Language-Driven System Design

S. Mauw, W.T. Wiersma, T.A.C. Willemse

Eindhoven University of Technology, Department of Mathematics and Computer Science, P.O. Box 513, NL–5600 MB Eindhoven, The Netherlands

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.

Key words: Software Engineering, Domain-Specific Languages, Language-Driven Approach, Operational Semantics, Traffic Light Control.

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 [3], and the incremental model [30].

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.

Email addresses: s.mauw@tue.nl (S. Mauw), w.t.wiersma@stud.tue.nl (W.T. Wiersma), t.a.c.willemse@tue.nl (T.A.C. Willemse).

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. [12]) 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. [12,15]).

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 techniques, there is ample literature available (e.g. [6,20]).

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 [16,26], 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 the case study. In Section 2, we sometimes refer to the case study to exemplify some of the more abstract notions we encounter.

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.

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 [22] or the spiral model [3]. 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.

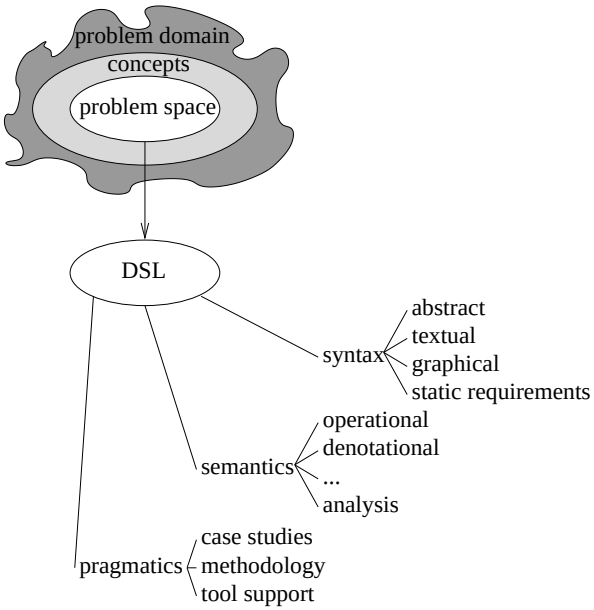


Fig. 1. The language-driven approach.

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 dependency between the stages. Also, these stages should not be considered *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. We occasionally refer to the traffic light case study described in Section 4 to explain some of the more abstract notions discussed in these sections.

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 [20,6] 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 alread been made. An example of the concepts that are revealed in the domain analysis of the traffic light case study is given below.

Example 2.1. In the traffic light case study, we describe the domain model by means of basic mathematical constructs, such as sets and relations. Typical concepts in our case study are plain entities like *road user* and *lane* and relations like *conflict*, which describes which lanes may have potentially conflicting traffic. **End example.**

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.

Variable concepts come in two flavours. A concept is called *variable* whenever it varies depending on the actual problem instance, e.g. when its actual values are fixed at run-time. Whenever a concept varies *within* the actual problem instance, we also refer to this concept as *variable*. 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 ac-

tual 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. Moreover, by classifying concepts as fixed important steps towards a design are taken.

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. Since the notions discussed in this section are rather abstract, the four types of concepts are exemplified below.

Example 2.2. In our case study, we have introduced the concept of *priorities*. This concept is derived from our desire to model the traffic lights as a competitive system. However, the notion of priorities is not a notion that could have been derived during the identification of the problem domain. Note that different design decisions might have led to the introduction of other concepts.

The concept of a *road user* turns out to be irrelevant in our case study. Although a road user plays a vital role in the domain analysis, it does not re-occur in any of the subsequent phases. This is because road users are not important to the goals set for solving our actual problem; their presence can only be detected indirectly by sensors.

An example of a variable concept, is the notion of a *conflict matrix*. The conflict matrix describes the actual situation at a particular traffic intersection. In order to deal with more than one fixed intersection, we require this concept to be variable. This means that this concept also reappears in the syntax of the language, as it is needed there to describe the intersection in terms of its conflicts. An example of a concept which is variable within a problem instance is the *current colour* of a traffic light. This will change during operation and is, therefore, included in the state of the system.

When we fix the order in which a traffic light displays its colours, e.g. from green to yellow to red, this concept can be considered as an example of a fixed concept. Not determining the order of the colours in advance would support any ordered list of colours and therefore would be much more general. However,

this would also increase the complexity of the syntax, because it would need constructs for specifying the order of the colours. Since the order for traffic light colours is more or less standardised, fixing the order is not a severe restriction. **End example.**

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 writeability 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 expressions in the abstract syntax are generally not meant for human processing or usage, but they are useful during design of the language.

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). For the graphical syntax, no generally accepted format for defining the language exists. The most popular way of defining the graphical language is by means of *graph grammars* [21].

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*. This title is somewhat misleading as it deals with syntactical properties of the language. The notion of static semantics also has a different interpretation, namely the semantics of the static (i.e. non-behavioural) part of a language. Therefore, we prefer to use the term *static requirements* whenever we refer to these additional syntax requirements. Most often, attribute grammars [17] 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 [14]).

Not all three formats may be necessary: one might skip e.g. the textual syntax. Two issues must be kept in mind, namely, that there has to be a syntactical representation which covers all semantically relevant issues and that a formal definition of the syntactical ingredients should be given.

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 components 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, see e.g. [10,24].

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. [11]). 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 [1].

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 [16,26] or MAUDE [4] 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 translation to a well-understood mathematical model, as described in the beginning of this section (see e.g. [23]). 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 [13] or the use of axioms in the context of concurrency [2].

The semantics of the language are mostly defined on the abstract syntax rep-

resentation. 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. [16,4,8]).

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 [27] for an overview of existing literature). We discuss some relevant work below.

Consel and Marlet [5] 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 [25], which describes a methodology, similar to the methodology of [5].

Weiss [29] 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 [18]) 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 [9], 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 [19] 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 (see e.g. [28]). The case study is discussed in great detail in the subsequent sections.

The problem deals with traffic junctions and the traffic passing the junction. We can distinguish between traffic junctions that do not need any control and traffic junctions that do need control. The former are often traffic junctions that have only little traffic passing it, whereas the latter are often junctions that have many conflicting traffic streams. Regulation of traffic streams 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. Notice that the addition of sensors renders systems that can respond to events from their environments, i.e. the traffic light controllers we consider are dynamic, reactive systems.

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.

Our presentation of the case study will be in a linear way. However, it must be noticed that the results described in this section are the product of several iterations. In our presentation of the deliverables for our case study, we closely follow the order prescribed by the language-driven approach. In Section 4.1 we focus on the problem domain. The concretisation of the problem domain into the problem space is discussed in Section 4.2. Finally, in Sections 4.3 to 4.6, the language definition is presented.

4.1 The Problem Domain

The first step of the language-driven approach to software engineering is a proper identification of all concepts that are essential in the problem domain. As most people are familiar with traffic junctions, obtaining an initial set of concepts is rather straightforward (e.g. by means of a brainstorm session and interviews). 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.

