

Language-Driven System Design

S. Mauw, W.T. Wiersma, T.A.C. Willemse
Department of Mathematics and Computer Science,
Eindhoven University of Technology,
P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands.

sjouke@win.tue.nl, w.t.wiersma@stud.tue.nl, timw@win.tue.nl

Abstract

Studies have shown significant benefits of the use of Domain-Specific Languages. However, designing a DSL still seems to be an art, rather than a craft following a clear methodology. In this paper, we discuss a first step towards a methodology for designing such languages. The presented approach, which is referred to as the Language-Driven Approach, is rooted in formal techniques and independent of accepted software engineering process models. We illustrate the approach with a small and instructive case study.

1 Introduction

The complexity of software has steadily increased over the past decades. This necessitated the development of techniques to master the difficulties and problems due to this increase. Over the years, several process models have been introduced, for structuring the design process of software. Many different variants of these process models exist, e.g. Boehm's spiral model [2], and the incremental model [28].

Although the differences between these models can be large, all models prescribe a partitioning of the software engineering process into a number of stages. These stages are distinguished on the basis of the activities that have to be conducted. The order in which these stages must be addressed, along with the deliverables that can be expected in each stage are prescribed by the process model.

In theory, most process models are fairly general with respect to the means that must be used to obtain the deliverables in each stage, although some of the process models define best practices for a number of stages. In practice, however, the tools that are used are often determined by external influences, such as a company's policies. This traditional approach to software engineering focuses mainly on a software product that must be developed. Alternatively, the focus could be on a language (or a class of languages)

that is tailored to the software product. These languages are often referred to as *Domain-Specific Languages*.

Domain-Specific Languages (DSLs) have emerged as a tool for tackling the complexity of software development projects. Many studies (e.g. [9]) have shown significant benefits of using DSLs in software development. Noteworthy are the increase in the reliability and the maintainability of the produced software, but also improved reusability of a software product's code and design (see e.g. [9, 12]).

Although the use of DSLs and their benefits have been well documented, there is only little literature available on the relation between process models and software engineering methods on the one hand and the use of DSLs on the other hand. Moreover, developing a DSL still seems to be an ad-hoc process, rather than a clearly defined process with a clearly defined methodology. In order to support the acceptance of the ideas and concepts of DSLs, a clear methodology needs to be defined. This methodology must focus on the aspects for developing a DSL, with a clear emphasis towards the intended application of the DSL for a specific problem domain.

In this paper, we discuss the *language-driven approach* to software engineering. This approach can be considered as a first step to a general methodology for designing a DSL. The emphasis of the approach is on the interplay between standard software engineering methods and its best practices, various process models and the concepts and ideas behind DSLs. The key issue in this approach is the focus on the development of a suitable DSL for writing (part of) a software product, rather than the development of the software product itself.

The language-driven approach combines and extends well-known and accepted methods from software engineering. It inherits techniques and concepts from the area of *formal methods* (in its broadest sense), but also the basic ideas and notions behind various *programming paradigms* are incorporated. Moreover, the language-driven approach relies heavily on the expertise and the techniques that are needed to conduct a *domain analysis*. Also for these tech-

niques, there is ample literature available (e.g. [5, 19]).

A major reason for incorporating techniques from specialist areas such as formal methods is our firm believe that a language consists not only of a syntax, but also of an (unambiguous) semantics. Moreover, every language has its own pragmatics that needs to be clear to its users. The techniques that have been studied and developed in the area of formal methods are essential in analysing and defining an understandable language, its syntax, its semantics and its pragmatics. The tool ASF+SDF [13, 25], for instance, can be used to define and analyse the semantics of programming languages. Apart from this, the use of a formal semantics for validating programs is a key issue in formal methods' research.

The traditional scalability problems often encountered when applying formal methods in the design of software are not likely to be an issue in the language-driven approach. The problem of scalability is often caused by the large gap between the methods that are used to describe a software product and the key concepts of the software product. For the language-driven approach, this gap is relatively small, as the software product is defined in terms of its natural concepts.

In this paper, we introduce and discuss the phases of the language-driven approach. In each phase, we mention the deliverables needed and the techniques for producing them. Additional information is given for selecting alternative techniques or paradigms. This is needed if special requirements are posed on the deliverables, like the need for formal verification.

The abstract ideas in this paper are illustrated with a case study. The case study discusses the design of a language for controlling traffic lights at a junction.

This paper is organised as follows. Section 2 describes the language-driven approach in detail. In Section 3 we discuss related work and research that has already been conducted in this area. Section 4 discusses a case study. An extended and more detailed version of this paper appeared as [16].

2 The language-driven approach

In this section we discuss and elaborate on the ingredients that play a role in the language-driven approach. These ingredients can all be found in literature. Wherever possible, we provide pointers to the literature. In our discussion, we emphasise on the formal aspects of the design approach.

2.1 Overview and rationale

In many ways, the language-driven approach resembles a standard software development process. However, there are some differences. These differences are due to the fact

that in the language-driven approach, the design is centred around the development of a (formal) language. The language itself, which will be a Domain-Specific Language (DSL), constitutes the major result of the design process. Moreover, the centre of activities in the software design process shifts to earlier stages, such as the user requirements and specification phases. This has several well-known advantages, e.g. reduction of the time-to-market, early detection of errors, etc.

