University of Maribor
Faculty of Electrical Engineering and Computer Science

B.Sc. Thesis

Domain-specific language for time measuring on sport competitions

Author:	Iztok Fister, Jr.
Study Program:	University, Computer Science and Information Technologies
Mentor:	prof. dr. Marjan Mernik
Co-mentor:	prof. dr. Barrett R. Bryant

Maribor, June 2011

Univerza v Mariboru

Fakulteta za elektrotehniko, računalništvo in informatiko

Diplomsko delo univerzitetnega študijskega programa

Domensko specifični jezik za merjenje časa na športnih tekmovanjih

Avtor:	Iztok Fister ml.
Študijski program:	Univerzitetni, Računalništvo in informacijske tehnologije
Mentor:	prof. dr. Marjan Mernik
Somentor:	prof. dr. Barrett R. Bryant

Maribor, Junij 2011

Avtor:	Iztok Fister ml.
Naslov:	Domensko specifični jezik za merjenje časa na športnih tekmovanjih
UDK:	004.434(043.2)
Ključne besede:	domensko specifični jeziki, programski jeziki, tehnologija RFID
Število izvodov:	5

Author:	Iztok Fister Jr.
Title:	Domain-specific language for time measuring on sport competitions
UDC:	004.434(043.2)
Keywords:	domain-specific languages, programming languages, RFID technology
Number of issues:	5

Univerza v Mariboru

Fakulteta za elektrotehniko, računalništvo in informatiko

Številka: BRIT-56
Datum in kraj: 17. 05. 2011, Maribor

Na osnovi 330. člena Statuta Univerze v Mariboru (Ur. l. RS, št. 1/2010)

SKLEP O DIPLOMSKEM DELU

1. **Iztoku Fisterju,** študentu univerzitetnega študijskega programa RAČUNALNIŠTVO IN INFORMACIJSKE TEHNOLOGIJE, se dovoljuje izdelati diplomsko delo pri predmetu Prevajanje programskih jezikov.

2. **MENTOR: red. prof. dr. Marjan Mernik**
   SOMENTOR: red. prof. dr. Barret Bryant,
   The University of Alabama at Birmingham

3. **Naslov diplomskega dela:**
   DOMENSKO SPECIFIČNI JEZIK ZA MERJENJE ČASA NA ŠPORTNIH TEKMOVANJIH

4. **Naslov diplomskega dela v angleškem jeziku:**
   DOMAIN SPECIFIC LANGUAGE FOR TIME MEASURING ON SPORT COMPETITIONS

5. Diplomsko delo je potrebno izdelati skladno z "Navodili za izdelavo diplomskega dela" in ga oddati v treh izvodih (dva trdo vezana izvoda in en v spiralo vezan izvod) ter en izvod elektronske verzije do 17. 05. 2012 v referatu za študentske zadeve.

Pravni pouk: Zoper ta sklep je možna pritožba na senat članice v roku 3 delovnih dni.

UNIVERZA V MARIBORU FAKULTETA ZA ELEKTROTEHNIKO RAČUNALNIŠTVO IN INFORMATIKO 8

Dekan:

[illegible]

Obvestiti:

- kandidata,
- mentorja,
- somentorja,
- odložiti v arhiv.

Zahvala

Zahvaljujem se mentorju prof. dr. Marjanu Merniku za pomoč in vodenje pri opravljanju diplomskega dela. Prav tako se zahvaljujem somentorju prof. dr. Barretu R. Bryantu. Posebna zahvala velja staršem, ki so mi omogočili študij.

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.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Domain-specific languages (DSLs) are tailored to the application domains and have definite advantages over the general-purpose languages (GPLs) in a specific domain. These advantages are especially clear in their greater expressive power and, therefore, higher productivity, the ease of use, easier verification and optimization [42] [15] [16] [38] [29] [33]. In this thesis, we have developed the domain-specific language EasyTime for measuring time in sporting competitions.

In the past, a 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 [14] [54] 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 [61] by measuring time in the 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 events tracking and is typically based on the RFID technology. A competitor triggers an event with a RFID tag when they cross over an antenna field that is covered by a measuring device. This tag is normally strapped to a 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. 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 keeping of their monopoly position on the market, there are few papers to treat this topic in existing literature. However, many parts of these solutions have been published in literature that deals with RFID technology [20] [26].

In this thesis, 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 [28] [43] [40]. LISA automatically generates a compiler from formal attribute grammar-based language specifications. While many DSLs have been developed in the past [42] only a few of them have been developed rigorously with a formal semantics approach. Notable examples are found in [11] [60] [41]. Finally, the timing system based on EasyTime was tested in a 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 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 thesis is as follows: in Chapter 2 we describe a basics of programming languages. We focus on differences between general-purpose languages and domain-specific languages. The chapter is finished with a compiling of programming languages. In Chapter 3, we introduce problem of measuring time in sport competitions. In line with this, two case studies from real-world are taken into consideration, i.e., triathlons and time trial bicycle. Development and implementation of EasyTime is presented Chapter 4. In Chapter 5, we illustrate practical experiences using EasyTime. The last chapter summarizes conclusions of performed work and indicates directions for the future work. Abbreviations, Easytime source code and extended abstract in Slovene language are included in appendix.

Chapter 2

Programming Languages

A programming language is an interface between a user and a computer. This interface is defined by the grammar that describes the syntax of how keywords, names and symbols are used to build constructions, such as expressions and statements, as well as the semantics that describe what the computer should do for each construct. The theory of computer languages is one of the most surveyed topics in computer science. Currently, there are more than 3,500 computer languages [12] [6] [4] [8] [9]. For programmers, it is virtually impossible to be familiar with all of these. Some computer languages are intended for quick solutions, some are for prototyping and others are for bigger projects. Programming languages have many formal definitions. We used informal definitions of programming languages [53] [44], as follows:

- a programming language is a language for creating computer programs that contains calculations, or algorithms with the possibility of control of external devices such as printers, robots, etc...
- a programming language maintains communication with humans,
- a programming language allows instructions to be given that perform computing procedures, which are easy for humans and computers to read,
- a programming language is the basic tool of a programmer.

A programming language needs to be universal. It must allow one to formally describe every solvable task. Programs are processed by compilers and interpreters that translate the program into a form that can be executed by another program, or have it executed immediately.

Programming languages are divided into general-purpose languages and domain-specific languages.

2.1 GPL

Among GPLs, there are dedicated languages, whose expression power goes beyond the specific domain. With these languages, we can develop programs for an arbitrary domain [13] [19]. The most popular GPLs are:

- Python,
- Ruby,
- C++,
- Java,
- Pascal,
- PHP.

2.2 DSL

DSLs are languages tailored to a specific application domain. DSL is a small, usually declarative, language that offers expressive power focused on a particular problem domain. In many cases, DSL programs are translated to calls into a common subroutine library; the DSL can be viewed as a means to hide the details of that library. These libraries contain subroutines that perform related tasks in well-defined domains such as differential equations, graphics, user-interfaces and databases. The subroutine library is the classical method for packaging reusable domain-knowledge. The most popular DSL languages are:

- BNF notation,
- Hyper Text Markup Language (HTML) for World Wide Web (WWW),
- LaTeX word processing software,
- Make software,
- SQL database queries.

As with other languages, DSLs have both advantages and disadvantages. The most important advantages of DSLs are the following:

- DSLs allow solutions to be expressed in an idiom and at the level of the problem domain abstraction. Consequently, domain experts themselves can understand, validate, modify, and often even develop DSL programs.
- DSL programs are concise, self-documenting to a large extent, and can be reused for different purposes. DSLs enhance productivity, reliability, maintainability and portability.
- DSLs embody domain knowledge, and thus enable the conservation and reuse of this knowledge; DSLs allow validation and optimization at the domain level.
- DSLs improve testability.

The disadvantages of DSLs are the following:

- the costs of designing, implementing and maintaining a DSL,
- the costs of education for DSL users,
- the limited availability of DSLs,
- the difficulty of finding the proper scope for a DSL,
- the difficulty of balancing between domain-specificity, and general-purpose programming language constructs,

- the potential loss of efficiency when compared with hand-coded software.