Time. A traffic light controller is a dynamic, time-dependent system. The concept of time is therefore indispensable. There are two different types of time, i.e. relative time and absolute time. Relative time is usually employed if one wants to refer to periods of time (e.g. a traffic light must be green for at least ten seconds). Absolute time is used to model that events must occur at specific moments (e.g. a traffic light is out of order until May 1, 2001). For traffic light controllers, both types of time are possible. Hence, we introduce the following concepts:

$Duration^{(f)}$	Lapse of time (relative)
$Time^{(v)}$	'Calendar' time (absolute)

Participants. As mentioned before, the traffic light controller is a reactive system. It responds to the events it receives from its environment. These environmental events are triggered by traffic participants, i.e. road users. These road users can be of a specific type, e.g. car, pedestrian, train, etc.

$Roadusers$	Set of all possible traffic participants
$T\_Roadusers$	Set of all possible types of traffic participants
$UserType$	$: Roadusers \rightarrow T\_Roadusers$

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.

$InLanes^{(v)}$	Set of all traffic lanes entering a junction
$OutLanes^{(v)}$	Set of all traffic lanes leaving a junction
$Lanes$	$= InLanes \cup OutLanes$
	requirement: $InLanes \cap OutLanes = \emptyset$
$PossibleLaneUsers$	$: Lanes \rightarrow \mathcal{P}(T\_Roadusers)$
$PossibleContinuations$	$: InLanes \rightarrow (\mathcal{P}(OutLanes) - \{\emptyset\})$

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 consider a subset thereof. This is motivated by the fact that, although the physical possibilities are there, the law forbids these continuations. We thus arrive at the notion of continuations.

$LaneUsers$	: $Lanes \rightarrow \mathcal{P}(T\_Roadusers)$
	requirement: $\forall_{l \in Lanes}\ LaneUsers(l) \subseteq PossibleLaneUsers(l)$
$Continuations$	: $InLanes \rightarrow (\mathcal{P}(OutLanes) - \{\emptyset\})$
	requirements:
	$\forall_{l \in InLanes}\ Continuations(l) \subseteq PossibleContinuation(l)$, and
	$\forall_{a \in InLanes, b \in Continuations(a)}\ LaneUsers(a) \subseteq LaneUsers(b)$

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), for which $i \in InLanes$, and
	$o \in Continuations(i)$
$ClearanceDuration^{(v)}$	: $InLanes \rightarrow Duration$
	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	$: TLights \rightarrow InLanes$
$MinStateTime^{(v)}$	$: TL\_State \rightarrow Duration$
$CurrentDuration^{(v)}$	$: TLights \rightarrow Duration$

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. As mentioned before, the problem space is both a restriction

of concepts of the problem domain and an extension of the problem domain with concepts due to design decisions. The restriction of the problem domain is discussed in Section 4.2.2. First, the design decisions (i.e. the extensions of the problem domain) are discussed in Section 4.2.1.

4.2.1 Design Decisions

We will model the traffic lights as a *competitive system*. This means that every traffic light competes with other lights for the right to change colour. So we have to keep information local to the traffic stream to reach a global decision which lights can change colour. For this reason 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. For sensor s, this will be denoted by *InitPrio(s)*. Finally, we have a priority update function, which determines the new priority value of a lane after the elapse of one time unit. This function will be denoted by *UpdatePrio*.

$Prio^{(f)}$	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

The concepts that are irrelevant to traffic light control are the concepts of Section 4.1 that are not marked with $^{(f)}$ or $^{(v)}$. The concepts that turn out to be relevant, but can be fixed have been marked with $^{(f)}$, whereas the concepts that need to be variable are marked with $^{(v)}$. In this section, we restrict our discussion to only a few examples of irrelevant, fixed and variable concepts.

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*.

A second example of an irrelevant concept is the concept of *Roaduser*. This concept is irrelevant to our actual problem, since we have decided to build a system in which individual road users do not play a role. Traffic participants

can only be detected indirectly by a sensor. They might play a role, however, in case one of the goals was to build a simulator showing behaviour of individual road users.

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.

Another example of a variable concept is the concept of clearance duration. As we have seen, the notion of clearance duration is important to guarantee safety of the traffic junction. Hence, the concept cannot be considered irrelevant. If we consider this clearance duration as a fixed concept, then we restrict our language to describing intersections that have a single (fixed) clearance duration for all traffic streams. This, of course, is too restrictive. Hence, the concept of clearance duration must be defined as a variable concept and can thus be defined using the syntax of the language.

For the other variable concepts (*InLanes*, *Continuations*, *Outlanes*, *MinStateTime*, *Sensors*, *SensorLoc*, *InitPrio*, and *UpdatePrio*), a similar reasoning holds.

An example of a variable concept that is variable within the problem instance is the concept *CurrentSensorState*. This concept defines the relation between the concepts of *Sensors* and the concepts of *State*. This relation, however, is not static, as the state of a sensor can change over time (e.g. by means of traffic passing the sensor). In fact, this relation clearly illustrates the dynamic nature of traffic light control.

Similarly, the concept of *CurrentTLightState* is a variable concept.

Fixed concepts. We have fixed several notions to a concrete value in order to make the problem less abstract. First of all, we will restrict the colours that a traffic light can have by defining $TL\_State = \{green, yellow, red\}$, which also determines the standard order $StateCycle = green \circ yellow \circ red$. Such a decision may come from the fact that the system is only to be applied in countries where this is the standard order of operation. It might be considered a severe restriction that this also implies that special operation of traffic lights (e.g. a flashing yellow light) is not supported.

For ease of reasoning, we will take $Prio = \mathbb{N}$. We will assume a discrete time domain, and set $Time = Duration = \mathbb{N}$

4.3 Syntax

This section describes 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. We refrain from providing the definitions of the concrete and graphical syntax, as these do not add to the understanding of the language.

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.

$\langle junction \rangle ::= \langle lane \rangle^* \ \langle conflict \rangle^* \ \langle mintime \rangle$
$\langle lane \rangle ::= \langle inlaneid \rangle \ \langle continuation \rangle^* \ \langle sensor \rangle^* \ \langle updateprio \rangle$
$\langle continuation \rangle ::= \langle outlaneid \rangle \ \langle clearance \rangle$
$\langle sensor \rangle ::= \langle sensorid \rangle \ \langle initprio \rangle$
$\langle conflict \rangle ::= \langle inlaneid \rangle \ \langle inlaneid \rangle$
$\langle mintime \rangle ::= \langle greentime \rangle \ \langle yellowtime \rangle \ \langle redtime \rangle$

As an example of a static requirement defined on this abstract syntax, we specify the predicate *irreflexive-conflict*. This static requirement follows from the requirement on the conflict matrix as specified in Section 4.1, and in fact expresses the irreflexivity of the conflict matrix.

