Centro de Informática U·F·P·E

Pós-Graduação em Ciência da Computação

"A rigorous methodology for developing GUI-based DSL formal tools"

Por

Robson dos Santos e Silva

Dissertação de Mestrado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, AGOSTO/2013

Universidade Federal de Pernambuco
Centro de Informática
Pós-graduação em Ciência da Computação

Robson dos Santos e Silva

“A rigorous methodology for developing GUI-based DSL formal tools”

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: *Alexandre Cabral Mota*

RECIFE, AGOSTO/2013

Catalogação na fonte
Bibliotecária Jane Souto Maior, CRB4-571

Santos e Silva, Robson dos
A rigorous methodology for developing GUI-based DSL formal tools / Robson dos Santos e Silva. - Recife: O Autor, 2013.
93 f.: il., fig.

Orientador: Alexandre Cabral Mota.
Dissertação (mestrado) - Universidade Federal de Pernambuco. CIn, Ciência da Computação, 2013.

Inclui referências e apêndice.

1. Ciência da Computação. 2. Engenharia de Software. 3. Métodos formais. I. Mota, Alexandre Cabral (orientador). II. Título.

004 CDD (23. ed.) MEI2013 – 118

Dissertação de Mestrado apresentada por **Robson dos Santos e Silva** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**A rigorous methodology for developing GUI-based DSL formal tools**" orientada pelo **Prof. Alexandre Cabral Mota** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Marcio Lopes Cornélio
Centro de Informática / UFPE

Prof. Adalberto Cajueiro de Farias
Departamento de Sistemas de Computação / UFCG

Prof. Alexandre Cabral Mota
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 23 de agosto de 2013.

Profa. Edna Natividade da Silva Barros
Coordenadora da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

I dedicate this dissertation to my family, friends and professors who supported me with all necessary to get here.

Acknowledgements

Firstly to God for every blessing, protection and strength throughout my life.

To my mother for all support, love and friendship. I am very grateful for your wisdom and understanding that even geographically distant we are always together.

My special thanks to my advisor Alexandre Cabral Mota. I am grateful for the excellent academic orientation, the suggestions in this work and for all confidence given to me.

Thanks to Franklin Ramalho, Juliano Iyoda, Rodrigo Starr and Edson Watanabe for valuable comments.

To my friends for their friendship and understand the times I have been missing.

Resumo

A Engenharia Dirigida a Modelos ou (MDE—Model-Driven Engineering) é uma metodologia de desenvolvimento de *software* que se concentra na criação e manipulação de modelos específicos de domínio. É comum o uso de linguagens específicas de domínio (DSL) para descrever os elementos concretos de tais modelos.

Ferramentas de MDE podem facilmente construir linguagens específicas de domínio (DSL), capturando seus aspectos sintáticos assim como sua semântica estática. No entanto, ainda não possuem uma forma clara de capturar a semântica dinâmica de uma DSL, assim como a verificação de propriedades de domínio antes da geração de código executável. Métodos formais são tidos como uma solução para prover *software* correto, onde podemos garantir que desejadas propriedades são satisfeitas.

Infelizmente, as ferramentas de métodos formais disponíveis concentram-se quase que exclusivamente na semântica enquanto que a interação homem-computador é "deixada para o usuário". Indústrias em que a segurança é crítica, usam representações matemáticas para lidar com os seus domínios de problemas. Historicamente, essas representações matemáticas têm um apelo gráfico. Por exemplo, Cadeias de Markov e Árvores de Falha.

Em geral, devido à dificuldade em obter *softwares* formalmente verificados, essas indústrias utilizam sistemas comerciais prontos para uso (Commercial Off-the-shelf - COTS) ou os constróem especificamente para satisfazerem as suas necessidades com um esforço considerável em testes. Tais DSLs são difíceis de capturar, usando apenas ferramentas MDE por exemplo, porque possuem uma semântica particular para prover as informações específicas desejadas para as indústrias que as utilizam.