Domain-specific languages are used in numerous fields:

- pharmacy: simulations, calculations, cure productions,
- bioinformatics: protein structure prediction, applications,
- software engineering: financial products, behavior control and coordination, software architectures, and databases,
- systems software: description and analysis of abstract syntax trees, video device driver specifications, cache coherence protocols, data structures in C, and operating system specialization,
- multi-media: web computing, image manipulation, 3D animation, and drawing,
- telecommunications: string and tree languages for model checking, communication protocols, telecommunication switches, and signature computing,
- sport applications: simulations, time measuring, competitor tracking,
- science applications: optimization, measuring, calculations,
- computer graphics: triangulations, surfaces and functions.

The development of DSLs is a complicated process. The process typically consists of three phases, i.e.:

- Analysis,
- Design,
- Implementation.

2.3 Compiling Programming Languages

The program in a source code cannot be executed directly. Before execution, we have to compile the source code into a machine code. This translation is performed by a compiler [2]. The compiler is a computer program that compiles source code written in a programming language into a computer machine language. The result of the compiling is an executable file [56].

The name compiler is primarily used for programs that translate source code from a high-level programming language to the low level programming language. Compiling consists of three stages [53] [21]:

- lexical analysis,
- syntactical analysis and
- semantic analysis.

In the next subsections, these stages will be briefly discussed.

2.3.1 Lexical Analysis

The lexical analysis is the first task to analyze the source program that looks for lexical symbols. The lexical symbols include the names of identifiers, character strings, numbers, operators, separator and reserver words. A program designed to conduct a lexical analysis of the source code is called a lexical analyzer. Its mission is to provide terminal symbols for the syntactical analyzer. For the realization of a lexical analyzer finite-state-automata are usually used. The finite-state machine enables the transition of one state to another, according to the current input [39]. The transition always starts in the initial state and ends in the final stage.

2.3.2 Syntactical Analysis

The syntactical analyzer determines a structure of sentences written in a specific language. The input of the syntactic analysis is a sequence of symbols, where a lexical analyzer breaks the code. The output is a description of the syntactic structure of the original program. A syntactic analysis can be performed in several ways, but all ways are equivalent to a derivation tree. The analysis procedure needs to satisfy certain conditions dictated by simplicity and practicality. These conditions are as follows:

- unambiguous grammar,
- reading the input line from left to right, and
- a deterministic process of syntactic analysis - execution without backtracking.

The part of the compiler that performs a syntactic analysis is known as a parser or syntactic analyzer. In general, the parsers are realized in two ways:

- a top-down parser, and
- a bottom-up parser.

The former starts from the initial symbol and tries to build a derivation tree. The latter starts from the leaves of the tree (terminal symbols) and tries to move towards the root of the tree.

2.3.3 Semantic Analysis

Syntactic analysis enables the identification of programming language constructs, i.e., whether a sequence of input symbols is correct or not. However, it ignores the meaning of sentences. To interpret the meaning of sentences, semantic analysis is used. The semantic information is obtained by extending the context-free grammar which defines the syntactic structure of the program with the context-sensitive information by appending attributes to each non-terminal symbol. In this manner, an attribute grammar is obtained. Normally, the attribute grammar is represented in an abstract syntax tree, where attributes move via the branches of trees from the leaves towards the root and vice versa [7]. Attributes are usually divided into two groups:

- synthesized,
- inherited.

Synthesized attributes are used to pass semantic information up the parse tree, while inherited attributes help pass semantic information down and across it. For instance, when constructing a language translation tool, such as a compiler, it may be used to assign semantic values to syntax constructions. Also, it is possible to validate the semantic checks associated with a grammar, representing the rules of a language not explicitly imparted by the syntax definition [1].

2.4 Compiler Generators

The Compiler generators are programs that convert a formal description of a programming language into a compiler for that language. The language description may take many forms, but usually contain a large amount of program code. Several recent compiler generators accept descriptions in terms of attribute grammars or denotational semantics [35] [58] [49]. Some of the most popular compiler generators are:

- LISA,
- Antlr, and
- Yacc/Bison.

In the next subsections, these generators are briefly presented.

2.4.1 LISA

LISA(Language Implementation System based on Attribute Grammars) is a compiler-compiler, i.e., a system that automatically generates a compiler/interpreter from formal attribute grammar-based language specifications. LISA was developed at the University of Maribor in the late 1990s [40]. The LISA tool produces highly efficient source code for the scanner/parser/interpreter/compiler in Java. The lexical and syntactical parts of a language specification use well known formal methods, such as regular expressions and BNF. The semantics are further defined with attribute grammars. The graphical user interface (GUI) of LISA (Figure 2.1) is written in Java. Moreover, the LISA tool provides a Web-Service user interface. The main features of LISA are the following:

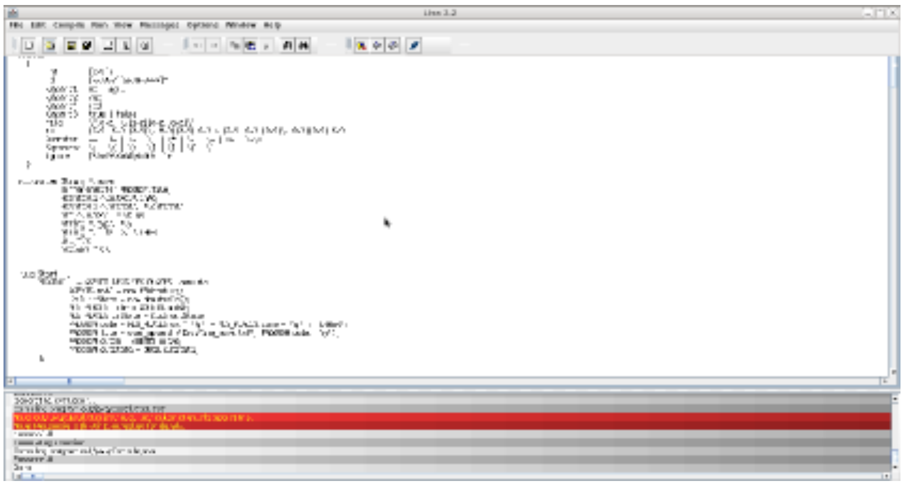

Figure 2.1: LISA GUI.

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

- modular and incremental language development.

2.4.2 ANTLR

ANTLR (ANother Tool for Language Recognition) is a parser generator that uses LL(*) parsing. The tool was developed in 1989. [48]. Currently, professor Terence Parr at the University of San Francisco is a maintainer of ANTLR. ANTLR is still under active development. The tool takes a grammar as an input and generates a source code for a syntax recognizer. The specified language uses a context free grammar, which is expressed using an Extended Backus-Naur Form (EBNF) [10] [47]. ANTLR allows for the generation of:

- parsers,
- lexers,
- tree parsers, and
- combined lexer parsers.

ANTLR is free software published under a BSD-license. The tool works on all computer platforms.

2.4.3 Yacc/Bison

Yacc is a parser generator developed by Stephen C. Johnson for the Unix operating system [31]. Generating results in a parser based on analytic grammar written in a notation similar to BNF [27] [30]. A newer version of Yacc is Bison. It is a parser generator that was developed by the GNU project [18]. Bison is a parser generator that reads the specifications of a context-free language, providing warnings about any parsing ambiguities, and then generates a parser (either in C, C++, or Java), which reads sequences of tokens and decides whether the sequence conforms to the syntax specified by the grammar. Bison generates LALR parsers (i.e., a type of parser defined as a Look-Ahead LR parser) that are based on a finite-state-automata concept. Currently, Bison is free software. It can be obtained either as source code or in executable formats and works across all computer platforms.

Chapter 3

Measuring Time on Sport Competitions

In practice, time in sporting competitions can be measured *manually* (with a classical or computer timer) or *automatically* (with a measuring device). In this thesis, 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 the measuring place, where the event happened and $TIME$ 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 the 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 $\langle Ev, \# \rangle$ needs to be recorded to a database via file transfer protocol [25].

For automatic time measuring devices are used today. Typically, these are based on RFID technology [20] that is discussed in the next section in more details.

3.1 RFID technology

An identification is performed with an electromagnetic wave motion in the range of a radio frequency by RFID technology [32]. 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 of RFID tags,
- the identification number in tags can be modified.