$irreflexive\text{-}conflict(lane\text{-}list\ conflict\text{-}list\ mintime) = irreflexive\text{-}conflict(conflict\text{-}list)$
$irreflexive\text{-}conflict(\varepsilon) = true$
$irreflexive\text{-}conflict(inlaneid1\ inlaneid2\ conflict\text{-}list) =$
$\quad (inlaneid1 \neq inlaneid2) \wedge irreflexive\text{-}conflict(conflict\text{-}list)$

We have overloaded the predicate name such that it accepts expressions of type $\langle junction \rangle$ and $\langle conflict \rangle^*$. With ε we denote the empty list. Typing of the other variables follows from their naming scheme.

In the same way we can define auxiliary functions to extract information from the abstract syntax, such as the function *sensors* which determines for each lane l the set of available sensors.

$$
\begin{aligned}
&\mathit{sensors}(l, \mathit{lane\text{-}list\ conflict\text{-}list\ mintime}) = \mathit{sensors}(l, \mathit{lane\text{-}list})\\
&\mathit{sensors}(l, \mathit{inlaneid\ continuation\text{-}list\ sensor\text{-}list\ updateprio\ lane\text{-}list}) =\\
&\qquad \begin{cases} \mathit{sensors}(\mathit{sensor\text{-}list}) & \text{if } l = \mathit{inlaneid}\\ \mathit{sensors}(l, \mathit{lane\text{-}list}) & \text{if } l \neq \mathit{inlaneid} \end{cases}\\
&\mathit{sensors}(\varepsilon) = \emptyset\\
&\mathit{sensors}(\mathit{sensorid\ initprio\ sensor\text{-}list}\ ) = \{\mathit{sensorid}\} \cup \mathit{sensors}(\mathit{sensor\text{-}list})
\end{aligned}
$$

This function will be used in the definition of the semantics.

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 slightly 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 functions[1] $\lambda x.x + 1$ 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.

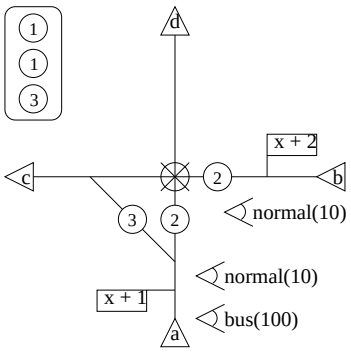
4.4 Semantics

In this section we provide an operational semantics for the traffic light control system. We define a *state space* (see Section 4.4.1) and *transition functions* (see Section 4.4.2).

4.4.1 State

In order to give an operational semantics based on state transitions, we will first define the *state* of the system. The state is based on the concepts which are variable within a problem instance, i.e. the priorities of the lanes, the states of the traffic lights, the time that the traffic lights have their current

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



lanes
  a **to** $c(3)$, $d(2)$;
  b **to** $c(2)$;
sensors
  a: $normal(10)$,
    $bus(100)$;
  b: $normal(10)$;
priorities
  a: $x + 1$;
  b: $x + 2$;
conflicts
  $a|b$;
mintime
  red 1;
  yellow 1;
  green 3;

Fig. 2. Example junction in textual and graphical syntax.

colour, the states of the sensors, and the absolute time. We define state domain $\Sigma = \Pi \times \Gamma \times \Delta \times \Xi \times Time$, where

1. Π $= InLanes \rightarrow Prio$
2. Γ $= InLanes \rightarrow TL\_State$
3. Δ $= InLanes \rightarrow Duration$
4. Ξ $= Sensors \rightarrow SensorState$
5. $Time$ is the absolute time domain

Then the state of the system is represented by a five-tuple $\sigma = (\pi, \gamma, \delta, \xi, \tau) \in \Sigma$. Henceforth, we will use the notations σ and $(\pi, \gamma, \delta, \xi, \tau)$ interchangebly. Without making it explicit, e.g. the function γ_2 will denote the second component of state σ_2. The initial state of the system is $\sigma_0 = (\pi_0, \gamma_0, \delta_0, \xi_0, \tau_0)$, where

$\pi_0 = \lambda l.0$

$\gamma_0 = \lambda l.red$

$\delta_0 = \lambda l.0$

$\xi_0 = \lambda s.false$

$\tau_0 = 0$

4.4.2 Transition rules

There are three ways in which the state of the system can be changed. First, there can be an input event (i.e. a sensor changes its state). Secondly, an output event can be generated (i.e. a traffic light changes colour). And thirdly, time can progress. These three events give rise to three types of transitions.

To simplify matters, we assume a slotted operation. By this, we mean that during each time interval first all sensor inputs are collected (if any), then all traffic light outputs are generated (if any), and finally, time progresses to the next time slot. Henceforth, we will consider only transition graphs which satisfy this restriction on the order of transitions.

A transition generated by an input event is denoted by $\xrightarrow{i}$, where $i \subseteq \mathit{Sensors}$ denotes the set of sensors which have detected traffic during the time slot. A transition based on an output event is denoted by $\xrightarrow{\langle tr,ty,tg\rangle}$, where $tr, ty, tg \subseteq \mathit{InLanes}$ denote the sets of lanes whose traffic lights should advance to *red*, *yellow*, and *green*, respectively. Finally, $\xrightarrow{t}$ denotes progress of time with one time unit.

Now, we will discuss the three transition rules which define the transition system.

Check for input. We consider an input from a sensor as the indication that traffic has been detected during the just finished time slot. This does not mean that the traffic is (still) waiting. Sensor information is kept in the state variable ξ, which is set to *true* for a given sensor every time that the sensor yields an input. Since the system cannot control its inputs, every possible collection of sensor inputs should be accepted. This is modelled by having a transition for every subset of $\mathit{Sensors}$.

For every $i \subseteq \mathit{Sensors}$, we have the following transition:

$$(\pi, \gamma, \delta, \xi, \tau) \xrightarrow{i} (\pi', \gamma, \delta, \xi', \tau),$$

where

$$\pi' = \lambda l. \begin{cases} \max(\pi(l), \max\{\mathit{InitPrio}(s) \mid & \\ \qquad s \in \mathit{sensors}(l) \wedge (\xi(s) \vee s \in i)\}) & \text{if } \gamma(l) = \mathit{red} \\ 0 & \text{otherwise} \end{cases}$$

$$\xi' = \lambda s.(\xi(s) \vee s \in i)$$

The priority of lane l is set to the maximum value of the initial priorities

from all triggered sensors that belong to the lane. However, if the priority is already larger than the initial priority, the old value remains. The sensor status is changed as explained above. All other components in the state remain unchanged.

Generate output. The transition rule in which the output to the traffic lights is generated has the following appearance.

$$(\pi, \gamma, \delta, \xi, \tau) \xrightarrow{\langle TR_\sigma, TY_\sigma, TG_\sigma \rangle} (\pi', \gamma', \delta', \xi', \tau),$$

The sets TR_σ, TY_σ, and TG_σ are defined below in such a way that the lane with highest priority can go first, while allowing non-conflicting traffic of lower priority to pass too.