Neste sentido, dada uma DSL (L), composta por sintaxe e semântica estática (SS_L), e semântica dinâmica (DS_L), este trabalho propõe uma metodologia rigorosa para combinar a facilidade de ferramentas MDE em capturar SS_L, com a corretude assegurada por métodos formais para capturar DS_L e verificar suas propriedades. Esta combinação é especificamente tratada da seguinte maneira: captura-se todos os aspectos de L utilizando métodos formais, verificam-se as propriedades desejadas e as ajustam caso necessário. Em seguida, parte de L é traduzida automaticamente em termos de artefatos para uma ferramenta MDE, a partir da qual é possível construir uma interface amigável (*front-end*) facilmente (automaticamente). Por fim, o código do *front-end* é integrado com o código sintetizado automaticamente a partir da semântica dinâmica formal (*back-end*).

Palavras-chave: Engenharia Dirigida a Modelos (MDE), Métodos Formais, Linguagens específicas de domínio, Ferramentas formais com interface gráfica

Abstract

It is well-known that model-driven engineering (MDE) is a software development methodology that focuses on creating and exploiting (specific) domain models. Domain models (conceptually) capture all the topics (for instance, entities and their attributes, roles, and relationships as well as more specific constraints) related to a particular problem. It is common to use domain-specific languages (DSL) to describe the concrete elements of such models.

MDE tools can easily build domain-specific languages (DSL), capturing syntactic as well as static semantic information. However, we still do not have a clear way of capturing the dynamic semantics of a DSL as well as checking the domain properties prior to generating the implementation code. Formal methods are a well-known solution for providing correct software, where we can guarantee the satisfaction of desired properties.

Unfortunately the available formal methods tools focus almost exclusively on semantics whereas human-machine interaction is "left to the user". Several industries, and in particular the safety-critical industries, use mathematical representations to deal with their problem domains. Historically, such mathematical representations have a graphical appeal. For example, Markov chains and fault-trees are used in safety assessment processes to guarantee that airplanes, trains, and other safety-critical systems work within allowed safety margins. In general, due to the difficulty to obtain correct software, such industries use Commercial Off-The-Shelf (COTS) software or build them specifically to satisfy their needs with a related testing campaign effort. Such DSLs are difficult to capture, using just MDE tools for instance, because they have specific semantics to provide the desired (core) information for the industries that use them.

In this sense, given a DSL (L) composed of a syntax and static semantics (SS_L), and dynamic semantics (DS_L) parts, our work proposes a rigorous methodology for combining the easiness of MDE tools, to capture SS_L, with the correctness assured by formal methods, to capture DS_L as well and check its properties. This combination is specifically handled in the following way, we capture all aspects of L using formal methods, check the desired properties and adjust if necessary. After that, we automatically translate part of it in terms of constructs of a MDE tool, from which we can build a user-friendly (GUI) front-end very easily (automatically). Finally, we link the front-end code to the automatically synthesized code from the formal dynamic semantics back-end.

Although we require the use of a formal methods tool, the distance from the mathematical representations used in industry and the formal methods notation is very close. With this proposed methodology we intend that safety-critical industries create their domain specific software as easy as possible and with the desired static and dynamic properties formally checked.

Keywords: Model-Driven Engineering, Formal Methods, Domain-Specific Languages, GUI-based formal tools

Contents

List of Figures

List of Acronyms

MDE	Model-Driven Engineering
DSL	Domain-Specific Language
EMF	Eclipse Modeling Framework
GMF	Graphical Modeling Framework
OCL	Object Constraint Language
JML	Java Modeling Language
FT	Fault Tree
PD	*Perfect* Developer Tool
PL	*Perfect* Language
PL2EMF	Tool that automates the process of generating a metamodel and constraints from a formal specification
SDF	Syntax Definition Formalism
EWL	Epsilon Wizard Language
MIC	Model-Integrated Computing
EVL	Epsilon Validation Language
MDA	Model Driven Architecture
AST	Abstract Syntax Tree

1

Introduction

1.1 Motivation

Model-Driven Engineering (MDE) is a software development philosophy that focuses on the development process of building high-level technology-independent models for a specific problem domain Bézivin (2005). In MDE, modeling is essential and descriptions of problem domains are given in terms of Domain-Specific Languages (DSLs).

DSLs are defined in (Mernik *et al.*, 2005) as languages tailored to a specific application domain. This means that DSLs are languages designed to target particular domains, filling the lack of expressivity of general purpose languages, in their domain of application, focusing on the relevant concepts and features of that domain.