The most commonly, the active and passive RFID technologies are used today. For the active RFID technology, no electrical power is needed. Furthermore, the tags can be read from large distances (more than 100 meters) and they can keep more information than passive RFID technology. However, the main weakness of this technology is the use of batteries that needs 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 primarily 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 a 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 the 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 by 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 [59]. 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 $Ev2 = \langle \#, RFID, MP, TIME \rangle$. Typically, the courses in sporting competitions are divided into multiple *control points* (CP), where the organizers needs to track the measured time. This time can be *intermediate* or $final$. The location of the control points depend on the kind of competition and the configuration of the course in which the competition takes place.

In order to illustrate, how to use the proposed measuring technology in the real-world two case studies are taken. The first case study presents a scheme of time measuring in triathlon competition, while the second a scheme of the time trial bicycle.

3.2 Case Study 1: Triathlon Competitions

The triathlon is a modern sporting event involving the completion of three continuous and sequential endurance disciplines [57] [50], i.e., the swimming (Fig. 3.1), cycling and running. Triathletes compete for the fastest overall course completion time, including *transitions* times, where they move from swimming to bicycling (transition 1) and from bicycling to running (transition 2). Transition area is the place, where triathletes prepare himself for the next discipline. In the transition area 1 competitor take off swimming wetsuit and take a bike. In transition area 2 competitor let a bike, take shoes, and start to run. Triathlon races vary in distance. According to the organizations International Triathlon Union (ITU) and European Triathlon Union (ETU) the main international race distances are standardized, as follows:

- sprint distance (i.e., 750 m swim, 20 km bike, 5 km run),
- olympic distance (i.e., 1.5 km swim, 40 km bike, 10 km run),
- half Ironman (i.e., 1.9 km, 90 km, 21.1 km),
- full Ironman (i.e., 3.8 km, 180 km, 42.2 km),
- Ultra Triathlon (i.e., n * IRONMAN, where n = 2, 3,...).

Ironman is one of the long-distance triathlon series organised by the World Triathlon Corporation (WTC). It consists of 3.8 km swimming, 180 km bicycling and 42.2 km running. Ironman's are limited in time. Normally, the competitors have 17 hours to finish the race. Simultaneously with Ironman, Ultra Triathlon series that are organized by International Ultra Triathlon Association (IUTA) have been rising as well. Because of long duration, these races are very complicated for

Figure 3.1: IRONMAN 70.3 Monaco.

measuring. Therefore, special time keeping companies for time measuring are need by organizers of sporting competitions. Most races consists of short laps, which also complicate the time measuring.

The measuring time in the triathlon is divided into nine control points. In Fig. 3.2, the scheme of the World Championship 2009 in ultra double triathlon is illustrated, 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. 3.2 that one measuring device with two measuring places (MP3 and MP4)

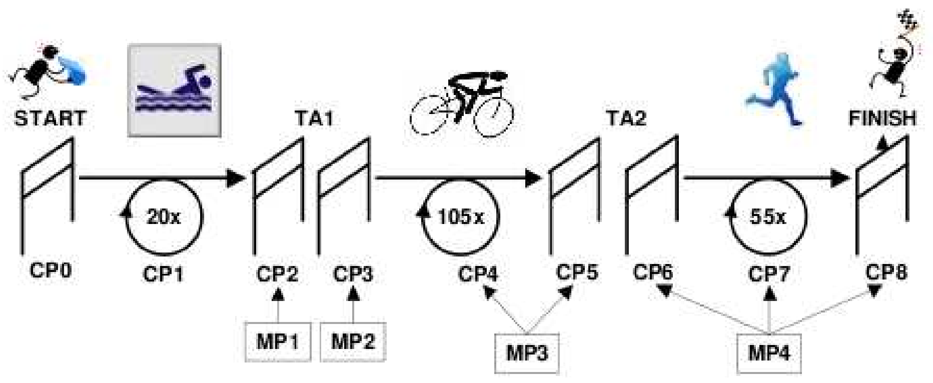

Figure 3.2: Definition of control points in a triathlon competition.

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 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.

As can be seen from this scenario, each registered event on particular measuring place can have a different meaning according to a moment in which it is issued. For example, the each last event on MP3 represents the finish time BIKE, while all others 104 events update the lap counter INTER_2. However, this different meaning of events can be described with DSL that enables the automatic configuration of measuring devices according to requests of particular sporting competition.

3.3 Case Study 2: Time Trials Bicycle

A time trial bicycle is a road bicycle discipline race in which riders race alone against the clock. Thereby, slip-streaming is prohibited [5] [62].

The time trial bicycle is supported by the organization International Cycling Union (UCI). Time trial bicycle is also referred to as "the race of truth" by the competitors because that race can only be won by the strongest and the most endurance riders. That race is involved on all big races (e.g., Tour de France, Giro, Vuelta, etc). There exist also track-based time trials where riders compete in velodromes, and team time trials bicycle.

Measuring time in a national championship in a time trial for bicycle 2010 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 are 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. 3.3, where organizers for the time trial course assess the 5 kilometer 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 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.

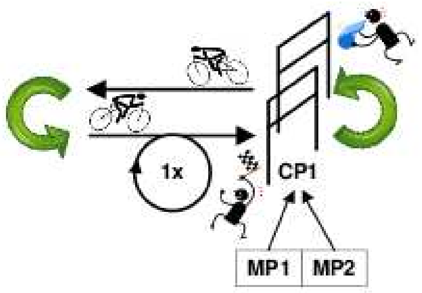

Figure 3.3: National Championship in time trial (bicycle).

Suppose that for the measurements in Fig. 3.3 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.

Once again, the usage of DSL can simplify the configuration of measuring devices.

Chapter 4

EasyTime

Measuring time in mass sporting competitions is, typically, performed with a timing system that consists of a measuring technology and a computer system [22] [23] [24]. 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 order to automate a measuring time in sporting competitions, 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. In this thesis, we are focused on the development of this domain-specific language that was tested in real-world, i.e., by measuring time in two sporting competitions presented in section 3.2 and section 3.3.

Development of domain-specific language Easytime is divided into three phases, as follows:

- domain analysis using feature diagrams,
- DSL design that includes definitions of:
  - abstract syntax,
  - formal semantics,

  – abstract machine.

- implementation.

In the next sections, the particular phases are discussed in detail.

4.1 Domain Analysis

A prerequisite for the design of a DSL is the detailed analysis and structuring of the application domain [17]. The analysis of the application domain is provided by a *domain analysis*. The results of the domain analysis are obtained in a *feature model* [55]. 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. 4.1.

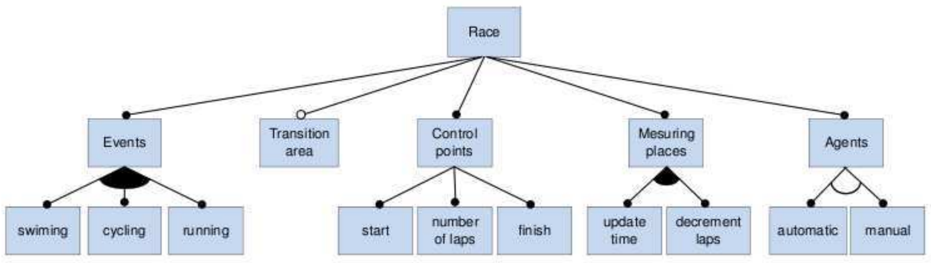

Figure 4.1: The feature diagram of EasyTime.

From the FD in Fig. 4.1, 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: $7(Events)\times2(Transition\ area)\times1(Control\ points)\times3(Measuring\ places)\times2(Agents)$.

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 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.

4.2 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 on 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 4.1 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 4.1.

Table 4.1: Translation of the application domain concepts to a context-free grammar

Application domain concepts	LHS non-term.	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).

The abstract syntax of EasyTime, which is based on Table 4.1, is presented in Table 4.2, whereas the syntactic domains of variables are presented in Table 4.3. 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 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 (! =).