In order to calculate these sets, we define $M_\sigma \subseteq \mathit{InLanes}$ as the set of all lanes that, based on their priorities, should receive green light. Possibly, not all lights from M_σ will be set to green in the current state, because conflicting streams may be crossing the junction. M_σ is a *conflict-free* subset of *InLanes* with *maximal priority*. A set has maximal priority if it contains at least a lane with highest priority, but it also contains lanes with possible next-highest priorities.

In order to choose a *unique* conflict-free subset M_σ, we impose an arbitrary total ordering $\preceq$ on the set of *InLanes*. This ordering is necessary to avoid complications with fairness that could otherwise arise in a situation where more than one set has maximal priority. The total order $\dot{\preceq}$ is then induced by $\preceq$, defining a lexicographical ordering on finite sets. This lexicographical ordering is obtained by associating to each set A a list, consisting of the elements of A in $\preceq$-decreasing order. Then, $A \dot{\preceq} B$ iff the list associated to A is lexicographically smaller than the list associated to B.

Based on the priority function π, we define an extended priority function, $\dot{\pi} : \mathcal{P}(\mathit{InLanes}) \to \mathcal{M}(\mathit{Prio})$, where $\mathcal{M}(\mathit{Prio})$ denotes the *multi-sets* of *Prio*, defined as $\dot{\pi}(A) = [\pi(a) | a \in A]$, for $A \subseteq \mathit{InLanes}$. Let $\leq$ be the total multi-set ordering for $\mathcal{M}(\mathit{Prio})$, (see e.g. [7]). We subsequently define the set NC of *non-conflicting* subsets of lanes, i.e.

$$NC = \{A \subseteq \mathit{InLanes} \mid \forall_{l,m \in A}\ \neg \mathit{Conflict}(l, m)\}$$

This yields a lattice $(NC, \lesssim)$, where we define the ordering $\lesssim$ for all $A, B \in NC$ as $A \lesssim B$ iff either $\dot{\pi}(A) < \dot{\pi}(B)$ or $\dot{\pi}(A) = \dot{\pi}(B) \wedge A \dot{\preceq} B$. In words, two elements A, B of NC are ordered $A \lesssim B$ in the following two cases. First, $A \lesssim B$ if the multi-set of priorities for B is greater than the multi-set of priorities for A (according to the multi-set ordering). Second, $A \lesssim B$ if the multi-sets of priorities for A and B are equivalent (which means both

A and B have (relative) maximal priority), but B is lexicographically larger than A. The lexicographical ordering essentially resolves the non-determinism in chosing the set with maximal priority. It follows from standard set-theory that the thus defined lattice $(NC, \lesssim)$ is also a complete lattice.

Since $(NC, \lesssim)$ is a complete lattice, we know the set NC has a unique maximum with respect to $\lesssim$. We therefore define M_σ as the maximum of the set NC. The reader is invited to check that, by construction, M_σ is a conflict-free subset of *InLanes* with maximal priority.

Example 4.1. Let us consider a junction with five *InLanes*, e.g. $\mathit{InLanes} = \{A, B, C, D, E\}$. We consider an arbitrary state σ of this junction. Let the conflict-matrix, and the priority function π of the state σ be given by the matrix as defined by Table 1.

Table 1
Configuration of the conflict-matrix.

	A	B	C	D	E	π
A		$\times$	$\times$	$\times$	$\times$	5
B	$\times$		$\times$			5
C	$\times$	$\times$				5
D	$\times$				$\times$	3
E	$\times$			$\times$		2

For the computation of M_σ, we must first determine the set of all non-conflicting subsets NC. Note that for every program written in our language, this set can be computed at the start of the system. Below, the set NC and the multi-sets of priorities, associated to each element in NC are listed.

$$\begin{array}{cccccccccccc} NC = \{ & \emptyset, & \{A\}, & \{B\}, & \{C\}, & \{D\}, & \{E\}, & \{B,D\}, & \{B,E\}, & \{C,D\}, & \{C,E\}\} \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ & [] & [5] & [5] & [5] & [3] & [2] & [5,3] & [5,2] & [5,3] & [5,2] \end{array}$$

If we apply the extended priority function $\dot{\pi}$ to the elements in NC and use the multi-set ordering $\leq$, we obtain the following ordering between these induced multi-sets:

$$[] \leq [2] \leq [3] \leq [5] \leq [5,2] \leq [5,3]$$

which is derived from the following chain:

$$\begin{aligned} &\dot{\pi}(\emptyset) \leq \dot{\pi}(\{E\}) \leq \dot{\pi}(\{D\}) \leq \dot{\pi}(\{A\}), \dot{\pi}(\{B\}), \dot{\pi}(\{C\}) \\ &\leq \dot{\pi}(\{E,B\}), \dot{\pi}(\{E,C\}) \leq \dot{\pi}(\{D,B\}), \dot{\pi}(\{D,C\}) \end{aligned}$$

Note that $\dot{\pi}(\{D,B\}) = \dot{\pi}(\{D,C\}) = [5,3]$, and therefore, the multi-set ordering is not sufficient to find *the* set with maximal priority. As our arbitrary ordering, we assume the well-known alphabetic ordering, i.e. $A \preceq B$. This induces the (complete) lattice $(NC, \dot{\preceq})$, which is spelled out below:

$$\emptyset \dot{\preceq} \{A\} \dot{\preceq} \{B\} \dot{\preceq} \{C\} \dot{\preceq} \{D\} \dot{\preceq} \{D,B\} \dot{\preceq} \{D,C\} \dot{\preceq} \{E\} \dot{\preceq} \{E,B\} \dot{\preceq} \{E,C\}$$

Now, the set with maximal priority is the set $\{D,C\}$, which is the lexicographically largest of the two sets having maximal priority (i.e. $\{D,B\}$ and $\{D,C\}$). **End example.**

The set TR_σ contains all yellow lights for which the minimal yellow time has elapsed. The set TY_σ contains all green lights for which the minimal green time have elapsed and which are in conflict with one of the *InLanes* in M_σ. The set TG_σ contains all red lights from M_σ for which the minimal red time have elapsed and which are not in conflict with any of the currently crossing traffic streams.

$$\begin{aligned}
TR_\sigma &= \{l \in \mathit{InLanes} \mid \gamma(l) = \mathit{yellow} \wedge \delta(l) \geq \mathit{MinStateTime}(\mathit{yellow})\} \\
TY_\sigma &= \{l \in \mathit{InLanes} \mid \gamma(l) = \mathit{green} \wedge \delta(l) \geq \mathit{MinStateTime}(\mathit{green}) \wedge \\
&\qquad \exists_{m \in M_\sigma} \mathit{Conflict}(l,m)\} \\
TG_\sigma &= \{l \in M_\sigma \mid \gamma(l) = \mathrm{red} \wedge \delta(l) \geq \mathit{MinStateTime}(\mathit{red}) \wedge \\
&\qquad \neg \exists_{m \in \mathit{InLanes}} (\mathit{Conflict}(l,m) \wedge \mathit{crossing}_\sigma(m))\}
\end{aligned}$$

We use the predicate $\mathit{crossing}_\sigma$, which states that there can be traffic on a certain lane due to either a green or yellow light, or a red light for which the clearance duration has not yet expired. Below, the predicate $\mathit{crossing}_\sigma$ is defined.

$$\mathit{crossing}_\sigma(m) = (\gamma(m) = \mathit{red} \Rightarrow \delta(m) < \mathit{ClearanceDuration}(m))$$

Next, we define the state resulting after an output transition.