Ad hoc DSL development is hard and requires domain knowledge and language development expertise that few people have Mernik *et al.* (2005). On the other hand, following MDE principles and approaches, the development of DSLs has the advantage of not requiring any specific expertise in programming because the focus of the development is on modelling all the relevant information of the domain of interest using some MDE notation. As a consequence, there are several ways of creating DSLs based on metamodeling and following the MDE principles, where the Eclipse Modeling Framework (EMF) is the most well-known solution Steinberg *et al.* (2008).

Within EMF, the definition of a DSL syntax is usually given using meta-languages such as ECore (Steinberg *et al.*, 2008) (used to specify metamodels) and static semantics with Object Constraint Language (OCL) (OMG, 2013). Additionally, OCL provides a limited support to dynamic semantics (that describes the dynamic behavior of a language) through the definition of pre/post-conditions on operations (Gargantini *et al.*, 2009b).

However, metamodeling frameworks still do not have some standard manner to provide static and/or dynamic semantics in a rigorous way (Jackson *et al.*, 2011). Furthermore, it is rarely used to verify formally, by means of an automatic inference engine and theorem prover, certain properties about the DSL being specified. So, instead of using only metamodeling frameworks to define DSLs we need to rely on the support of formal methods.

Formal methods are a particular kind of mathematically based techniques for the specification, development and verification of software and hardware systems. The use of formal methods for the specification of systems brings many benefits. For instance, a deeper understanding of the problem, avoiding the occurrences of errors during the early stages of the development process (statistically the hardest errors to find and fix (Boehm, 1981)) as well as checking the domain properties prior to generate the implementation code. Nowadays, there are several formal tools such as ESC/Java2 (Cok and Kiniry, 2005) and JML RAC (Chalin and Rioux, 2008) for Java Modeling Language (JML) (Leavens and Cheon, 2006), *Perfect* Developer (Escher, 2012) and SCADE (Esterel Technologies, 2012) that can generate formally verified code based on formal specifications.

However, formal methods in general do not provide mechanisms to easily construct DSL with graphical user interfaces (GUIs). On the other hand, MDE has weaknesses exactly where formal methods are very good: (i) capture the complete dynamic semantics of a language, or its business-like functionalities and (ii) verify formally certain properties about the DSL being specified. As consequence, it is difficult to find an integrated solution involving correct code derived from formal specifications with a user-friendly GUI-based DSL front-end. We see this shortcoming as a valuable opportunity to combine formal methods with MDE approaches.

1.2 Proposal

Our main proposal in this work is showing how we can combine metamodeling and formal methods to create correct software in a productive way. To accomplish this, we propose a rigorous methodology to create GUI-based domain-specific languages (DSLs) formal tools. The key point is a formal specification of a DSL (L) capturing the: syntax and static semantics of L (SS_L), as well as its dynamic semantics (DS_L). From the formal specification of L we can check desirable properties, adjust if necessary, and translate part of it (SS_L) in terms of modeling artifacts. Particularly, from SS_L we show how a metamodel MM and a set of constraints SC_{MM} over MM (Figure 1.1) can be extracted automatically by using systematic translation rules that we have developed and implemented.

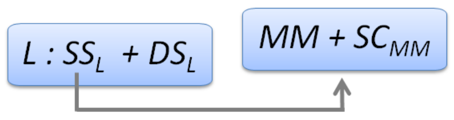


Figure 1.1 A metamodel *MM* and a set of constraints SC_{MM} obtained from SS_L of a DSL L

The metamodel *MM* and its constraints SC_{MM} are the primary artifacts to generate a user-friendly constraint preserving front-end for the DSL. Additionally, from the formal specification L we generate the executable back-end by using an automatic code generator. Finally, the relationship among the metamodel *MM*, its constraints SC_{MM} and the formal specification L serves as the glue to connect the GUI to the executable back-end. Therefore, following our methodology it is only necessary to focus on the creation of the intended DSL formal specification.

To illustrate the proposed methodology, we create a Fault Tree (FT) Analysis tool. FT is a domain-specific formalism commonly used in critical systems (NASA, 2002). We formally specify an FT using the *Perfect* language (Escher, 2012) and prove properties of this formalization using the *Perfect* Developer Tool (Escher, 2012). We then extract a metamodel and constraints from this specification to create a constraint preserving GUI front-end using a traditionally available metamodeling technology. Currently the metamodel and constraint extraction is performed by a tool we also have developed, called *PL2EMF*. This tool and the executable FT analysis tool can be found in (Silva, 2013).