Table 4.2: The abstract syntax of EasyTime

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

Table 4.3: 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**	

4.3 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 manual "abc.res";
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

```
var ROUND2 := 105;
var INTER2 :=0;
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

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

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 4.4, the EasyTime concrete syntax is given. In Fig. 4, a complete example of the EasyTime program for measuring time in a triathlon is also presented.

Table 4.4: 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_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

4.4 Formal Semantics

The translation of an EasyTime program into a code that is going to be executed on several abstract machines (AM), described in subsection 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 4.5. Among classical semantic domains [45] 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 4.5: 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**

Algorithm 4 EasyTime program for measuring time in a triathlon

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

var ROUND1 := 20;
var INTER1 := 0;
var SWIM := 0;
var TRANS1 :=0;
var ROUND2 := 105;
var INTER2 :=0;
var BIKE := 0;
var TRANS2 :=0;
var ROUND3 := 55;
var INTER3 := 0;
var RUN := 0;

mp[1] → agnt[1] {
  (true) → upd SWIM;
  (true) → dec ROUND1;
}
mp[2] → agnt[1] {
  (true) → upd TRANS1;
}
mp[3] → agnt[2] {
  (true) → upd INTER2;
  (true) → dec ROUND2;
  (ROUND2 == 0) → upd BIKE;
}
mp[4] → agnt[2] {
  (true) → upd INTER3;
  (ROUND3 == 55) → upd TRANS2;
  (true) → dec ROUND3;
  (ROUND3 == 0) → upd RUN;
}
```

The translation of an EasyTime program into an AM code is specified in Table 4.6. 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 4.7). 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

Table 4.6: Translation of the program

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

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.

Table 4.7: Translation of measuring places

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

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 4.8). 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 4.9.

Table 4.8: Meaning of declarations

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

Table 4.9: Meaning of agents

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

The translation of the statements **Stm** into AM code is specified in Table 4.10. Since the update statement ($\mathbf{upd}\ x$) 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 ($\mathbf{upd}\ x$) 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.

Other translations of statements are straightforward. The assignment statement $x := a$ 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 $\mathcal{CS}[\![S_2]\!]$.

Table 4.10: 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
		$y = \begin{cases} \text{accessfile}(ag(n) \downarrow 2) & \text{if } ag(n) \downarrow 1 = manual \\ \text{connect}(ag(n) \downarrow 2) & \text{if } ag(n) \downarrow 1 = automatic \end{cases}$
$\mathcal{CS}[\![x := a]\!](ag,n)$	=	$\mathcal{CA}[\![a]\!]$:STORE x
$\mathcal{CS}[\![(b) \to S]\!](ag,n)$	=	$\mathcal{CB}[\![b]\!]$:BRANCH($\mathcal{CS}[\![S]\!](ag,n), NOOP$)
$\mathcal{CS}[\![S_1; S_2]\!](ag,n)$	=	$\mathcal{CS}[\![S_1]\!](ag,n) : \mathcal{CS}[\![S_2]\!](ag,n)$

The translation of boolean expressions into AM code are specified in Table 4.11. The Boolean expressions *true* and *false* are straightforward because these are translated to the corresponding instructions TRUE and FALSE, while the Boolean expression 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 4.12, 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.

Table 4.11: 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 4.12: Translation of arithmetic expressions

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

An example of generated code is presented in Figure 4.13, 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 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.

4.5 The Abstract Machine

The abstract machine (AM) [45] is configured in the form of $< c, e, db, i >$, where:

Table 4.13: 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)
```

- 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, **Stack** = (**Int** ∪ **Bool**)*, where **Int** denotes integers and **Bool** = $\{true, false\}$ denotes boolean values),
- db is the database (formally, $db \in$ **DataBase**, where **DataBase**= $(Id \times Var_1 \times \ldots \times Var_n)^*$),
- i is the starting number of a competitor.

Therefore, the configuration is described as $< c, e, db, i > \in$ **Code** × **Stack** × **DataBase** × **Int**. The configuration of AM is similar to those described in [45], except with the 3^{rd} (db) and 4^{th} 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:

$$
\begin{array}{lll}
c & ::= & inst : c \mid \epsilon \\
inst & ::= & \text{PUSH } n \mid \text{TRUE} \mid \text{FALSE} \\
 & \mid & \text{EQ} \mid \text{NEQ} \mid \text{DEC} \\
 & \mid & \text{WAIT } i \mid \text{FETCH } x \mid \text{FETCH } connect(ip) \mid \text{FETCH } accessfile(fn) \\
 & \mid & \text{STORE } x \mid \text{NOOP} \mid \text{BRANCH}(c, c)
\end{array}
$$

Table 4.14: The abstract machine specification

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

The abstract machine specification is given by operational semantics (Table 4.14). 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 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 $(true) \to S$ is equivalent to the statement S. In other words, we would like to prove that $(true) \to S \equiv S$. We can show this property by showing that the generated code has the same semantics. The generated code for the statement $(true) \to S$ is TRUE : BRANCH($CS[\![S]\!](ag, n)$,NOOP) (Table 4.15), 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 4.16).

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 $(true) \to S$ into the statement S and thereby optimize the generated code, which will be executed faster on AM.

Table 4.15: Translation of the statement $(true) \rightarrow S$

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

Table 4.16: Proving the correctness of the optimization step in EasyTime

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

4.6 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 [42]. A study by Kosar et al. [38] 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 [3] 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 [43], ANTLR [48], Silver [65], YAJCO [51]) 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 [28] [52] [64].

LISA specifications are based on attribute grammars [36] [46] 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

```
lexicon
{
    Int          [0-9]+
    Id           [a-zA-Z][a-zA-Z0-9]*
    Keywords     mp | agnt | dec | upd | true | false
    file         \"[a-z]+\.[a-z][a-z][a-z]\"
    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]
    Operator     == | != | := | \+ | \* | \- | \= | <= | \-\>
    Separator    \; | \( | \) | \[ | \] | \{ | \}
    ignore       [\0x09\0x0A\0x0D\ ]+
}
```

While LISA automatically infers whether an attribute is inherited or synthesized [36] [46], 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

```
attributes String *.code;
           BufferedWriter PROGRAM.file;
           Hashtable *.outAG,*.inAG;
           Hashtable *.inState, *.outState;
           int *.number, *.value;
           String *.type, *.y;
           String *.file_ip, *.name;
           int *.n;
           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 4.7. 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: $MES\_PLACES ::= MES\_PLACE$ $MES\_PLACES \mid MES\_PLACE$. 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 $MES\_PLACES[0].code = MES\_PLACE.code+$ $''\backslash n'' + MES\_PLACES[1].code$ 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 measuring place $\mathbf{mp}[n_1] \rightarrow \mathbf{agnt}[n_2]S$ 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

```
attributes String *.code;
rule Mes_places {
    MES_PLACES ::= MES_PLACE MES_PLACES compute {
        ... // some rules are omitted
        MES_PLACES[0].code = MES_PLACE.code + "\n" + MES_PLACES[1].code;
    };
    MES_PLACES ::= MES_PLACE compute {
        ... // some rules are omitted
        MES_PLACES.code = MES_PLACE.code;
    };
}

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

Overall, building a DSL compiler using various compiler-generator tools drastically reduce implementation efforts [37], while the maintainability of DSL implementation is also improved [34].

Chapter 5

Practical Experiences

The goal of this section is to acquaint the reader with the practical experiences that were obtained by using of EasyTime. We have therefore selected two case studies (as presented in Section 3.2 and 3.3 of EasyTime applications):

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

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 section.

5.1 World Championship in Ultra double triathlon

Many staff about measuring time in triathlons we learned at the World Championships in the ultra double triathlon in 2009. The rules of this competition are presented in Subsection 3.2. 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 the same subsection as well. Additionally, 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 hours to finish all three disciplines.

5.2 National Championships in a time trial for bicycle

To manipulate the measurements for the case study 2, 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 auto 192.168.225.100;

var ROUND1 := 2;
var INTER1 := 0;
var INTER2 :=0;
var INTER3 := 0;
var BIKE := 0;

mp[1] → agnt[1] {
  dec ROUND1;
  (ROUND1 == 3) → upd INTER3;
  (ROUND1 == 2) → upd INTER2;
  (ROUND1 == 1) → upd INTER1;
  (ROUND1 == 0) → upd BIKE;
}
mp[2] → agnt[1] {
  dec ROUND1;
  (ROUND1 == 3) → upd INTER3;
  (ROUND1 == 2) → upd INTER2;
  (ROUND1 == 1) → upd INTER1;
  (ROUND1 == 0) → upd BIKE;
}
```

As can be seen in Algorithm 8, for 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 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.

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 5.1.