The development of the DSL is described by a collection of deliverables. These deliverables include the definition of its syntax and semantics, and define appropriate tool support. The language-driven design approach can be integrated in today's software process models, such as the waterfall model [21] or the spiral model [2]. Using proven software process models assists the design of the language in a structured way.

Figure 1 shows an overview of the artifacts produced during the development process. We refrain from using a specific process model for the development of the deliverables. In practice, the final product will be the result of several iterations of the development process.

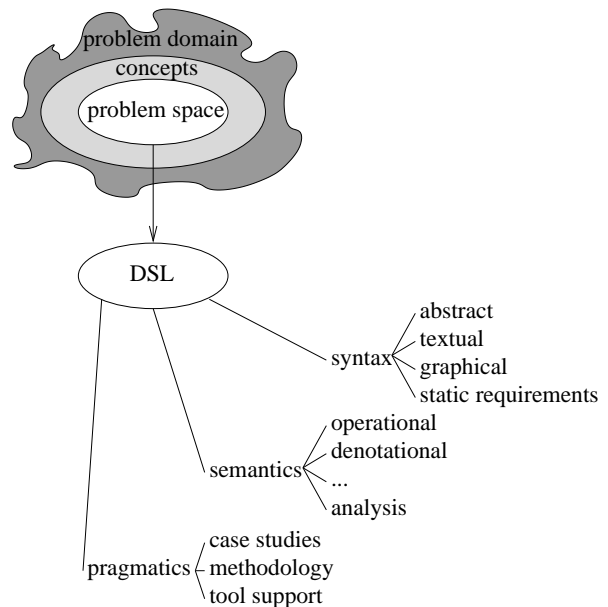


Figure 1. The language-driven approach.

Due to the inherent causalities and dependencies between the deliverables, a natural ordering is imposed on their development. This ordering is made explicit in the distinction between the following stages in the design process:

1. identification of the *Problem Domain*,
2. identification of the *Problem Space*,

3. formulation of the *Language Definition*.

Note that due to the iterative nature of many process models, the actual order in which these stages are addressed is not fixed, even though there is a clear and intended dependence between the stages. Also, these stages should not be considered to be *atomic*, i.e. it is possible to start a parallel trajectory on the next stage if sufficient information from another stage is available.

In the subsequent sections, the three stages are explained in greater detail.

2.2 Identification of the Problem Domain

The identification of the problem domain is the first stage in the language-driven approach. Rather than focusing on the single problem that needs to be investigated, the language-driven approach focuses on a class of problems stemming from a common problem domain. A thorough domain analysis is necessary to give a complete and precise definition of all essential concepts in the problem domain. Fortunately, there are many existing techniques that support a domain analysis and domain specification. A proper demarcation of the problem domain is vital, as all subsequent artifacts depend on the concepts captured and described by the problem domain. We refer to [19, 5] for an overview of best practices and techniques for conducting the domain analysis.

The problem domain can be (and usually is) much larger –both in generality of the concepts and in the number of concepts– than is strictly needed to solve the actual problem. This has several advantages, e.g. reuse of domain knowledge and self-containment of the problem domain. A restricted problem domain often implies that some design decisions have already been made.

2.3 Identification of the Problem Space

The second stage of the language-driven approach is the identification of the problem space. As already remarked, the previously determined problem domain is a mostly exhaustive collection of all concepts used and related to the actual problem. As mentioned, this has definite advantages; however, for solving the actual problem the problem domain is often too general and too large. Therefore, a restriction of the problem domain is necessary.

A first observation is that in order to provide a direction for solving the actual problem, *design decisions* must be made. These design decisions possibly lead to concepts that have not been identified in the problem domain. The concepts thus introduced play a pivotal role in solving the actual problem. The fact that these concepts are not part of the problem domain follows straightforwardly from the

fact that the identification of the problem domain is not a design-driven activity.

A second observation is that with respect to the actual problem, the problem domain contains some inherent redundancy. Whenever concepts do not appear to play any part in solving the actual problem, we can consider them irrelevant for solving this problem. Hence, we only need to consider the concepts that are relevant to our problem. The last observation is that with respect to the actual problem, many of the identified concepts are too general. Therefore, a natural second classification of the concepts is to make a distinction between the concepts that can be constrained in some sense, and the concepts that are inherently variable.

These observations lead to a classification of all concepts into the following three categories:

- concepts that are *irrelevant* to the actual problem,
- concepts that are *variable*, and
- concepts that have been *fixed* for the actual problem.

As already remarked, a concept is *irrelevant* whenever it does not play any part in the solution to the actual problem. Moreover, concepts can often be classified as irrelevant due to abstraction and aggregation.

A concept is called *variable* whenever it varies depending on the actual problem instance, or it varies within the actual problem instance. The variable concepts that vary depending on the actual problem instance can often be considered as problem parameters; every (allowed) instantiation of the problem parameters calls for its own solution. These problem parameters are the part of the actual problem which will be specified by means of an expression in the DSL. As a result, this collection of variable concepts determines the syntax of the language. The variable concepts that vary within the actual problem determine the behaviour of the system. In an operational semantics, these concepts reappear as a part of the state space. They cannot be specified by means of expressions in the syntax.