Translating languages always demand attention concerning completeness and soundness. Ideally one has to prove that the translation is sound and complete. With respect to completeness, it is almost direct when the translation rules cover all productions of the grammar of the source language. Concerning soundness, instead of a formal proof, for this work we investigate it indirectly by exercising invariants and operation contracts through the constraint-preserving generated GUI.

Several industries can take advantage of this methodology, in particular the safety-critical industries that use mathematical representations to deal with their problem domains. We aim at providing a productive and trustworthy development methodology to safety critical industries. Based on our proposal we expect such industries to be able to create specific systems as easily as possible and with the desired static and dynamic properties formally checked. In particular we are experimenting our proposed strategy on case studies of our industrial partner and obtaining good productivity.

Although we base our work on formal specifications, that the MDE community can consider too difficult and of great effort to work with in practice, we argue that for our scope, these formalisms are considerably simple rewritten of mature mathematical models.

1.3 Dissertation Structure

The remainder of this dissertation is organized as follows: In Chapter 2 we provide an overview of the undelying concepts, approaches and technologies that form the basis of our proposed rigorous methodology (presented in Chapter 3). In Chapter 4 we detail some steps of our methodology. In Chapter 5 we detail how the *PL2EMF* was developed. Chapter 6 shows our case study: a Fault Tree Analysis Tool. In Chapter 7 we present related works and provide some conclusions and future work.

2
Background

In this chapter, we provide an overview of the undelying concepts, approaches and technologies that form the basis of our proposed rigorous methodology. In Section 2.1 we present the contract-based *Perfect* Language formalism and its code synthesizer tool (*Perfect* Developer). In Section 2.2 we give a brief overview about the Model Driven Engineering (MDE) principles and in Section 2.3 we present Model Driven Architecture (MDA) approach. In Sections 2.4, 2.5 and 2.6 we present some tools and frameworks of the Eclipse environment that conforms to MDA approach.

2.1 The *Perfect Developer* Tool

The *Perfect Developer* Tool (PD for short) is part of the *Escher Verification Studio* (Escher, 2012). PD uses a powerful automatic inference engine and theorem prover to reason about the requirements, contract-based specification and code. PD[1] is a contract-based tool able to specify and refine specifications of software systems using the *Perfect* language (*P*L for short). It is also able to generate ready-to-compile code (in Java, C++, C# or ADA) directly from the specification. This code can be a functional prototype or the final code for a system.

The PD tool is able to use automated reasoning to understand the descriptions and to verify whether the requirements are met. The proof obligations are generated and (almost all of them are) automatically proved by the tool. When the theorem prover is not able to discharge a proof obligation automatically, it generates a report to the user which can be used to give hints to the theorem prover about how the proof can be achieved.

A specification in *P*L is composed of a set of related classes, each one with their

[1]PD runs under Linux and Windows (XP, Vista and 7). In this work we use the academic license, Version 5.0.

variables, invariants, schemas, functions, pre and post-conditions, constructors, (loop) invariants, variants, properties, axioms, assertions, and so on. Variables define the model state-space, invariants constrain this state-space, *schemas* are operations that change the state and *functions* are side-effect free operations. *Schema* names are prefixed with the symbol '!' to denote that they change the system state. A class definition in *P*L is divided into sections (abstract, internal, confined and interface), each one with its specific elements.

Variables declared in the ***abstract*** section represent the abstract data model of the class (its abstract state-space). Inside this section we can have, for example, invariants that define what must be true independently of the current model state. We can have several levels of data refinements of the abstract class data model. The ***internal*** section (not shown here), is used to declare the data refined. For each level of refinement, it is required to define a *retrieve relation* between the previous level and the current level of refinement. The ***confined*** (not shown here) and ***interface*** sections are used to declare the public interface of a class. The main difference between the elements declared in these sections is that the elements of ***confined*** are only accessible by derived classes and the elements of ***interface*** are also visible by non-derived classes (Escher, 2012).

To illustrate some of the main constructs of the *Perfect* Language we specify a generic stack data structure as shown in Figure 2.1.

The main aspect of *P*L that should be noted here is that it allows, using a unique notation, the definition of behavioral and structural features, constraints and properties. For a complete description of *P*L, see (Escher, 2012).

We start by defining the class *SItem* that models items to be stored in the stack data structure. We declare this class simply as an alias for the *P*L *string* type.