The priorities of the lights that switch to green are reset to zero:

$$\pi' = \lambda l. \begin{cases} 0 & \text{if } l \in TG_\sigma \\ \pi(l) & \text{otherwise} \end{cases}$$

The switching lights receive their new colours:

$$\gamma' = \lambda l. \begin{cases} \mathit{red} & \text{if } l \in TR_\sigma \\ \mathit{yellow} & \text{if } l \in TY_\sigma \\ \mathit{green} & \text{if } l \in TG_\sigma \\ \gamma(l) & \text{otherwise} \end{cases}$$

If a light changes its colour, the colour duration of this light should be reset to zero:

$$\delta' = \lambda l. \begin{cases} 0 & \text{if } l \in TR_\sigma \cup TY_\sigma \cup TG_\sigma \\ \delta(l) & \text{otherwise} \end{cases}$$

The sensor states of the lights that switch to yellow are reset:

$$\xi' = \lambda s. \begin{cases} \mathit{false} & \text{if for some } l \in TY_\sigma, s \in \mathit{sensors}(l) \\ \xi(s) & \text{otherwise} \end{cases}$$

Delay. When time advances, we have to update the priorities and the duration of the current colours. This is expressed in the following transition rule.

$$(\pi, \gamma, \delta, \xi, \tau) \xrightarrow{t} (\pi', \gamma, \delta', \xi, \tau + 1),$$

where

$$\pi' = \lambda l. \begin{cases} \mathit{UpdatePrio}(l, \pi(l)) & \text{if } \gamma(l) = \mathit{red} \\ 0 & \text{otherwise} \end{cases}$$

$$\delta' = \lambda l. \delta(l) + 1$$

4.5 Analysis

This phase deals with the development of (mathematical) techniques which aid in validating expressions in our language. We have already mentioned some important properties of a good traffic regulation system. Three of these properties will be discussed in some detail: *safety*, *fairness* and *redundancy*.

Safety. The basic safety property of a regulated junction is that never two conflicting traffic streams are allowed to pass the intersection at the same time. Since we do not have any knowledge of states, prior to the initial state, we can only prove something slightly weaker: two conflicting traffic streams are never allowed to pass the intersection and as long as we cannot guarantee the initial safety of the traffic streams, we do not allow any traffic to enter the junction.

We subsequently introduce some additional notation to ease the reasoning about our system. We define the set of states Σ_R, as the set of states that are reachable via zero or more transitions (denoted by $\rightarrow^*$), starting in the initial state σ_0.

$$\Sigma_R = \{\sigma \in \Sigma \mid \sigma_0 \rightarrow^* \sigma\}$$

The set of traces, starting in the initial state and formed by the sequence of states that are reachable from the initial states *and* for which a stream has had a red light continuously is referred to as the *initialisation* of this stream. Then, a state σ is part of an initialisation of a traffic stream l iff $\mathit{initialising}_\sigma(l)$ holds, where *initialising* is defined as:

$$\mathit{initialising}_\sigma(l) = (\forall_{\sigma' \in \Sigma_R}\ \sigma_0 \rightarrow^* \sigma' \rightarrow^* \sigma \Rightarrow \gamma'(l) = \mathit{red})$$

Then, the safety property is formulated as follows: for all reachable states $\sigma \in \Sigma_R$ and all traffic lanes $l, m \in \mathit{InLanes}$, where $\mathit{Conflict}(l, m)$, we have either

(1) $\mathit{initialising}_\sigma(l) \wedge \mathit{initialising}_\sigma(m)$, or
(2) $\neg(\mathit{crossing}_\sigma(l) \wedge \mathit{crossing}_\sigma(m))$.

where the predicate *crossing* is as defined in Section 4.4,

The safety property holds for every expression in our language. We give an outline of the inductive proof. The safety property clearly holds for the initial state, as for the initial state requirement (1) holds for all lanes. Assume either (1) or (2) holds for state σ_1, and a successor state σ_2 is reached by one of the three transitions defined in Section 4.4. An input transition does not change the functions γ and δ, leaving both (1) and (2) unchanged, so the safety property also holds for state σ_2. If σ_2 is reached via an output transition, possibly (1) and (2) are violated. Violation of (1) is only serious if (2) does not hold. However, we observe that the only way in which (2) can be violated is when a lane becomes crossing, while it was not crossing before (i.e. in σ_1). A lane becomes crossing because it is in the set TG_{σ_1}. Then this lane must be in M_σ, which is conflict-free by definition, and, moreover, it can not be in conflict with any other lane that is at that moment crossing. Thus, (2) holds, thereby guaranteeing our safety property. Finally, if σ_2 was reached via a delay transition, remark that δ is incremented (but γ stays the same, leaving (1) unchanged). This will at best cause some incoming lanes not to

be crossing any more, i.e. (2) only becomes "more true". Therefore, again our safety property holds.

Note that in general there can be a number of states for which both (1) and (2) hold simultaneously. The bound T_{max} for the time it takes for (2) to hold for all lanes, is defined as

$$T_{max} = max\{t \in Duration \mid t = ClearanceDuration(l) \wedge l \in InLanes\}$$

The lower bound T_{min} defines the first moment since the start-up of the system, for which there is a subset of lanes $A \in NC$ for which (2) holds for all lanes in A. In other words, from time T_{min} we can start expecting to see green lights in (subparts of) the traffic junction. Note that up to time T_{min}, all traffic lights have had red lights.

$$\begin{aligned} T_{min} = min\{\ & t \in Duration \mid t = ClearanceDuration(l) \wedge l \in InLanes \wedge \\ & t \geq max\{ClearanceDuration(m) \mid Conflict(l, m)\}\ \} \end{aligned}$$

Remark that lanes will be able to start receiving a green light only after time T_{min}, but it may still take some time before the first green light is received, possibly up to (and including) time T_{max}.