The category of the *fixed* concepts consists of the concepts which are identical for all problems considered. This can be caused by the fact that the notion is inherently constant (e.g. a law of nature), but more often it concerns a variable notion which is restricted to simplify the problem setting.

The class containing all variable concepts and all fixed concepts is referred to as the *problem space*. Obviously, this part of the problem space is more concrete compared to the problem domain. This, however, reduces the complexity of the basic notions that are relevant to the actual problem, while still retaining enough information to describe the actual problem accurately.

2.4 Formulation of the Language Definition

The design of the language concretises the notions and concepts that can be found in the problem domain. In this section, we discuss the constituent parts of a DSL. We advocate a formal treatment for the specification of both the syntax and semantics of a DSL.

The language definition stage is divided into three sub-phases in which the *syntax*, the *semantics* and the *pragmatics* of the language are defined.

2.4.1 Syntax

The appearance of a language is defined by means of its *syntax*. In the language-driven approach, the constructs of the language are related to the concepts that have been identified in the domain space. The syntax of the language consists of expressions of the variable concepts that have been identified in the problem space. Unlike the variable concepts, fixed concepts are not defined in the syntax.

The syntax serves several purposes. It supports the user in expressing the properties of the problems the user wants to solve using the language. Second, the semantics is based on the syntactical expressions. Moreover, the language constructs serve as a basis for applying analysis techniques on both the language and the problems described using the language.

The format of the language is constrained in several ways. Most importantly, it must be susceptible to interpretation and/or transformation by means of a computer. Moreover, the syntax is often constrained by a number of generally accepted requirements such as readability and writability of the language constructs. Finally, we stress the importance of choosing syntax expressions for which the mathematical semantics correspond to the intuitive semantics.

In general, a language can have one or more syntactical descriptions. These descriptions depend on the required use of the language. Three of the more popular formats are the following:

- the *abstract* syntax,
- the *textual* or *linear* syntax,
- the *graphical* syntax.

The *abstract* syntax is often used to express all semantically relevant information in a minimal way (e.g. without keywords or superfluous transitions in the defining grammar). Moreover, the data structure that is used by computers to store the information that is obtained while processing programs is often strongly related to the abstract syntax.

The *textual* or *linear* syntax is the description that is most often encountered in language descriptions. The information in the textual syntax is essentially the same as in the abstract syntax. However, the textual syntax is easier to read and use. This is best exemplified by constructs such as the *if-then-else* construct. This construct will have all the appropriate keywords in the textual syntax, but in the abstract syntax it will simply be a triplet.

The third format, the *graphical* syntax, is gradually gaining popularity. Graphical, or *visual* languages have several benefits over linear languages, such as the ability to express spatial properties or complex relations in a more intuitive fashion. The general availability of graphical workstations makes it possible for regular users to work with visual languages. Although a graphical syntax may seem capable of expressing more than the textual or abstract syntax, the semantically relevant information should be identical.

The abstract syntax and the textual syntax can be partly defined by means of BNF grammars, or BNF-like grammars (i.e. BNF grammars enhanced with simple (mathematical) structuring mechanisms such as sets or records). The most popular way of defining the graphical language is by means of *graph grammars* [20].

In most cases BNF-like grammars are not expressive enough to exactly describe which expressions in the language are *well-formed*. Context-sensitive properties, such as the *declare-before-use* property of variables, must be expressed in a different way. These additional requirements on well-formedness of expressions are often referred to as the *static semantics*. As this title is somewhat misleading, since it deals with syntactical properties of the language, we prefer to use the term *static requirements*. Most often, attribute grammars [14] are used for specifying the static requirements, but logical predicates can also be applied.

To conclude this section on syntax, we mention that there are several other syntactical aspects which can be specified. A requirement that is often posed on expressions in a graphical languages is to have a layout that is transparent to tools. It is not likely that the detailed layout will have any semantical meaning, so one may not expect that the textual syntax is capable of expressing such properties. This is resolved either by extending the textual syntax with information that is semantically irrelevant, or by defining an additional syntax which is tailored to expressing such details. The latter approach is often called a *tool interchange format* (see e.g. the Common Interchange Format CIF for the SDL language [11]).

2.4.2 Semantics

A semantics for a language is a mathematical model that reflects the intended computational behaviour of expressions in the language. In essence, one can classify language com-

ponents in two ways:

- Components dealing with dynamic behaviour,
- Components describing purely static information.

This distinction is also reflected in the semantics of the language.

As for general-purpose languages, various approaches exist to defining a semantics for a language. The choice of a suitable semantical approach depends largely on the characteristics of the language itself, i.e. the class to which the language belongs. However, the practical use of the semantics is important as well. Dependent on the type of semantics, techniques such as behavioural analysis, invariant analysis or simulation of expressions in the language can be used. Most designers of a language are biased towards certain approaches.

Basic to most semantical approaches is the existence of a semantical domain. Such a domain often consists of a set (or collection of sets) with an additional structure defined by relations. Expressions in the language relate to entities in this domain and obtain their meaning via the properties of the related entities.