Secondly, we define the class *StackNode* that models a stack node. Each node is composed by an *item*, of the type *SItem*, to be stored in the data structure and a reference *prox* to the next stack node. The statement *import "SItem.pd"* is used to invoke the definition of the class *SItem* in the *StackNode* context.

In *P*L, the *null* value has a specific type called *void*. The union (contruct "||") of the types *StackNode* and *void* enable the *null* value to be assigned to the variable *nxt*.

We now define the *Stack* class that defines the intended data structure itself. In the ***abstract*** section, two variables are declared: *top* is a reference to the stack top and *size* is an integer variable that stores the stack size. We declare, in section ***invariant***, that *size* must be always greater than or equal to zero. This invariant must be true regardless of the stack state.

```
class SItem ^= string;

import "SItem.pd";
class StackNode ^=
abstract
  var item: SItem;
  var prox : StackNode || void;
interface
  function item;
  function prox;
  build{itm: SItem,
      px: StackNode || void}
    post item! = itm,
         prox! = px;
end;
```

```
import "SItem.pd"; import "StackNode.pd";
class Stack ^=
abstract
  var top : StackNode || void;
  var size : int;
  invariant size >= 0;
interface
  function empty: bool ^= ...
  function validate (itm: SItem): bool ^= ...

  schema !push(newItem: SItem)
    pre validate(newItem)
    post top! = StackNode{newItem, top},
         size! = size + 1;

  schema !pop(x!: out SItem)
    pre ~empty
    post x! = (top is StackNode).item,
        top! = (top is StackNode).prox,
        size! = size - 1;
  build{}
    post size! = 0, top! = null;

  property (x: SItem)
    assert ~(self after it!push(x)).empty;

  ghost schema !pushToEmptyThenRemove(
        e: SItem, r!: out SItem)
    pre empty
    post !push(e) then !pop(r!)
    assert r' = e;

  ghost schema !pushTwoToEmptyThenRemove(
        e1, e2: SItem, r!: out SItem)
    pre empty
    post !push(e1) then !push(e2) then !pop(r!)
    assert r' = e2;

end;
```

Figure 2.1 A Stack Model specified in *Perfect* Language

In the ***interface*** section, we declare (not shown here) two functions: *empty*, that checks whether the stack is empty, and *validate* that verifies if the item to be inserted in the stack is valid with respect to specific restrictions. These restrictions depend on how *SItem* is modeled for a particular context. These functions return a boolean value and the second one has a *SItem* as input parameter

Besides the fact that a schema can change the state of the model, it also differs from a function by not being able to return values explicitly to the caller. The schema *push* receives as input a new value *newItem* of type *SItem*. Before adding this new value to the stack, (i) *push* checks whether this new item is valid and if this is true, (ii) the *newItem* is added at the the top of the stack and the stack size is incremented by one. Otherwise, the state of the model remains unchanged.