Table 5.1: Translated code for 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)
```

5.3 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.

In the case study 1, 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 mentioned problems have been solved with parallel manual events tracking. Because the number of competitors on the track is 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, the parallel manual measurements still remain at most competitions. However, the most difficult problem for these kinds of competitions are 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.

The case study 2 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 summary, EasyTime allows domain-users capable 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.

Chapter 6

Conclusion

In this thesis we developed domain-specific language named Easytime. We presented entire development of domain-specific language with all stages. This DSL achieved good results in practical solutions. Because of efficiency it has been quickly adopted to anxious races. Usage of DSL Easytime is very simple. Also people who are not programmers can simply use it. As a matter of fact they can quickly learn how to operate with Easytime. In the future we want to create domain-specific modelling language which would more simplify adaptation to various sporting competitions.

Bibliography

[1] Aho, A.V. and Ullman J.D., *The theory of parsing, translation, and compiling (Volume I: Parsing).* Prentice Hall PTR, Upper Saddle River, NY, USA, 1972

[2] Aho, A.V. and Ullman J.D., *The theory of parsing, translation, and compiling (Volume II: Compiling).* Prentice Hall PTR, Upper Saddle River, NY, USA, 1972

[3] Aho, A., V., Lam, M., S., Sethi, R., Ullman, J., D., *Compilers: Principles, Techniques, and Tools with Gradiance.* Prentice Hall, Upper Saddle River, NJ, 2007

[4] Appel, A., W., *Modern compiler implementation in Java.* Cambridge University Press, 1997

[5] Armstrong, L., Jenkins, S. *Every second counts* Broadway, New York, 2003

[6] Arnold, K., Gosling, J., *The Java Programming Language.* Addison-Wesley, 1996

[7] Bakker, J. de, Vink, E. de, *Control flow semantics.* MIT Press, 1996

[8] Barret, W., A., Couch, J., D., *Compiler construction - Theory and practice.* Science research associates, 1979

[9] Bergin, T., J., Gibson, R., G., *History of programming languages.* Addison-Wesley, 1996

[10] Bovet, J., Parr, T., *ANTLRWorks: an ANTLR grammar development environment.* Software practice and experience, 38(12):1305-1332, 2008.

[11] Cardelli, L. and Davies R., *Service Combinators for Web Computings.* IEEE Transactions on Software Engineering, 25:309-316, 1999.

[12] Cardelli, L., Wegner, P., *On understanding types, data abstraction and polymorhism.* ACM Computing Surveys, 17(4), 1985.

[13] Cezzar, R., *A guide to programming languages.* Artech House, 1995

[14] Champion Chip, *ChampionChip2010.* http://www.championchip.com, 20. Dec 2010

[15] Deursen van, A. and Klint, P., *Little languages: little maintenance.* Journal of Software Maintenance, 10(2):75-92, 1998.

[16] Deursen van, A. and Klint, P. and Visser, J., *Domain-specific languages: An annotated bibliography.* ACM Sigplan Notices, 35(6):26-36, 2000.

[17] Deursen van, A. and Klint P., *Domain-specific language design requires feature descriptions* Journal of Computing and Information Technology, 10:1-17, 2002.

[18] Donnelly C., Stallman R., *Bison: The Yacc-compatible parser generator.* Free Software Foundation, December 1990.

[19] Ellis, M., A., Stroustrup, B., *The annotated C++ reference manual.* Addison-Wesley, 1990

[20] Finkenzeller, K., *RFID Handbook.* John Willey, Chichester, UK, 2010

[21] Fister, I., *Jezik CCS za paralelno programiranje* BS.c Thesis, University of Ljubljana, 1983

[22] Fister, I., Jr., Fister, I., Mernik, M., Brest, J., *Design and implementation of domain-specific language EasyTime.* Computer languages, systems and structures, Article accepted for publication 28 April 2011, doi: 10.1016/j.cl.2011.04.001.

[23] Fister, I., Jr., Fister, I., *Measuring time in sports competition with the domain-specific language EasyTime.* Electrotechnical review. Ljubljana, 2011, In press.

[24] Fister, I., Jr., Fister, I., *Uporaba domensko specifičnega jezika pri merjenju časa na športnih tekmovanjih.* Zbornik devetnajste mednarodne Elektrotehniške in računalniške konference ERK 2010, Portorož, Slovenija, 2010, 409-410.

[25] Forouzan, B., *TCP/IP Protocol Suite.* McGraw-Hill, New York, NY, USA, 2009

[26] Glover, B. and Bhatt, H., *RFID Essentials.*, O'Reilly Media, Inc., Sebastopol, USA, 2006

[27] Harford AG, Heuring VP, Main MG, *A new parsing method for non-LR(1) grammars.* Software-practice & experience, 22(5):419-437, 1992.

[28] Henriques, P. and Varanda Pereira, M.J. and Mernik, M. and Lenič, M. and Gray, J. and Wu, H., *Automatic generation of language-based tools using LISA.* IEEE Proceedings - Software Engineering, 152(2):54-69, 2005.

[29] Hudak, P., *Modular domain specific languages and tools.* Proceedings of Fifth International Conference on Software Reuse, 134-142, 1998.

[30] Joel E. Denny, Brian A. Malloy, *The IELR(1) algorithm for generating minimal LR(1) parser tables for non-LR(1) grammars with conflict resolution.* Science of Computer Programming, Special Section on the Programming Languages Track at the 23rd ACM Symposium on Applied Computing - ACM SAC 08 75(11):943-979, 2010.

[31] Johnson, SC., *YACC-yet another compiler compiler.* Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, July 1975

[32] Juels, A., *RFID Security and privacy: A research survey.* IEEE Journal on Selected Areas in Communications, 24(2):394, 2006.

[33] Kieburtz, R.B. and McKinney, L. and Bell, J.M. and Hook, J. and Kotov, A. and Lewis, J. and Oliva, D.P. and Sheard, T. and Smith, I. and Walton, L., *A software engineering experiment in software component generation.* Proceedings of the 18th International Conference on Software Engineering, 542-553, 1996.

[34] Klint, P. and van der Storm, T. and Vinju, J., *On the impact of DSL tools on the maintainability of language implementations.* Proceedings of the Tenth Workshop on Language Descriptions, Tools and Applications, LDTA 10, 2010, 10:1-9.

[35] Klint, P., *A meta-environment for generating programming environments.* ACM transactions on software engineering and methodology, 2(2):176-201, 1993.

[36] Knuth, D., *Mathematical Systems Theory.* Mathematical Systems Theory, 2(2):127-145, 1968.

[37] Kosar, T. and Oliveira, N. and Mernik, M. and Varanda Pereira, M.J. and Črepinšek, M. and da Cruz, D. and Henriques, P.R., *Comparing General-Purpose and Domain-Specific Languages: An Empirical Study.* Computer Science and Information Systems, 7(2):247-264, 2010.

[38] Kosar, T. and Martinez Lopez, P.E. and Barrientos, P.A. and Mernik, M., *A preliminary study on various implementation approaches of domain-specific language.* Information and Software Technology, 50(5):390-405, 2008.

[39] Lesk, M., E., Schmidt, E., *Lex - A lexical analyzer generator.* Bell Laboratories, Murray Hill, New Jersey 07974

[40] LISA, *Lisa2010.* http://labraj.uni-mb.si//lisa, 20. Dec 2010

[41] Mauw, S. and Wiersma, W. and Willemse, T., *Language-driven system design.* International Journal of Software Engineering and Knowledge Engineering, 6(14):625-664, 2004.

[42] Mernik, M. and Heering, J. and Sloane, A., *When and how to develop domain-specific languages.* ACM computing surveys, 37(4):316-344, 2005.

[43] Mernik, M. and Žumer, V., *Incremental programming language development.* Computer Languages, Systems and Structures, 31(1): 1-16, 2005.

[44] Mernik, M., *Ponovno uporabni semantični opis pri načrtovanju in implementaciji programskih jezikov.* Ph.D Thesis, University of Maribor, 1998

[45] Nielson, H.R. and Nielson, F., *Semantics with Applications: A Formal Introduction.* John Wiley & Sons, New York, NY, USA, 1992

[46] Paakki, J., *Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation.* ACM computing surveys, 27(2):196-255, 1995.

[47] Parr T. J., Quong R. W., *ANTLR: A predicated-LL(k) parser generator.* Software-Practice and experience, 25(7):789-810, 1995

[48] Parr, T., *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Pragmatic Bookshelf, 2007

[49] Paulson, L., *A semantics-directed compiler generator.* Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 224-233, 1982

[50] Petschnig, S., *10 Jahre IRONMAN Triathlon Austria.* Meyer & Meyer Verlag, 2007

[51] Porubän, J. and Forgáč, M. and Sabo, M. and Behálek, M., *Annotation Based Parser Generator.* Computer Science and Information Systems, 7(2):291-307, 2010.

[52] Rebernak, D. and Mernik, M. and Wu, H. and Gray, J., *Domain-specific aspect languages for modularizing crosscutting concerns in grammars.* IET Software, 3(3):184-200, 2009.

[53] Rebernak, D., *Aspektno usmerjene atributne gramatike.* Ph.D Thesis, University of Maribor, 2009

[54] RFID Time System, *RFIDTechnology2010.* http://www.rfidtiming.com, 20. Dec 2010

[55] Schobbens, P.-Y. and Heymans, P. and Trigaux, J.-C. and Bontemps, Y., *Generic semantics of feature diagrams.* Computer Networks, 51:456-479, 2007.

[56] Sloane, AM., *Debugging eli-generated compilers with Noosa.* Compiler Construction 1999, Amsterdam, The Netherlands, 1999

[57] Speedy, D.B., Noakes, T.D., Kimber, N.E., Rogers, I.R., Thompson, J.M.D., Boswell, D.R., Ross, J.J., Campbell, R.G.D., Gallagher, P.G., Kuttner, J.A., *Fluid balance during and after an ironman triathlon*, Clinical Journal of Sport Medicine 11(1):44-50, 2001

[58] Sprinkle, J. and Mernik, M. and Tolvanen, J-P. and Spinellis, D., *What kinds of nails need a domain-specific hammer?.* IEEE Software, 26(4):15-18, 2009.

[59] Tanenbaum, A., S., *Computer Networks (Second edition).* Prentice Hall, Englewood Cliffs, NJ, 1989

[60] Thibault, S. and Marlet, R. and Consel, C., *Domain-specific languages: from design to implementation - application to video device drivers generation.* IEEE Transactions on Software Engineering, 25(3):363-377, 1999.

[61] Timing Ljubljana, *Timing2010.* http://www.timingljubljana.si, 20. Dec 2010

[62] Wenzel, K., Wenzel, R. *Bike racing 101* Human Kinetics, Champaign, IL, 2003

[63] Wirth, N., *Algorithms + Data Structures = Programs.*, Prentice Hall PTR, Upper Saddle River, NY, USA, 1978

[64] Wu, H. and Gray, J.G. and Mernik, M., *Grammar-driven generation of domain-specific language debuggers.* Software practice and experience, 38(10):1073-1103, 2008.

[65] Wyk van, E. and Bodin, D. and Gao, J. and Krishnan, L., *Silver: an Extensible Attribute Grammar System.* Science of Computer Programming, 75(1-2):39-54, 2010.

Appendix A

Abbreviations

DSL - domain-specific language
GPL - general-purpose language
GUI - graphical user interface
RFID - radio frequency identification
LCD - liquid crystal display
LAN - local area network
TCP/IP - transmission control protocol and the internet protocol
MP - measuring place
CP - control point
ITU - international triathlon union
ETU - european triathlon union
IUTA - international ultra triathlon association
UCI - international cycling union
WTC - world triathlon corporation

Appendix B

EasyTime source code

