 2011.
16. S. Mauw, W. Wiersma, and T. Willemse. Language-driven system design. *International Journal of Software Engineering and Knowledge Engineering*, 14:1–39, 2004.
17. M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM computing surveys*, 37(4):316–344, 2005.
18. M. Mernik, M. Lenič, E. Avdičaušević, and V. Žumer. Lisa: an interactive environment for programming language development. In *11th International Conference Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 1–4, 2002.
19. M. Mernik and V. Žumer. Incremental programming language development. *Computer Languages, Systems and Structures*, 31(1):1–16, 2005.
20. J. Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995.
21. S. Pakin. The design and implementation of a domain-specific language for network performance testing. *IEEE Transactions on Parallel and Distributed Systems*, 18(10):1436–1449, 2007.
22. L. Paulson. A semantics-directed compiler generator. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 224–233, 1982.
23. Rfid timing system website, 2010.
24. J. Sprinkle, M. Mernik, J-P. Tolvanen, and D. Spinellis. What kinds of nails need a domain-specific hammer? *IEEE Software*, 26(4):15–18, 2009.
25. V. Štuikys and R. Damaševicius. Measuring complexity of domain models represented by feature diagrams. *Information Technology And Control, Kaunas, Technologija*, 38(3):179–187, 2009.
26. V. Štuikys, R. Damaševicius, and A. Targamadze. A model-driven view to metaprogram development process. *Information Technology And Control, Kaunas, Technologija*, 39(3):89–99, 2010.
27. M. Viroli, J. Beal, and M. Casadei. Core operational semantics of proto. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1325–1332, New York, NY, USA, 2011. ACM.
28. R. Vitiutinas, D. Silingas, and L. Telksnys. Model-driven plug-in development for uml based modeling systems. *Information Technology And Control, Kaunas, Technologija*, 40(3):191–201, 2011.
29. M. Weiser. The computer for the 21st century. *Scientific American*, 3:94–104, 1991.

Iztok Fister Jr. is a first-year post-graduate student in Computer Science and Information Technologies at the Faculty of Electrical Engineering and Computer Science, University of Maribor. Besides his study and research activities, especially in the field of web-oriented programming, he is an enthusiastic competitor in triathlons. He is a student member of IEEE.

Mernik Marjan received his M.Sc., and Ph.D. degrees in computer science from the University of Maribor in 1994 and 1998 respectively. He is currently a professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. He is also a visiting professor at the University of Alabama in Birmingham, Department of Computer and Information Sciences, and at the University of Novi Sad, Faculty of Technical Sciences. His research interests include programming languages, compilers, domain-specific (modeling) languages, grammar-based systems, grammatical inference, and evolutionary computations. He is a member of the IEEE, ACM and EAPLS.

Iztok Fister graduated in computer science from the University of Ljubljana in 1983. In 2007, he received his Ph.D. degree from the Faculty of Electrical Engineering and Computer Science, University of Maribor. Since 2010, he has worked as a Teaching Assistant in the Computer Architecture and Languages Laboratory at the same faculty. His research interests include computer architectures, program languages, operational research, artificial intelligence, and evolutionary algorithms. He is a member of IEEE.

Dejan Hrnčič received his B.Sc. degree from the Faculty of Electrical Engineering and Computer Science, University of Maribor, in 2007. Currently he is working on his Ph.D. thesis in computer science. His research interests include evolutionary computation, grammatical inference, and optimization techniques.

Received: November 10, 2011; Accepted: March 6, 2012.