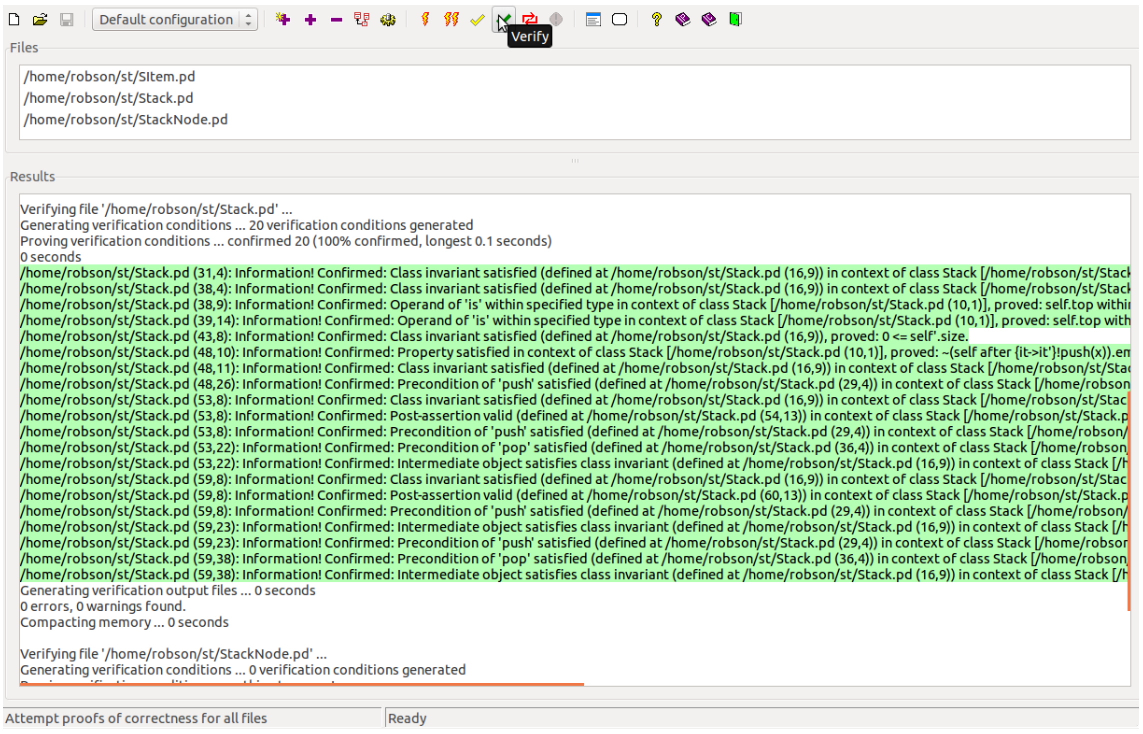
The statement (i) is the precondition (***pre***) of the *push* operation, that is, what the state of the model should be, for this operation to be executed successfully. The statement (ii) is the postcondition (***post***) of the *push* operation. It states how the variables *top* and *size* will be changed (*st!* and *size!*, respectively) after the *push* operation if the precondition is satisfied. The class constructor (***build***) initializes the variable *size* as 0 (zero) and *top* as *null*.

The schema *pop* removes the top element of the stack if is not empty. This schema has an input parameter *x*. Note that this input parameter syntax, *x!: out SItem*, differs from the syntax used in the schema *push*. This special syntax is used to simulate a call by reference. So, the parameter *x* works as a return of the *pop* operation. The expression *(top is StackNode)* is a cast of the *top* variable to the *StackNode* type.

*P*L allows to define properties of a class. Properties are used to verify expected behaviors of all instances of a class. We can define properties by using the ***property assert*** clause or by writing ***ghost*** schemas. A ghost schema is one for which no executable code is generated. It goal is to generate proof obligations over the defined specification.

We define a ***property*** that this model should have. We use property declarations to express theorems. The first line, ***property*** *(x: SItem)*, is an implicit universal quantification over the parameter. We can also declare, using the keyword ***pre***, facts to be assumed (not used in this case). The consequence, ***assert*** ~(***self after it***!push(x)).empty, states that we can insert a value in the stack and after this insertion, the stack becomes not empty.