```
language EasyTime
{
  lexicon
  {
    Int         [0-9]+
    Id          [a-zA-Z][a-zA-Z0-9]*
    Keyword1    mp | agnt
    Keyword2    dec
    Keyword4    upd
    Keyword3    true | false
    file        \"[a-z]+\.[a-z][a-z][a-z]\"
    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]
    Operator    == | != | := | \+ | \* | \- | \= | <= | \-\>
    Separator   \; | \( | \) | \[ | \] | \{ | \}
    ignore      [\0x09\0x0A\0x0D\  ]+
  }

  attributes String *.code;
    BufferedWriter PROGRAM.file;
    Hashtable *.outAG,*.inAG;
    Hashtable *.inState, *.outState;
    int *.number, *.value;
    String *.type, *.y;
    String *.file_ip, *.name;
    int *.n;
    boolean *.ok;

  rule Start  {
    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";
      PROGRAM.file = open_append ("EasyTime_code.txt",
                          PROGRAM.code, "\n");
    PROGRAM.outAG = AGENTS.outAG;
    PROGRAM.outState = DECS.outState;
  };
}

rule 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;
  };
}

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

rule Decs {
  DECS ::= DECS DEC compute {
    DECS[1].inState = DECS[0].inState;
    DECS[0].outState = put(DECS[1].outState, DEC.name, DEC.value);
  };
  DECS ::= epsilon compute {
    DECS.outState = DECS.inState;
  };
}

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

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() + ")";
  };
}

rule Stmts {
  STMTS ::= STMT STMTS compute {
    STMT.n = STMTS[0].n;
    STMTS[1].n = STMTS[0].n;
    STMT.inAG = STMTS[0].inAG;
    STMTS[1].inAG = STMTS[0].inAG;
    STMT.inState = STMTS[0].inState;
    STMTS[1].inState = STMTS[0].inState;
    STMTS[0].ok = STMT.ok && STMTS[1].ok;
    STMTS[0].code = STMT.code + "\n" + STMTS[1].code;
  } |
  STMT compute {
    STMT.n = STMTS[0].n;
    STMT.inAG = STMTS[0].inAG;
    STMT.inState = STMTS[0].inState;
    STMTS.ok = STMT.ok;
    STMTS.code = STMT.code;
  };
}

rule Statement {
  STMT ::= dec #Id \; compute {
    STMT.y =  ((Agent)STMT.inAG.get(STMT.n)).getType().equals("manual") ?
  "accessfile(" + ((Agent)STMT.inAG.get(STMT.n)).getFile_ip() + ")" :
  "connect(" + ((Agent)STMT.inAG.get(STMT.n)).getFile_ip() + ")";
    STMT.ok =  STMT.inState.containsKey(#Id.value());
    STMT.code = " FETCH " + #Id.value() +" DEC STORE " + #Id.value();
  };
  STMT ::= upd  #Id \; compute {
    STMT.y =  ((Agent)STMT.inAG.get(STMT.n)).getType().equals("manual") ?
  "accessfile(" + ((Agent)STMT.inAG.get(STMT.n)).getFile_ip() + ")" :
  "connect(" + ((Agent)STMT.inAG.get(STMT.n)).getFile_ip() + ")";
    STMT.ok =  STMT.inState.containsKey(#Id.value());
    STMT.code = " FETCH " + STMT.y + " STORE " + #Id.value();
  };
  STMT ::= #Id \:\= EXPR \; compute {
    STMT.y =  ((Agent)STMT.inAG.get(STMT.n)).getType().equals("manual") ?
  "accessfile(" + ((Agent)STMT.inAG.get(STMT.n)).getFile_ip() + ")" :
  "connect(" + ((Agent)STMT.inAG.get(STMT.n)).getFile_ip() + ")";
    STMT.ok =  STMT.inState.containsKey(#Id.value()) && EXPR.ok;
    STMT.code = EXPR.code + " STORE " + #Id.value();
    EXPR.inState = STMT.inState;
  };
  STMT ::= \( LEXPR \) \-\> STMT compute {
    STMT.y =  ((Agent)STMT.inAG.get(STMT.n)).getType().equals("manual") ?
  "accessfile(" + ((Agent)STMT.inAG.get(STMT.n)).getFile_ip() + ")" :
  "connect(" + ((Agent)STMT.inAG.get(STMT.n)).getFile_ip() + ")";
    STMT[1].n = STMT[0].n;
    STMT[1].inAG = STMT[0].inAG;
    STMT[1].inState = STMT[0].inState;
    STMT[0].ok = STMT[1].ok && LEXPR.ok;
    LEXPR.inState = STMT[0].inState;
    STMT.code = LEXPR.code + "BRANCH(" + STMT[1].code + ", NOOP)";
  };
}

rule Lexpr {
  LEXPR ::= true compute {
    LEXPR.code = "TRUE ";
    LEXPR.ok = true;
  };
  LEXPR ::= false compute {
    LEXPR.code = "FALSE ";
    LEXPR.ok = true;
  };
  LEXPR ::= EXPR == EXPR compute {
    LEXPR.code = EXPR[1].code + " " + EXPR[0].code + " EQ ";
    LEXPR.ok = EXPR[0].ok && EXPR[1].ok;
    EXPR[0].inState = LEXPR.inState;
    EXPR[1].inState = LEXPR.inState;
  };
  LEXPR ::= EXPR != EXPR compute {
    LEXPR.code = EXPR[1].code + " " + EXPR[0].code + " NEQ ";
    LEXPR.ok = EXPR[0].ok && EXPR[1].ok;
    EXPR[0].inState = LEXPR.inState;
    EXPR[1].inState = LEXPR.inState;
  };
}

rule Expr {
EXPR ::= #Int compute {
  EXPR.code = "PUSH " + #Int.value() + " ";
    EXPR.ok = true;
  } |
#Id compute {
    EXPR.code = "FETCH " + #Id.value()+ " ";
    EXPR.ok = EXPR.inState.containsKey(#Id.value());
};
```