Fairness. A desirable property of traffic regulation is *fairness*. In this case, by fairness we mean that all traffic lanes must always eventually receive a green light. We again first introduce some auxiliary notation. The set of all infinite *runs* of our system is the set $\mathcal{R}$, where a run $\rho \in \mathcal{R}$ takes on the following form, reflecting the three types of transitions as defined in Section 4.4.2:

$$\rho = (i \cdot \langle TG, TY, TR \rangle \cdot t)^{\omega}$$

Derived from the set of all runs $\mathcal{R}$ is the set $\mathcal{R}_o$, obtained from all runs from $\mathcal{R}$ where both the input events and the delay transitions are removed from the runs. As a shorthand, we refer to the k^{th} element in a run ρ as ρ_k. The fairness requirement, is then translated to the following mathematical expression:

$$\forall_{l \in InLanes}\ \forall_{\rho \in \mathcal{R}_o}\ \forall_{k \in \mathbb{N}}\ \exists_{k' \in \mathbb{N}}\ (k' > k \wedge \rho_{k'} = \langle TG, TY, TR \rangle \wedge l \in TG)$$

Notice that for poor instances of the *UpdatePrio* function, the property cannot hold (e.g. if *UpdatePrio* is represented by a decreasing function). Therefore, the fairness property relies strongly on the choice for *UpdatePrio*. Here, we restrict our attention to update functions characterised by strictly increasing linear functions, i.e. functions of the form $ax+b$, where $a \geq 1$ and $b > 0$. Notice that the x in the update function represents the previous value of the priority function, so in fact we are dealing with simple linear recursive equations.

Without giving a proof, we first state that our traffic regulation always eventually switches a number of traffic lights to green when using strictly increasing linear functions as update functions. This property is weaker than the fairness property, but is useful in proving fairness, as it expresses that it cannot be the case that our regulation prevents traffic lights from obtaining a green light.

$$\forall_{\rho \in \mathcal{R}_o}\ \forall_{k \in \mathbb{N}}\ \exists_{k' \in \mathbb{N}}\ (k' > k \wedge \rho_{k'} = \langle TG, TY, TR \rangle \wedge TG \neq \emptyset)$$

We now give a sketch of the fairness proof. Let $l \in \mathit{InLanes}$ be an arbitrary lane, and ρ an (infinite) run in $\mathcal{R}_o$, containing only output events. Let k be an arbitrary natural number. It suffices to show there always exists a $k' > k$, such that $\rho_{k'} = \langle TG, TY, TR \rangle$ and $l \in TG$.

Now, let $\mathcal{M}$ be the set of natural numbers $m > k$, where $\rho_m = \langle TG, TY, TR \rangle$ such that $\pi(l)$ is strictly maximal (i.e. there is no other lane l', with $\pi(l) \leq \pi(l')$). Note that whenever $\mathcal{M} = \emptyset$, this means that the priority of l is reset to zero at least once. This, however, can only happen when l receives a green light. Hence, we can safely assume $\mathcal{M} \neq \emptyset$.

Then, due to our definition of M_σ, we know that for all ρ_m, $l \in M_\sigma$. Moreover, we have an upper bound to the time it takes for l to be consecutively in M_σ without being set to green (i.e. $l \notin TG$). This bound is defined as τ_{min}, where

$$\begin{aligned} \tau_{min} = {} & \mathit{MinStateTime}(\mathit{green}) + \mathit{MinStateTime}(\mathit{yellow}) \\ & + \mathit{max}\{\, \mathit{ClearanceDuration}(l') \mid l' \in \mathit{InLanes} \wedge \mathit{Conflict}(l, l') \,\} \end{aligned}$$

Hence, if there is an interval where $\mathcal{I} = [t, t + \tau_{min}] \subseteq \mathcal{M}$, for some t, then we know for sure that $\rho_{t+\tau_{min}} = \langle TG, TY, TR \rangle$ and $l \in TG$.

Now, suppose there is no such interval $\mathcal{I}$, i.e. for all intervals $\mathcal{I}$ of $\mathcal{M}$, we have $|\mathcal{I}| < \tau_{min}$. This can only be the case if there are other traffic lights, that have priority functions that always overtake the priority function of l. Since all priority functions are linear functions, eventually one of these lanes must receive a green light (since always eventually some traffic light is set to green) and its priority is reset to zero. Although this can happen very often, it can only happen a finite number of times, as eventually $\pi(l)$ becomes so large it cannot be overtaken in τ_{min} time units. Therefore, there must exist an interval $\mathcal{I}$ where $|\mathcal{I}| \geq \tau_{min}$, and subsequently, l is set to green.

Redundant Traffic Lights. Apart from verifying whether our language satisfies the design criteria, such as fairness and safety, there are many more interesting conclusions that can be drawn when analysing the language, such as *throughput* and other efficiency measures. As an example, we show that our language is capable of efficiently dealing with redundant traffic lights.

More concretely: if a traffic lane has no conflicts with other traffic lanes, it will eventually receive a continuous green light. Formally, this is expressed as follows:

$$\forall_{l \in InLanes}\ (\forall_{m \in InLanes}\ \neg \mathit{Conflict}(l, m) \Rightarrow (\exists_{\sigma_1 \in \Sigma_R}\ \forall_{\sigma_2 \in \Sigma_R}\ \sigma_1 \rightarrow^* \sigma_2 \wedge \gamma_2(l) = \mathit{green}))$$

For every expression in the language, redundant traffic lights are detected. We proceed as follows. First, we show that traffic lanes for which there are no conflicts are always nominated to be set to green. Suppose $l \in \mathit{InLanes}$ and for all $m \in \mathit{InLanes}$, we have $\neg \mathit{Conflict}(l, m)$. This means that $l \in A$ for all $A \in NC \setminus \{\emptyset\}$, where NC is as defined in section 4.4.2. Note that $\emptyset \lesssim \{k\}$ for all $k \in \mathit{InLanes}$. Hence, $l \in M_\sigma$, for all states σ, no matter the distribution of the priorities. Second, we show that always being nominated to get a green light culminates in eventually receiving a green light. Now, for all states $\sigma \in \Sigma_R$ with $\tau = T_{\mathit{MinStateTime}}(\mathit{red})$, we have $l \in TG_\sigma$ (this is the first moment we can guarantee safe passage over the crossing via l). This means that in the immediate successor state of state σ, the colour of lane l will be green, i.e. for all $\sigma_1 \in \Sigma_R$, with $\sigma \rightarrow \sigma_1$, we know $\gamma_1(l) = \mathit{green}$. Now, it suffices to prove that for all successor states $\sigma_2 \in \Sigma_R$ of σ_1, we have $\gamma_2(l) = \mathit{green}$. This, however, follows from the fact that never $l \in TR_{\sigma_2} \cup TY_{\sigma_2}$, as l has no conflicts with any other lane. Hence, we know that for all σ_2, reachable from σ_1 in zero or more steps, $\gamma_2(l) = \mathit{green}$ holds, and in effect, the traffic light for l is redundant.

4.6 Pragmatics

Now that we have constructed the actual language, we can have a look at the pragmatics of using the language. As stated before, the pragmatics is concerned with all aspects of using the language. We restrict our discussion to a few interesting aspects.