Although the different approaches are all variations on a similar theme, each of these approaches emphasises on a different aspect and has its own benefits. We subsequently give a short overview of the main advantages of three commonly used approaches in the next paragraphs. For an overview of other semantical approaches such as Abstract State Machines and attribute grammars, etc. see e.g. [7, 23].

Operational semantics Operational semantics is used to give meaning to the dynamic part of a language. It is centred around the notions of a state and the transitions between the states (see e.g. [8]). The transitions between the state can be described by means of a transition function. Various ways exist for defining the operational semantics, e.g. by means of SOS-rules [18].

The operational semantics of a language is quite close to the intuition behind the language. It is often used by implementors. An operational semantics provides the means for performing simulations of expressions in the language by considering runs of the transition function. This is useful in areas of testing, or even automated testing. Moreover, there is also the possibility of analysing the transition graph that is induced by a language expression. This is often used in verification efforts. Finally, tools, such as ASF+SDF [13, 25] or MAUDE [3] may be used to develop prototypes of the language.

Denotational semantics A denotational semantics is centred around the idea of a mathematical function that describes the meaning of an expression by means of a transla-

tion to a well-understood mathematical model, as described in the beginning of this section (see e.g. [22]). Its virtue is the use of this mathematical model for the analysis and comparison of expressions in the language.

The theory of the denotational semantics is mathematically very rigorous. It is often used by language designers, as it precisely expresses the requirements on the language. Techniques to prove two expressions in the language equivalent are easily formulated using the underlying mathematical model. Such techniques can also be automated, using theorem provers.

Axiomatic semantics The axiomatic semantics is given by means of a number of axioms relating expressions in the language. The axioms can be based on some underlying logic. The axiomatic semantics is often used in combination with a denotational or operational semantics to provide for an underlying mathematical model and a suitable notion of equivalence.

The axioms defining the semantics of a language provide the possibility to interpret the axioms as a set of rewrite rules. This allows for rapid prototyping of the language. Moreover, based on the axiomatic system, there is an option for theorem proving. Examples of an axiomatic semantics are the pre-and post conditions used for programming languages [10] or the use of axioms in the context of concurrency [1].

The semantics of the language are mostly defined on the abstract syntax representation. In case the language has a graphical syntax, its semantics can be defined directly on the graphical syntax, but it is often more convenient to define a mapping from the graphical syntax onto the abstract syntax and formalise the semantics of the latter.

As may be expected, the definition of a (formal) semantics is crucial to unambiguously understand the programs and to define analysis techniques, together with proper support tools. More than one semantics may be defined, as long as these are consistent.

Analysis techniques are used for the semantical analysis of expressions in a language. We consider these techniques as part of the semantical development of the language, as these techniques are largely dependent on the choices made in defining the semantics of the language. The analysis techniques provide an increased insight into the meaning of possible expressions. Moreover, correctness of the language is better understood by determining the properties of expressions in the language.

Often, the analysis techniques follow some standard mathematical approach. However, it is conceivable that new theory needs to be developed for performing the desired analysis.

2.4.3 Pragmatics

The pragmatics of a language deals with all aspects of the use of the language. Obviously, a language design is not finished without guidelines on how to properly use the language. A collection of examples may show the application of typical features, case studies will prove usefulness for real examples. Moreover, documentation, including tutorials and educational material, together with rules of thumb, etc. are needed to advocate the proper use of the language. These guidelines are called the *methodology* of the language.

Apart from the methodology of the language, tools need to be defined for interpreting or compiling the language, and to support the analysis of programs written in the language. Ideally, these tools should follow from the semantical definitions. For instance, an interpreter of a language needs to show exactly the behaviour described by the operational semantics. Several *meta-tools* support the generation of parsers and scanners based on the formally defined syntax. Dependent on the type of semantics, the generation of interpreters and other language processing tools is also viable (see e.g. [13, 25, 3, 7]).

3 Related Work

There are many publications describing the development of some specific DSL or which describe a set of (meta-) tools to support such development. There is, however, only little literature on methodological aspects of the design of domain-specific languages (see [26] for an overview of existing literature). We discuss some relevant work below.

Consel and Marlet [4] describe a methodology for developing DSLs. It relates two orthogonal perspectives (a programming language perspective and a software architecture perspective) and describes a staged development of DSLs. The methodology is based on the formal framework of denotational semantics, and uses techniques to obtain dedicated abstract machines from the denotational semantics of a language. See also Thibault's thesis [24], which describes a methodology, similar to the methodology of [4].

Weiss [27] has proposed the FAST process, which is a program family oriented software development process. This approach introduces a software engineering process called commonality analysis. This process yields information about the terminology that is used, commonalities between the members of a program family and variabilities of a program family. The FAST process consists of a set of procedures that are followed by domain engineers to produce a standard set of intermediate and final documents. This provides for a systematic way for defining a program family.

Montages and its graphical tool environment Gem-Mex

(see [15]) form a suite for describing several aspects of programming languages, such as syntax, static analysis and semantics, and dynamic semantics. Syntax is described by BNF rules, and Abstract State Machines (formerly known as evolving algebras) are used to define the semantics of a language. The system is able to generate a visual programming environment for the specified language. The Montages methodology has no support for domain analysis.