```
 }

method M_Agent {
  class Agent {
    int number;
    String type;
    String file_ip;
    Agent ( int number, String type, String file_ip) {
      this. number = number;
      this. type = type;
      this. file_ip = file_ip;
    }
    public String toString(){
      return "(" + this.number + ", " + this.type + ", " + this.file_ip + ")";
    }
    public int getNumber(){
      return this.number;
    }
    public String getType(){
      return this.type;
    }
    public String getFile_ip(){
      return this.file_ip;
    }
  }
}

method M_Insert {
  import java.util.*;
  Hashtable insert (Hashtable aAgents, Agent aAgent) {
    aAgents = (Hashtable)aAgents.clone();
    Agent hAgent=(Agent)aAgents.get(aAgent.getNumber());
    if (hAgent==null)
      aAgents.put(aAgent.getNumber(), aAgent);
    else
      System.out.println("Agent" + aAgent.getNumber() + "is already defined");
    return aAgents;
  } // java method
}// Lisa method

method Environment {
  import java.util.*;
  public Hashtable put(Hashtable env, String name, int val) {
    env = (Hashtable)env.clone();
    env.put(name, new Integer(val));
    return env;
  } // java method
} // LISA method

method M_append {
  import java.io.*;
  import java.lang.*;
  import java.util.*;
  BufferedWriter append(String fileName, String attribute, String separator) {
      try {
        BufferedWriter outFile = new BufferedWriter(new FileWriter(fileName,
          true));
        outFile.write(attribute + separator);
        outFile.close();
        return outFile;
      } catch (IOException e) {
        e.printStackTrace();
      }
      return null;
    } // java method
  }   // Lisa method

  method M_open {
    import java.io.*;
    import java.lang.*;
    import java.util.*;
    BufferedWriter open(String fileName, String attribute, String separator) {
      try {
        BufferedWriter outFile = new BufferedWriter(new FileWriter(fileName));
        outFile.write(attribute + separator);
        outFile.close();
        return outFile;
      } catch (IOException e) {
        e.printStackTrace();
      }
      return null;
    } // java method
  }   // Lisa method