4.6.1 Methodology

An important part of the methodology is the documentation of the language. This is needed to make clear what well-formed programs are and what they mean. This has already been described in some detail in the previous sections on syntax and semantics. However, for proper use of the language, a designer of a regulated intersection will need more information.

The first step a designer has to take is to determine the physical layout of the intersection. One can derive this from the existing situation, or in some cases

it must be developed from scratch. A designer will need guidelines in order to be able to determine the physical structure, e.g. concerning optimal throughput for a given traffic intensity, side conditions due to legislation, cost, etc. Although this is not part of the language proper, the language cannot be effectively used without such methodological issues. The layout of the intersection is (in an abstract way) represented in the program. Since also the grouping of lanes is taken into account, there must be guidelines of how to sensibly form such groups.

Given the physical layout, it is not necessary that all traffic streams that cross each other are in conflict. Some crossings may be considered harmless, e.g. because there is only little traffic. Therefore, the developer also needs to develop the conflict matrix, as a subset of all possible conflicts. Guidelines with respect to this issue should also be covered in a methodology handbook. The notion of a priority update function is very specific to our developed language and not likely to be generally known by developers of intersections. Furthermore, the selected function will have quite some impact on the actual behaviour of the system. A priority function which grows linearly will make a traffic stream less important than one with exponential growth. Therefore, a number of guidelines on which functions to use in which situations are necessary.

From a different perspective, one should not only describe proper use of the language, but also discourage improper use. One could, for instance, use the value of the clearance duration of one traffic stream to regulate the relative priority of conflicting streams. This could be considered bad style.

Another important function of the methodology is to explain when to use which tools to obtain certain results.

4.6.2 Tool support

Clearly, a set of tools should come with the language in order to facilitate the development of a regulated intersection. Ideally, these tools form an *Integrated Development Environment* (IDE). Here, we just mention some interesting tools. In section 4.6.3, we describe our experiences in developing a prototype of a simulation tool.

The first step is to design tool support for producing expressions in our language. Normally, one would expect tools for (syntax directed) editing, parsing, static checking, etc. Since part of the language is concerned with describing a two-dimensional layout, a graphical editor will show useful. There could be standard components like traffic lights, lanes, and sensors that can be "dragged-and-dropped" to the desired locations. This tool should be able to transform the visual model into the correct program text, and vice versa.

After a model of a traffic junction has been constructed or an existing model is loaded, we want to be able to analyse the model with respect to some of the issues that were raised in Section 4.5.

Two concepts that influence the throughput of the junction are the update functions and clearance duration. To optimise the throughput of the junction we have to build a tool that can assign stochastic distribution functions to the traffic streams so the mean, variance, and other statistical data for the waiting times of these streams can be calculated. Apart from writing a new tool to do this, we can also export the parameters of the junction into an existing statistical tool. Once we have the statistical data we can use this information to recommend values for clearance durations and the update functions. This can be done by building an expert system to calculate the optimal throughput for the given junction.

Finally, the given model must be implemented to control the target junction. Ideally, the behaviour represented in the model could be compiled into the command language of the device actually controlling a junction. Alternatively, an interpreter of the language could be run which controls the sensors and actuators via an appropriate interface. The latter option would allow for remote control of intersections (e.g. via Internet).

4.6.3 Experiences with building a simulation tool

In order to validate the work on syntax and semantics as expressed in the previous paragraphs, we have built a prototype of a simulation tool. After explaining the basic functionality of the tool, we discuss what we have learnt from this implementation effort.

The purpose of the simulation tool is to support a user in understanding and fine-tuning the behaviour of a controlled traffic junction. The prototype supports the following features:

- **Specification input and output**
  The program can read an existing specification, check its syntax and static requirements. It can save a specification.
- **Specification display and editing**
  The program can display a specification, supports graphical editing of the specification, and allows to interactively create a new specification.
- **Graphical simulation and dynamic control** The program can graphically simulate the dynamic behaviour of a specification, while giving the user control over e.g. simulation speed. Traffic arrives at random at the junction. During simulation the user can modify non-structural parts of the specification, such as the clearance duration.

We implemented the prototype in Java, using the parser generator Javacc to define the parser and the Java Swing library for developing the user interface. The prototype runs on any machine supporting a recent Java environment. The performance of the simulator is approximately 500 transition per second (on a Pentium II processor) when running the simulator on a junction of 7 lanes. Developing the simulation kernel (i.e. implementing the transition rules and appropriate data structures) took about two days of work. The implementation of the parser and the Graphical User Interface (GUI) took several weeks. Figure 3 shows a screen-shot of the simulation window of the tool. It displays a traffic junction after 75 time units (three transitions are taken every time unit). The vertical bar to the right of each traffic light dynamically displays its priority. The number within a traffic light displays the time a traffic light already has the current colour.

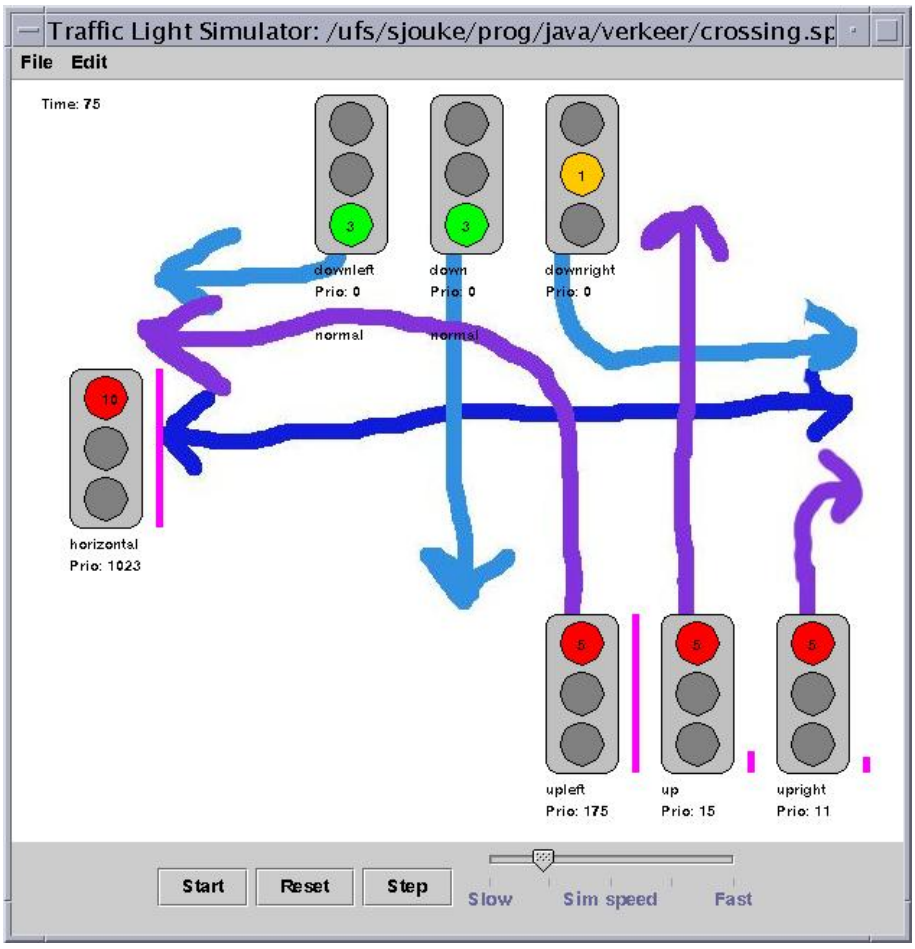


Fig. 3. Screen-shot of the prototype simulator

Building the simulator and experimenting with it gave us several new insights. First of all, it gave evidence that the semantical definitions in our case study were sound. The simulation engine, being an implementation of the transition rules, did not reveal any errors in these rules. However, the implementation of the simulation kernel was less direct than expected. The main reason is that