In [6], Gupta and Pontelli start reasoning from the observation that any software system can be understood in terms of how it interacts with the outside world. Thus, every system is in essence defined by its input language, which in turn can be considered a Domain-Specific Language. They use Horn logic to give a denotational definition of such DSL, which automatically yields a parser, an interpreter and tools to support verification. The focus of their research is on applying Horn logic for these purposes, without developing a more generally applicable methodology.

Pfahler and Kastens [17] discuss issues related to the maintenance of a DSL. Rather than updating a language by going through a new language development cycle again, they propose to develop DSLs in such a way that small maintenance can be performed easily. Thereto, they consider a language design based on a collection of components, which can be glued together in different ways, thus making for a more flexible language definition, or rather a language family. This DSL life-cycle is called the *Jacob* approach. Corresponding tool support makes it possible to automatically generate substantial parts of an implementation. The authors do not describe a methodology for designing a language family (i.e. an appropriate set of components). We expect that the methodology outlined here will also be applicable to language families.

4 The Case Study

We illustrate the language-driven approach by developing a domain-specific language for the regulation of traffic lights.

The problem deals with regulating traffic crossing traffic junctions. This is done by means of traffic lights and division of roads into lanes.

A standard approach to controlling these traffic lights is to fix an order in which these traffic lights allow traffic to cross the junction. This, however, leads to sub-optimal throughput, traffic congestion, etc. To overcome such problems, sensors are used that register the presence of traffic per lane. The sensors' information is the basis for the order in which these traffic lights allow traffic to cross the junction.

In order to cope with high-priority vehicles (e.g. police vehicles), special care must be taken to make sure these vehicles are allowed to cross the junction as soon as possible.

However, it is not allowed to have unsafe situations at the traffic junction at any moment in time. Hence, conflicting traffic streams are not allowed to cross the traffic junction at the same time. Moreover, we cannot *a priori* assume that a traffic stream has cleared the junction immediately after a traffic light has changed to red. Therefore, to each traffic light a *clearance duration* is associated. The clearance duration of a traffic light is the time that is needed to clear the junction from traffic. In order to prevent a traffic light from switching colours too fast, we associate a minimal duration to each colour of the traffic light.

The goal is to obtain autonomous traffic junction regulators that are more efficient than the controllers defined by the standard approach and still guarantee safety. To achieve this goal, we develop a DSL that is tailored to the control of traffic lights as we envision it.

In the subsequent sections, the main focus is on the first stages of the language-driven approach. For a full coverage of the case study, please refer to [16]. In Section 4.1 we focus on the problem domain. Its concretisation into the problem space is discussed in Section 4.2. Finally, in Section 4.3, the syntax of the language is sketched. The semantics of the language, the analysis and its pragmatics can all be found in [16].

4.1 The Problem Domain

We assume that the introductory text in the previous section is sufficient for a basic understanding of the problem domain.

The concepts that are a natural consequence of the characterisation of a traffic junction, made in the previous section, are discussed in the subsequent paragraphs. Notice that we have already marked the concepts (using ^(f) for *fixed* and ^(v) for *variable*) that are part of the problem space, as to avoid duplication of information. In Section 4.2 we provide a motivation for our choice for these concepts.

Miscellaneous. For a more detailed exposition of the concepts introduced in this paragraph, please, refer to [16]. Here, we mention the existence of two essentially different concepts of time, viz. relative time (*Duration*^(f)) and absolute time (*Time*^(v)). Furthermore, we recognise the importance of the concept of a traffic participant (*Roaduser*), the type of the participants (*T_Roaduser*), and their relation (*UserType* : *Roaduser* → *T_Roaduser*).

Junctions. One of the most natural concepts of our problem domain is the concept of a junction. For junctions, we can recognise three levels of concreteness: the physical topology of the intersection, the traffic rules that apply to the intersection and the logical characteristics of the intersection. These three levels are explained in greater

detail below.

The physical topology of the intersection consists of several crossing roads. Roads can be divided into a number of lanes. We define lanes as stretches of road that have identical behaviour (lanes can also be sidewalks or rail tracks), i.e. a lane consists of a number of (parallel) strips. The users of a lane are supposed to follow the same route (or set of routes) on an intersection. For traffic junctions, we consider two types of lanes, viz. lanes entering and lanes leaving an intersection. Since we are interested in traffic crossing an intersection, we must consider the possibilities for doing so. From the perspective of the physical topology, we arrive at the notion of possible continuations for every lane entering an intersection.

<i>InLanes</i> ^(v)	Set of all traffic lanes entering a junction
<i>OutLanes</i> ^(v)	Set of all traffic lanes leaving a junction
<i>Lanes</i>	= <i>InLanes</i> ∪ <i>OutLanes</i>
	requirement: <i>InLanes</i> ∩ <i>OutLanes</i> = ∅
<i>PossibleContinuations</i>	: <i>InLanes</i> → (P(<i>OutLanes</i>) - {∅})
<i>PossibleLaneUsers</i>	: <i>Lanes</i> → P(<i>T_Roadusers</i>)

Observing the traffic laws that hold for an intersection, we see that these laws restrict traffic in an essential way. Rather than considering all possible continuations of a lane entering an intersection, we should in fact only consider the subset of continuations that is allowed by law. We thus arrive at the notion of continuations. Note that a lane entering an intersection must always have at least one continuation.