  method M_open_append {
    import java.io.*;
    import java.lang.*;
    import java.util.*;
    BufferedWriter open_append(String fileName, String attribute, String separator) {
      try {
        BufferedWriter outFile;
        File f = new File(fileName);
        if (f.exists())
          outFile = new BufferedWriter(new FileWriter(fileName, true));
        else
          outFile = new BufferedWriter(new FileWriter(fileName));
        outFile.write(attribute + separator);
        outFile.close();
        return outFile;
      } catch (IOException e) {
        e.printStackTrace();
      }
      return null;
    } // java method
  }  // Lisa method
}
```

Dodatek C

Razširjeni povzetek v slovenskem jeziku

C.1 Uvod

V diplomski nalogi se ukvarjamo s problemom merjenja časa na športnih tekmovanjih. V zadnjem obdobju množične športne prireditve, kot npr. triatloni, maratoni, kolesarske dirke, ipd., postajajo vse bolj popularne. Zaradi velikega števila udeležencev je klasično merjenje časa, t.j. s štoparico, nemogoče. S pojavom tehnologije RFID se je merjenje poenostavilo. Poleg omenjene tehnologije pa potrebujemo tudi računalniški sistem, ki omogoča krmiljenje merilnih naprav. V ta namen smo razvili domensko specifični jezik Easytime.

Domensko specifični jeziki (angl. Domain-Specific Language, krajše DSL) so jeziki, prikrojeni aplikacijski domeni. Imajo tudi veliko prednosti v primerjavi s splošno namenskimi jeziki (angl. General-Purpose Language, krajše GPL). Njihove glavne prednosti se jasno vidijo v njihovi izrazni moči in s tem v večji produktivnosti, enostavni uporabi, enostavni verifikaciji ter optimizaciji.

C.2 Programski jeziki

Programski jezik je vmesnik med človekom in računalnikom. Programski jeziki so ena izmed najbolj raziskanih področij v računalništvu. Trenutno poznamo že preko 3500 programskih jezikov. Programske jezike lahko razdelimo na GPL in DSL jezike. Z GPL lahko pišemo programe za poljubno domeno. Primeri teh jezikov so:

- C/C++,
- Java,

- Ruby,
- PHP.

Po drugi strani so DSL prikrojeni določeni problemski domeni. DSL je majhen, ponavadi deklarativen jezik, ki je namenjen točno določenemu problemu. Ti jeziki imajo svoje prednosti in slabosti. Glavne prednosti so:

- lažje programiranje,
- ponovna uporaba,
- lažja verifikacija,
- uporabniško programiranje.

Imajo pa tudi nekaj slabosti:

- stroški razvoja,
- stroški učenja novih uporabnikov,
- slabša učinkovitost,
- omejena uporabnost,
- omejena dostopnost.

Če želimo program pognati na računalniku, ga moramo najprej prevesti. Prevajanje sestoji iz treh delov:

1. leksikalne analize,
2. sintaktične analize,
3. semantične analize.

V prvi fazi poiščemo leksikalne simbole. Glavna naloga te faze je v izvornem programu poiskati terminalne simbole za sintaktični analizator. V naslednji fazi poiščemo strukturo stavkov jezika. Izhod te faze je opis sintaktične strukture originalnega problema. V tej fazi se ukvarjamo s pravilnim zaporedjem stavkov in nas ne zanima, če je pomen stavkov pravilen (semantika). S problemom semantike se ukvarjamo v naslednji fazi, t.j. semantični analizi.

Izvajalno kodo za realni računalnik generira generator kode.

C.3 Merjenja časa na športnih tekmovanjih

Merjenje časa lahko izvajamo ročno ali avtomatsko. Pri ročnem merjenju merimo z ročno štoparico in rezultate zapisujemo na list. Pri avtomatskem merjenju merimo čas s pomočjo merilne naprave, ki deluje na podlagi RFID tehnologije. RFID tehnologija je lahko pasivna ali aktivna. Vsak tekmovalec ima na sebi pritrjen čip z unikatno identifikacijsko številko. Ko tekmovalec prečka merilno mesto, antensko polje inducira čip tekmovalca in pošlje identifikacijsko številko merilni napravi. Ta dogodek merilna postaja shrani v pomnilnik in hkrati pošlje tudi na računalniški sistem. Glavne kontrolne funkcije merilne naprave so:

- branje realnega časa,
- nastavljanje realnega časa,
- start registracije dogodkov,
- konec registracije dogodkov,
- branje dogodkov.

C.3.1 Triatloni

Triatlon je relativno mlad šport. Njegovi začetki segajo v leto 1975, ko so v ZDA prvič izvedli triatlonsko tekmovanje. Gre za olimpijsko panogo, ki sestoji iz treh disciplin:

1. plavanja,
2. kolesarjenja,
3. teka.

Discipline potekajo brez prekinitve, v skupni čas pa se štejeta tudi časa obeh menjav, t.j. ko tekmovalec preide iz vode na kolo in s kolesa na tek.

Najbolj značilni so naslednji triatloni:

- sprint triatlon (750 m plavanje / 20 km kolo / 5 km tek),
- olimpijski triatlon (1500 m plavanje / 40 km kolo, 10 km tek),
- polovični Ironman (1900 m plavanje, 90 km kolo, 21.1 km tek),
- Ironman (3800 m plavanje, 180 km kolo, 42.2 km tek).

Na triatlonu dobi vsak tekmovalec svojo startno številko, ki jo nosi okrog pasu. Zaradi lažje identifikacije je tekmovalec še dodatno označen s številkami na ramenu in stegnu. Kolo in ostalo opremo, potrebno za kolesarjenje in tek, tekmovalec pripravi v menjalnem prostoru pred začetkom tekmovanja.

Če je le mogoče, startajo vsi udeleženci hkrati. Ko tekmovalec premaga plavalno progo, steče v menjalni prostor. Tam obuje kolesarske čevlje, obleče majico, nadene čelado, vzame kolo in prične s kolesarjenjem. Ali je vožnja v zavetrju sotekmovalcev dovoljena, določi organizator tekme, praviloma je dovoljena na vseh krajših tekmah. Po prevoženi kolesarski progi tekmovalec odloži kolo, sname čelado, se preobuje in začne teči. Po končanem teku zabeležimo dosežen čas, na podlagi katerega se izračuna uvrstitev v absolutni konkurenci in v njegovi starostni kategoriji.

C.3.2 Kronometer

Kronometer je tip kolesarske dirke, kjer tekmovalec vozi sam in ne v skupini. Tekmovalci na kronometru navadno startajo v intervalih po ene ali dveh minuti. Vožnja v zavetrju je v tej disciplini prepovedana.

C.4 EasyTime

EasyTime je domensko specifični jezik, ki smo ga razvili z namenom povečanja fleksibilnosti merilnega sistema. Z njim lahko enostavno generiramo kodo za merilno napravo in jo preprosto prilagajamo potrebam tekmovanja. Sintaksa jezika EasyTime je zelo enostavna in jo lahko uspešno obvladuje tudi tisti, ki ni programer.

C.4.1 Razvoj

Razvoj tega jezika sestoji iz petih faz, t.j.:

1. analize domene na podlagi diagrama lastnosti (angl. Feature Diagrams),
2. abstraktne sintakse,
3. formalne semantike,
4. definicije abstraktnega stroja in
5. implementacije.

V prvi fazi podrobno analiziramo aplikacijsko domeno. Poiščemo vse značilnosti te aplikacijske domene. Rezultat te analize je diagram lastnosti. Ta diagram grafično ponazori odvisnosti med značilnostmi. Abstraktna sintaksa definira sintakso tega jezika. S pomočjo formalne semantike pa določimo pomen posameznih stavkov. Določimo kakšen pomen bodo imele posamezne strukture v stavkih. Naslednja faza je definicija abstraktnega stroja. Tukaj definiramo arhitekturo abstraktnega stroja, na katerem se izvaja prevedena koda. Abstraktni stroj v našem primeru sestoji iz četvorke

$<c, e, db, i>$, kjer pomenijo: c je kodni segment, e je računski sklad, db je podatkovna baza, i je startna številka tekmovalca. Zadnja faza je implementacija našega jezika s pomočjo generatorja prevajalnikov/interpreterjev LISA. Tukaj s pomočjo generatorja kode generiramo pravila za krmiljenje agentov, ki berejo dogodke z merilnih naprav sprotno.

C.5 Uporaba v praksi

EasyTime smo uporabili tudi na realnih primerih iz prakse. S pomočjo tega jezika smo merili čas na dveh velikih tekmovanjih:

- Svetovnem pokalu v dvojnem ultra triatlonu 2009 in
- Državnem prvenstvu v kronometru 2010.

Merjenje prvega tekmovanja je bilo zahtevno, predvsem zaradi dolgega trajanja same prireditve in velikega števila tekmovalcev. Proga na tekmovanju je bila razbita v kroge. Tako so morali tekmovalci preplavati 20 krogov, prevoziti 105 krogov in preteči 55 krogov. Merjenje je potekalo ročno in avtomatsko. Ker je merilna tehnologija za merjenje plavanja še zelo draga, smo merili plavanje ročno. Avtomatsko smo merili kolesarski in tekaški del, ter tranzicijo 2.

Drugo tekmovanje je bilo enostavnejše za merjenje, saj je bilo potrebno izmeriti samo končni čas tekmovalca.

C.6 Zaključek

V tej diplomski nalogi smo razvili domensko specifični jezik. Pokazali smo celoten razvoj od načrtovanja do implementacije z vsemi vmesnimi fazami. Ta DSL se je v praksi pokazal kot zelo učinkovit, saj se z njim da hitro spreminjati konfiguracije merilnih naprav potrebnih za registracijo dogodkov. Enostavno ga je tudi prilagoditi potrebam različnih športnih tekmovanj. Tudi tisti, ki niso programerji ali računalničarji se lahko hitro naučijo uporabljati ta jezik. V prihodnosti bi radi razvili tudi domensko specifični modelirni jezik, ki bi še bolj poenostavil pisanje programov v EasyTime.

Fakulteta za elektrotehniko, računalništvo in informatiko

Smetanova ulica 17
2000 Maribor

IZJAVA O ISTOVETNOSTI TISKANE IN ELEKTRONSKE VERZIJE DIPLOMSKEGA DELA IN OBJAVI OSEBNIH PODATKOV DIPLOMANTOV

Ime in priimek diplomanta-tke: Iztok Fister

Vpisna številka: E1007000

Študijski program: FERI – RIT UNI RAČUNALNIŠTVO IN INFORMACIJSKE TEHNOLOGIJE

Naslov diplomskega dela: DOMAIN – SPECIFIC LANGUAGE FOR TIME MEASURING ON SPORT COMPETITIONS

Mentor: prof. dr. Marjan Mernik

Somentor: prof. dr. Barrett R. Bryant

Podpisani-a IZTOK FISTER izjavljam, da sem za potrebe arhiviranja oddal elektronsko verzijo zaključnega dela v Digitalno knjižnico Univerze v Mariboru.
Diplomsko delo sem izdelal-a sam-a ob pomoči mentorja. V skladu s 1. odstavkom 21. člena Zakona o avtorskih in sorodnih pravicah (Ur. l. RS, št. 16/2007) dovoljujem, da se zgoraj navedeno zaključno delo objavi na portalu Digitalne knjižnice Univerze v Mariboru.

Tiskana verzija diplomskega dela je istovetna elektronski verziji, ki sem jo oddal za objavo v Digitalno knjižnico Univerze v Mariboru.

Podpisani izjavljam, da dovoljujem objavo osebnih podatkov vezanih na zaključek študija (ime, priimek, leto in kraj rojstva, datum diplomiranja, naslov diplomskega dela) na spletnih straneh in v publikacijah UM.

Datum in kraj: 3.6.2011, Maribor

Podpis avtorja: Iztok Fister

UNIVERZA V MARIBORU

FERI
Fakulteta za elektrotehniko, računalništvo in informatiko

Smetanova ulica 17
2000 Maribor

IZJAVA O USTREZNOSTI DIPLOMSKEGA DELA

Podpisani mentor prof. dr. Marjan Mernik izjavljam, da je

študent Iztok Fister izdelal diplomsko

delo z naslovom: Domain – specific language for time measuring on sport competitions

v skladu z odobreno temo diplomskega dela, Navodili o pripravi diplomskega dela in mojimi navodili.

Datum in kraj: 3.6.2011, Maribor

Podpis mentorja: [illegible]

Univerza v Mariboru

Fakulteta za elektrotehniko, računalništvo in informatiko

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani/-a IZTOK FISTER,

z vpisno številko E1007000,

sem avtor/-ica diplomskega dela z naslovom:

DOMAIN-SPECIFIC LANGUAGE FOR TIME MEASURING ON SPORT COMPETITIONS

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom (naziv, ime in priimek)

  prof. dr. Marjan Mernik

  in somentorstvom (naziv, ime in priimek)

  prof. dr. Barrett R. Bryant

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela

- soglašam z javno objavo elektronske oblike diplomskega dela v DKUM.

V Mariboru, dne 5.6.2011 Podpis avtorja/-ice: Iztok Fister