the formal definitions are mostly declarative, whereas we used an imperative language for implementation. The step from the mathematical definitions to the implementation language was performed in an ad-hoc manner, without paying much attention for efficiency and formal correctness. We expect that formal techniques for program transformation and derivation will be essential to make a correct and efficient implementation of the simulation kernel.

The development of the prototype implied some minor modifications of the proposed syntax of our language. We decided e.g. to restrict the priority update functions to the class of linear functions. More interestingly, it showed necessary to extend the syntax with graphical information, having no semantical interpretation. We decided e.g. to remember the position of the traffic lights on the computer screen by adding their coordinates to the specification.

It was our experience that the larger part of the programming effort was dedicated to building the graphical user interface. We can think of two reasons for this. First, building a GUI is –even with an extensive library like Swing– a non-trivial task [2] . Second, user interfaces did not play any role in the domain analysis conducted above, as a result of which, the developed language has no provisions for this aspect. Having learnt now that the GUI is an important (and time consuming) part of the program, we could start a new cycle in our language development methodology and focus on additional primitives to support the proper design of a GUI for a simulator.

Although experimenting with the prototype did not imply any (direct) modification of the developed semantics, it did result in some new insights with respect to the domain analysis. As an example, we mention that in the design the priority update function is attached to a traffic stream. While playing with the tool, we found out that this is not flexible enough. It would have been more appropriate to have separate update functions for each sensor. This again strengthens our believe that language development is not a sequential process. Domain analysis, syntax definition, semantics definition and pragmatics should be developed in an interwoven way.

Finally, the prototype made clear that a language-driven system development method does not necessarily has to result in language centred tools. In fact, the prototype can be used without having to see any expression in the language at all. This is because the GUI allows the user to create, edit, and display a traffic junction graphically. The syntactical format is just for saving junctions on disk. When developing e.g. a word processing system using the language-driven approach, both a language centred system like LaTeX, and a WYSIWYG system like MS-Word could be the resulting tool.

[2] One could consider Swing itself as a DSL for GUIs, and might question whether it is designed in the best way. An experiment in developing such a system with our methodology would be an interesting case study.

In summary, we can say that our implementation experiment, although only yielding a simple prototype, strengthened our confidence in the theoretical development, and thus in our proposed approach.

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 instances of the relevant concepts, as to better accomodate 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, confirming our believes that language development (and software engineering) are iterative activities. 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.

Our approach is centred around the development of a language, however, this does not result in language centred tools *per se*. This is also confirmed by our case study: in a prototype implementation of a simulator for the language, the defined syntax was only used for storage and retrieval of language expressions. Other issues could all be resolved graphically.

It is a generally accepted fact that it is preferable to detect errors during the early and more abstract phases of system design. The language-driven approach focuses the attention of the developers on the basic concepts of the

language and therefore on (the building blocks of) the semantics. When developing a software system these abstract building blocks must be completely understood. In our case study, we experienced that the discussion focussed mainly on the concepts and semantics. A traditional development process would have focussed more on the design and the implementation of the system.

Because the traffic light case study focuses on a relatively small domain, we cannot assess the applicability of the language-driven approach in a large domain. It must be investigated whether techniques such as top-down design and modularisation, which have a natural place in traditional life-cycle models, also have their counterpart in our approach. A natural way of dealing with larger problems is to identify substructures in the problem domain, which can be dealt with in isolation. The sub-domains give rise to a number of problem spaces, each with their own language. The composition of these partial solutions can e.g. be defined with a co-ordination language.

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] L. Aceta, W.J. Fokkink, and C. Verhoef. Structural operational semantics. In Jan A. Bergstra, Alban Ponse, and Scott A. Smolka, editors, *Handbook of Process Algebra*, pages 197–292. Elsevier (North-Holland), 2001.

[2] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[3] B.W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[4] 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.

[5] 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.

[6] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, 2000.

[7] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[8] R.E. Faith, L.S. Nyland, and J.F. Prins. Khepera: A system for rapid implementation of domain specific languages. In J.C. Ramming, editor, *Proceedings of the USENIX Conference on Domain-Specific Languages*, 1997.

[9] 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.

[10] J. Heering and P. Klint. Semantics of programming languages: A tool-oriented approach. *ACM Sigplan Notices*, 35(3):39–48, 2000.

[11] M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction using Structured Operational Semantics*. Wiley, New York, 1991.

[12] R.M. Herndon and V.A. Berzins. The realizable benefits of a language prototyping language. *IEEE Transactions on Software Engineering*, 14:803–809, 1988.

[13] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[14] ITU-T. *ITU-T Recommendation Z.106: Common Interchange Format for SDL*. ITU-T, Geneva, 1996.

[15] 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.

[16] P. Klint. A meta-environment for generating programming environments. *ACM Transactions of Software Engineering and Methodology*, 2(2):176–201, 1993.

[17] D.E. Knuth. *Semantics of Context-Free Languages*, volume 2, pages 127–145. Springer-Verlag, New York, 1968.

[18] P.W. Kutter and A. Pierantonio. Montages specifications of realistic programming languages. *Journal of Universal Computer Science*, 3(5):416–442, 1997.

[19] 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.

[20] R. Prieto-Díaz. Domain analysis: An introduction. *Software Engineering Notes*, 15(2):47–54, 1990.

[21] J. Rekers and A. Schürr. A graph grammar approach to graphical parsing. In *Proceedings of the 1995 IEEE Symposium on Visual Languages*, 1995.

[22] W.W. Royce. Managing the development of large software systems. In *Proceedings of the IEEE WESCON*, 1970.

[23] D.A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., Newton, MA, 1986.

[24] K. Slonneger and B. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley, 1995.

[25] S. Thibault. *Domain-Specific Languages: Conception, Implementation and Application*. PhD thesis, IRISA/University of Rennes 1, 1998.

[26] 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.

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

[28] S.F.M. van Vlijmen and A. van Waveren. Algebraic specification of a system for a traffic regulation at signalized intersections. Technical Report P9313, Programming Research Group, University of Amsterdam, 1993.

[29] 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.

[30] 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.