<i>LaneUsers</i>	⊆ <i>PossibleLaneUsers</i>
<i>Continuations</i> ^(v)	⊆ <i>PossibleContinuations</i>
	requirement: $\forall a \in \text{InLanes}, b \in \text{Continuations}(a)$
	<i>LaneUsers</i> (<i>a</i>) ⊆ <i>LaneUsers</i> (<i>b</i>)

From a logical point of view, the intersection can still exhibit unsafe behaviour. This unsafe behaviour has two causes. On the one hand, traffic entering the intersection via one lane can be in conflict with traffic entering the intersection via another lane. This conflict is dependent on the physical location of the lanes and the continuations of lanes entering the intersection. In order to reason about such lanes, we describe which lanes are conflicting, i.e. which lanes cannot simultaneously have a green light. Such a conflict relation is often called a *conflict matrix*.

On the other hand, we can observe that it takes some time for a traffic stream to clear the intersection after it has received a red light. This period needs to be taken into account in order to guarantee safety. We refer to this period as the clearance duration. Clearance duration is a binary function on the lanes entering and the lanes leaving an intersection. We can consider a more abstract notion of clearance duration, i.e. one that determines for an incoming lane the maximum clearance duration over all its continuations.

$Conflict$	$\subseteq (InLanes \times OutLanes)^2$ requirement: $Conflict$ is symmetric and irreflexive
$Conflict^{(v)}$	$\subseteq InLanes^2$ where $Conflict$ is derived as: $\{(i_1, i_2) \mid \exists o_1 \in Continuations(i_1) \exists o_2 \in Continuations(i_2) Conflict((i_1, o_1), (i_2, o_2))\}$
$ClearanceDuration$: $InLanes \times OutLanes \rightarrow Duration$ requirement: $ClearanceDuration$ is a partial function defined on all (i, o) , such that $i \in InLanes, o \in Continuations(i)$
$ClearanceDuration^{(v)}$: $InLanes \rightarrow Duration$ where $ClearanceDuration(l)$ is derived as: $\max\{ClearanceDuration(l, o) \mid o \in Continuations(l)\}$

Traffic lights. Various important characteristics of traffic lights can be identified. A main characteristic is the set of colours the traffic light has. A traffic light usually changes colour in a fixed order, i.e. a notion of state cycle can be identified. The state of a traffic light is tightly coupled to the traffic light itself, i.e. a *current* state can be identified. We are also interested in how long the light is already in this state. Traffic lights are often required to be in a state for a minimum time (e.g. a traffic light is required to show a green light for at least three seconds).

$TL_State^{(f)}$	Non-empty set of all possible traffic light states
$StateCycle^{(f)}$	$\in TL_State^+$
$TLights$	Set of all traffic lights
$CurrentTLightState^{(v)}$: $TLights \rightarrow TL_State$
TL_Loc	: $TLights \rightarrow InLanes$
$MinStateTime^{(v)}$: $TL_State \rightarrow Duration$
$CurrentDuration^{(v)}$: $TL_State \rightarrow Duration$

Sensors. To obtain information about their environment, traffic lanes must be equipped with sensors. When a sensor is triggered, it produces an input event that changes the state of that sensor. Thus, sensors have a notion of state. Moreover, at each moment in time, we can inspect the state of a sensor, hence, we can identify the *current* state for sensors.

Sensors can be placed at lanes for detecting specified types of road users. This is convenient for detecting speeding ambulances or police vehicles.

$Sensors^{(v)}$	Set of all sensors
$SensorState^{(f)}$	Set of all possible sensor states
$CurrentSensorState^{(v)}$: $Sensors \rightarrow SensorState$
$SensorLoc^{(v)}$: $Sensors \rightarrow InLanes$
$SensorRecog$: $Sensors \rightarrow \mathcal{P}(T_Roadusers)$

4.2 The Problem Space

An important step in the language-driven approach is the identification of the problem space, narrowing the problem domain and extending it with design information. The restriction of the problem domain is sketched in Section 4.2.2.

The design decisions (i.e. the extensions of the problem domain) are discussed in Section 4.2.1.

4.2.1 Design Decisions

We model the traffic lights as a *competitive system*, i.e. every traffic light competes with other lights for the right to change colour. To this end, information local to a traffic stream must be kept to reach a global decision on which lights can change colour. Thereto, we introduce the concept of assigning *priorities* to traffic streams. These priorities can dynamically change, based on the progress of time or the detection of traffic. We assume a totally ordered set $Prio$ of priority values. Then we have for every sensor the priority value to which the corresponding lane will be initialised if traffic is detected by that sensor (denoted by $InitPrio(s)$ for sensor s). Finally, we have a priority update function (denoted by $UpdatePrio$), which determines the new priority value of a lane after the elapse of one time unit.