The *pushToEmptyThenRemove* ghost schema verify, see its ***assert*** clause, that if we push an element to an empty stack, the next element we pop will be the one we pushed. The ***then*** clause is used to define the order (left to right) in which the *push* and *pop* schemas are called. The *pushTwoToEmptyThenRemove* ghost schema verify that if we



```
/home/robson/st/SItem.pd
/home/robson/st/Stack.pd
/home/robson/st/StackNode.pd
```

```
Verifying file '/home/robson/st/Stack.pd' ...
Generating verification conditions ... 20 verification conditions generated
Proving verification conditions ... confirmed 20 (100% confirmed, longest 0.1 seconds)
0 seconds
/home/robson/st/Stack.pd (31,4): Information! Confirmed: Class invariant satisfied (defined at /home/robson/st/Stack.pd (16,9)) in context of class Stack [/home/robson/st/Stack
/home/robson/st/Stack.pd (38,4): Information! Confirmed: Class invariant satisfied (defined at /home/robson/st/Stack.pd (16,9)) in context of class Stack [/home/robson/st/Stack
/home/robson/st/Stack.pd (38,9): Information! Confirmed: Operand of 'is' within specified type in context of class Stack [/home/robson/st/Stack.pd (10,1)], proved: self.top withi
/home/robson/st/Stack.pd (39,14): Information! Confirmed: Operand of 'is' within specified type in context of class Stack [/home/robson/st/Stack.pd (10,1)], proved: self.top with
/home/robson/st/Stack.pd (43,8): Information! Confirmed: Class invariant satisfied (defined at /home/robson/st/Stack.pd (16,9)), proved: 0 <= self'.size.
/home/robson/st/Stack.pd (48,10): Information! Confirmed: Property satisfied in context of class Stack [/home/robson/st/Stack.pd (10,1)], proved: ~(self after {it->it'}!push(x)).em
/home/robson/st/Stack.pd (48,11): Information! Confirmed: Class invariant satisfied (defined at /home/robson/st/Stack.pd (16,9)) in context of class Stack [/home/robson/st/Sta
/home/robson/st/Stack.pd (48,26): Information! Confirmed: Precondition of 'push' satisfied (defined at /home/robson/st/Stack.pd (29,4)) in context of class Stack [/home/robson
/home/robson/st/Stack.pd (53,8): Information! Confirmed: Class invariant satisfied (defined at /home/robson/st/Stack.pd (16,9)) in context of class Stack [/home/robson/st/Stac
/home/robson/st/Stack.pd (53,8): Information! Confirmed: Post-assertion valid (defined at /home/robson/st/Stack.pd (54,13)) in context of class Stack [/home/robson/st/Stack.p
/home/robson/st/Stack.pd (53,8): Information! Confirmed: Precondition of 'push' satisfied (defined at /home/robson/st/Stack.pd (29,4)) in context of class Stack [/home/robson/
/home/robson/st/Stack.pd (53,22): Information! Confirmed: Precondition of 'pop' satisfied (defined at /home/robson/st/Stack.pd (36,4)) in context of class Stack [/home/robson
/home/robson/st/Stack.pd (53,22): Information! Confirmed: Intermediate object satisfies class invariant (defined at /home/robson/st/Stack.pd (16,9)) in context of class Stack [/h
/home/robson/st/Stack.pd (59,8): Information! Confirmed: Class invariant satisfied (defined at /home/robson/st/Stack.pd (16,9)) in context of class Stack [/home/robson/st/Stac
/home/robson/st/Stack.pd (59,8): Information! Confirmed: Post-assertion valid (defined at /home/robson/st/Stack.pd (60,13)) in context of class Stack [/home/robson/st/Stack.p
/home/robson/st/Stack.pd (59,8): Information! Confirmed: Precondition of 'push' satisfied (defined at /home/robson/st/Stack.pd (29,4)) in context of class Stack [/home/robson/
/home/robson/st/Stack.pd (59,23): Information! Confirmed: Intermediate object satisfies class invariant (defined at /home/robson/st/Stack.pd (16,9)) in context of class Stack [/h
/home/robson/st/Stack.pd (59,23): Information! Confirmed: Precondition of 'push' satisfied (defined at /home/robson/st/Stack.pd (29,4)) in context of class Stack [/home/robsor
/home/robson/st/Stack.pd (59,38): Information! Confirmed: Precondition of 'pop' satisfied (defined at /home/robson/st/Stack.pd (36,4)) in context of class Stack [/home/robson
/home/robson/st/Stack.pd (59,38): Information! Confirmed: Intermediate object satisfies class invariant (defined at /home/robson/st/Stack.pd (16,9)) in context of class Stack [/h
Generating verification output files ... 0 seconds
0 errors, 0 warnings found.
Compacting memory ... 0 seconds

Verifying file '/home/robson/st/StackNode.pd' ...
Generating verification conditions ... 0 verification conditions generated
```

```
Attempt proofs of correctness for all files    Ready
```

Figure 2.2 Verification conditions of Stack model using PD

push two elements to an empty stack, the first element we pop will be the second one we pushed. For this example, PD has discharged all proof obligations automatically as can be seen in Figure 2.2.

As said before, PD generates automatically several kinds of proof obligations. By using assertions and properties the developer (specifier) can define its own proof obligations. In both cases, PD tries to automatically prove all of them. For the model showed in Figure 2.1 it was generated, for example: checking whether operations preconditions are satisfied, whether class invariants still hold after changes in the state of the model, if user-declared properties are satisfied and so on. All of these were completely proved by PD, without human intervention.

2.2 Model Driven Engineering

Model Driven Engineering (MDE) is a software development philosophy in which models, constructed using concepts related to the domain in question, play a key role in the software development process (Favre, 2004). The idea is on building high-level technology-independent models focused exclusively on the problem domain. These models can be transformed into code for various platforms and technologies.