$Prio^{(f)}$	Totally ordered set of priority values
$InitPrio^{(v)}$: $Sensors \rightarrow Prio$
$UpdatePrio^{(v)}$: $InLanes \times Prio \rightarrow Prio$
$CurrentLanePrio^{(v)}$: $InLane \rightarrow Prio$

4.2.2 Reduction of the Problem Domain

In Section 4.1, we have marked various concepts as fixed and variable, using (f) and (v) . The unmarked concepts turn out to be irrelevant with respect to traffic light control. In the subsequent paragraphs we restrict our discussion to only a single example of an irrelevant, a fixed and a variable concept.

Irrelevant concepts. An example of an irrelevant notion is $TLights$. Although this notion is at the right level of abstraction, we observe that it would not be a severe restriction if there is exactly one element of $TLights$ for every element of $InLanes$. Therefore, we can simply identify these two notions and discard $TLights$.

Variable concepts. The variable concepts that depend on the actual problem instance can be defined using the syntax. An example of such a variable concept is the conflict matrix (i.e. the concept $Conflict$). In our goal to describe traffic light control for more than a single fixed traffic junction, we need to take the conflict matrix into account. This is due to the fact that, dependent on the junction, the conflict matrix can differ. Hence, fixing the conflict matrix would be unwise, as it would restrict our language to describing only junctions with identical conflicting traffic streams.

Fixed concepts. Several notions can be fixed to a concrete value in order to make the problem less abstract. For

instance, the colours that a traffic light can have can be fixed by defining $TL_State = \{green, yellow, red\}$, which also determines a standard order $StateCycle = green \circ yellow \circ red$.

4.3 Syntax

This section will describe the syntax of the traffic regulation language. We provide a definition of the abstract syntax and we give examples of expressions in the concrete and graphical syntax. Due to space limitations we will not give the syntax definitions in full detail.

The abstract syntax serves to express in a minimal format the semantically relevant information which a designer of an intersection should provide in order to obtain an operational system. The abstract syntax has a clear correspondence with the variable concepts identified in the problem space.

Words between angular brackets, $\langle \rangle$, are the non-terminals of the language. We assume that the non-terminals $\langle inlaneid \rangle$, $\langle outlaneid \rangle$, and $\langle sensorid \rangle$ produce disjoint sets of identifier symbols. Furthermore, $\langle updateprio \rangle$ produces a natural expression (possibly containing occurrences of a variable, say x) which represents the priority update function. The initial priority of a sensor is captured by $\langle initprio \rangle$. Non-terminals $\langle initprio \rangle$, $\langle clearance \rangle$, $\langle greentime \rangle$, $\langle yellowtime \rangle$, and $\langle redtime \rangle$ produce a natural numeric constant.

```

(junction) ::= (lane)* (conflict)* (mintime)
(lane) ::= (inlaneid) (continuation)* (sensor)* (updateprio)
(continuation) ::= (outlaneid) (clearance)
(sensor) ::= (sensorid) (initprio)
(conflict) ::= (inlaneid) (inlaneid)
(mintime) ::= (greentime) (yellowtime) (redtime)

```

Of course, static requirements also need to be defined. For a more detailed overview of the syntax, see [16].

There are many ways in which the abstract syntax can be represented in a more readable format. The textual representation of the example in Figure 2 is a bit more verbose. This example describes a junction with two incoming lanes (a and b) and two outgoing lanes (c and d). Lane a continues at lanes c and d . The clearance duration of the path from lane a to lane c is 3. Lane a has two sensors, called *normal* and *bus*. The initial priority of the normal sensor is 10, while detection of a bus sets the priority to 100. The sensor at lane b cannot make a distinction between the type of traffic detected. The priorities of the lanes a and b are updated every time unit with the update functions¹ $\lambda x.x + 1$ and $\lambda x.x + 2$, respectively. The two lanes a and b have a conflict. Finally, the minimal state time of the traffic light colours is set to 1, 1, 3 (for red, yellow, and green).

We leave it to the reader to interpret the graphical symbols in Figure 2.

¹We denote a function f with parameter x by $\lambda x.f(x)$.

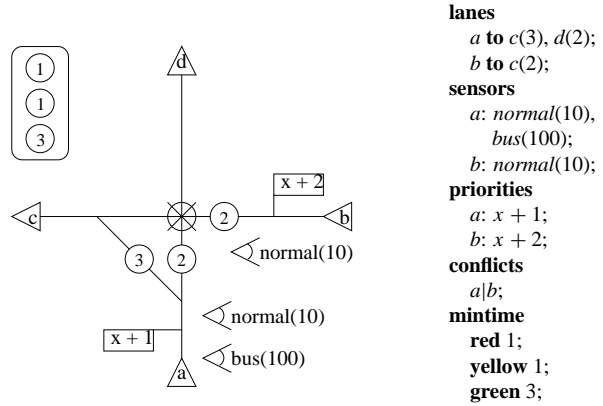


Figure 2. Example junction in textual and graphical syntax.

5 Closing Remarks

The purpose of this paper was to promote the use of Domain-Specific Languages as a regular part of the software engineering process. Therefore, based on well known material and published case studies, we described a language-driven approach for software development.

We identified three phases in this approach: formulation of the problem domain, the identification of the problem space and the development of the language. The problem domain follows from a domain analysis. The problem domain is necessarily general and abstract and therefore does not focus on the actual problem exclusively. The problem space adds concepts (concepts due to design decisions) to the concepts of the problem domain. Moreover, the problem space separates the relevant concepts from the irrelevant concepts and considers instantiations of the relevant concepts, as to better accommodate for the problem or class of problems that must be solved. The Domain-Specific Language being developed must exactly span the problem space. This language is developed in three sub-phases: syntax, semantics, and pragmatics.

This approach is illustrated by means of a conceptually simple case study. Although the case study is presented in a linear way, the process of developing the case study was iterative. It was our experience that one of the main factors with respect to the quality of the language design was the consistency of the deliverables involved. For instance, in our case study, the priority function as a concept was introduced only *after* developing the semantics, i.e. it was not obtained as a concept in the initial domain analysis. An integrated set of support tools covering all phases of the approach should take care of this consistency checking.

Acknowledgements. The authors would like to thank Paul Derksen, Ronald Middelkoop, Felix Ogg, and Robert Spee for their discussions and help on the case study. Marc Voorhoeve is acknowledged for his help in clarifying the ideas that led to this paper. Thanks are due to Michael van Hartskamp for proof reading.

References

- [1] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.
- [2] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.
- [3] M. Clavel, F. Duràn, S. Eker, and J. Meseguer. Maude as a formal meta-tool. In J. Wing and J. Woodcock, editors, *The World Congress On Formal Methods*, volume 1709 of *LNCS*, pages 1684–1703. Springer-Verlag, 1999.
- [4] C. Consel and R. Marlet. Architecturing software using a methodology for language development. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of the 10th International Symposium on Programming Languages, Implementations, Logics and Programs (PLIP/ALP '98)*, volume 1490, pages 170–194. Springer-Verlag, 1998.
- [5] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.
- [6] G. Gupta and E. Pontelli. A Horn logic denotational framework for specification, implementation and verification of domain specific languages. Technical report, New Mexico State University, 1999.
- [7] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM Sigplan Notices*, 35(3):39–48, 2000.
- [8] M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction using Structured Operational Semantics*. Wiley, New York, 1991.
- [9] R. M. Herndon and V. A. Berzins. The realizable benefits of a language prototyping language. *IEEE Transactions on Software Engineering*, 14:803–809, 1988.
- [10] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.
- [11] ITU-T. *ITU-T Recommendation Z.106: Common Interchange Format for SDL*. ITU-T, Geneva, 1996.
- [12] R. Kieburtz, L. McKinney, J. Bell, J. Hook, A. Kotov, J. Lewis, D. Oliva, T. Sheard, I. Smith, and L. Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th IEEE International Conference on Software Engineering ICSE-18*, pages 542–553. IEEE Computer Society Press, 1996.
- [13] P. Klint. A meta-environment for generating programming environments. *ACM Transactions of Software Engineering and Methodology*, 2(2):176–201, 1993.
- [14] D. E. Knuth. *Semantics of Context-Free Languages*, volume 2, pages 127–145. Springer-Verlag, New York, 1968.
- [15] P. W. Kutter and A. Pierantonio. Montages specifications of realistic programming languages. *Journal of Universal Computer Science*, 3(5):416–442, 1997.
- [16] S. Mauw, W. T. Wiersma, and T. A. C. Willemse. Language-driven system design. Technical Report 01-07, Department of Mathematics and Computer Science, Eindhoven University of Technology, 2001.
- [17] P. Pfahler and U. Kastens. Configuring component-based specifications for domain-specific languages. In *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*. IEEE Computer Society Press, 2000.
- [18] G. D. Plotkin. A structural approach to operational semantics. Technical Report DIAMI FN-19, Computer Science Department, Aarhus University, 1981.
- [19] R. Prieto-Díaz. Domain analysis: An introduction. *Software Engineering Notes*, 15(2):47–54, 1990.
- [20] J. Rekers and A. Schürr. A graph grammar approach to graphical parsing. In *Proceedings of the 1995 IEEE Symposium on Visual Languages*, 1995.
- [21] W. W. Royce. Managing the development of large software systems. In *Proceedings of the IEEE WESCON*, 1970.
- [22] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., Newton, MA, 1986.
- [23] K. Slonneger and B. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley, 1995.
- [24] S. Thibault. *Domain-Specific Languages: Conception, Implementation and Application*. PhD thesis, IRISA/University of Rennes 1, 1998.
- [25] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: A component-based language development environment. In R. Wilhelm, editor, *Compiler Construction*, volume 2027 of *LNCS*, pages 365–370. Springer-Verlag, 2001.
- [26] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.
- [27] D. Weiss. Creating domain-specific languages: The fast process. In S. Kamin, editor, *First ACM-SIGPLAN Workshop on Domain-Specific Languages; DSL'97*. Technical report, University of Illinois, Department of Computer Science., 1997. See URL at <http://www-sal.cs.uiuc.edu/kamin/dsl>.
- [28] R. T. Yeh. An alternate paradigm for software evolution. In P. A. Ng and R. T. Yeh, editors, *Modern Software Engineering: Foundations and Current Perspectives*, New York, NY, 1990. Van Nostrand Reinhold.