A domain model is basically a set of related classes used to represent its domain of interest (Martin and Odell, 1995; Fowler, 1996). It is independent of the application logic and its predominant features are the attributes, relationships and constraints over the model.

MDE benefits are directly related to its ability to avoid developers perform repetitive tasks. In most cases such tasks can be automatic transformations (Schmidt, 2006), increasing the productivity of the software development and reducing the time and effort that could then be allocated to other more valuable tasks.

Communication between stakeholders becomes more effective, since models are easier to understand than code and are represented using common modeling languages. So, domain experts can participate more actively and directly on the development process.

A single model can be transformed into code for various platforms. This can be accomplished by creating models of adapters and technology independent connectors and using them into code, so that the different platforms can communicate, promoting portability and interoperability.

Conceptual mistakes can be identified at the model level of abstraction, making it easier to solve these mistakes. Coding tasks can be automated by code generators reducing the introduction of trivial errors.

Unlike other approaches to software development, with mechanisms for automatic transformations, MDE is more productive because it works on a higher level of abstraction. Furthermore, it becomes possible to reuse of knowledge about the domain besides the reuse of software components.

As can be seen in Figure 2.3, there are different approaches (e.g Model Driven Architecture (MDA), Model-Integrated Computing (MIC), Software Factories) and tools (e.g. EMF/GMF, GME, Visual Studio DSL Tools) that follow MDE principles. MDA and MIC are both metamodel-based.

A metamodel is a description (language to describe) of a model (Favre, 2004). In MDE metamodeling is very useful since it provides an abstract notation that separates the abstract syntax and static semantics (static aspects of syntactic fragments, such as types, references, reachability, etc.) of a DSL from its possible concrete syntaxes.

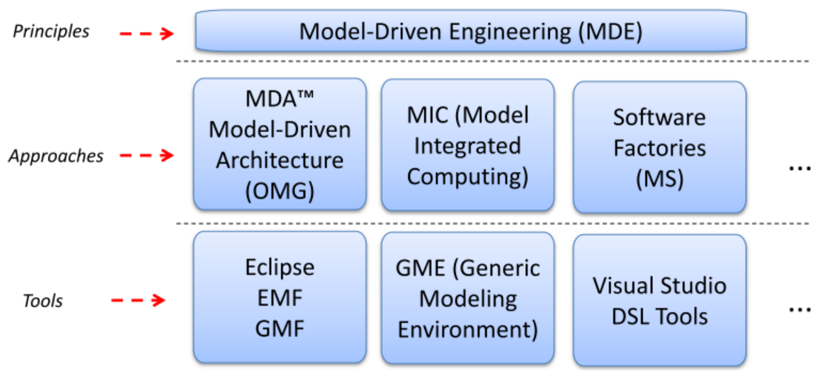

Principles
Model-Driven Engineering (MDE)
Approaches
MDA™ Model-Driven Architecture (OMG)
MIC (Model Integrated Computing)
Software Factories (MS)
...
Tools
Eclipse EMF GMF
GME (Generic Modeling Environment)
Visual Studio DSL Tools
...

Figure 2.3 MDE Approaches and Tools

2.3 Model Driven Architecture

Model Driven Architecture (MDA) is defined by the OMG (Object Management Group) as an approach that describes a basic process, standardized and extensible, to system development based on creation of models as the key step in building applications (Kleppe *et al.*, 2003). Models are not just a way of documenting systems. They are also used as the primary source for analyzing, designing, constructing, deploying and maintaining a system (Truyen, 2006).

Models are used to separate the specification from design details. This raises the level of abstraction of the software development process and consequently increases productivity, portability and interoperability.

It is worth noting that OMG MDA standard (Miller and Mukerji, 2003) is a particular vision of MDE and therefore depends on the use of other OMG standards (Favre, 2004). This is the main difference between MDA and other model-driven approaches. That is, MDA requires technologies that implement the OMG standards. However, it can be considered a subset of MDE.

The MDA approach provides ways for (Miller and Mukerji, 2003): (i) specifying the system regardless of the platform the system will run, (ii) specifying and choosing a particular platform for the system and (iii) specifying model transformations for a particular platform. In the context of MDA, models are classified into three types:

Computation Independent Model (CIM) specifies the system requirements using a computational independent point of view. The CIM, often called domain model or requirements model, describes situations in which the system is used. It hides all information about the use of the system and is fully independent of how the system is implemented.

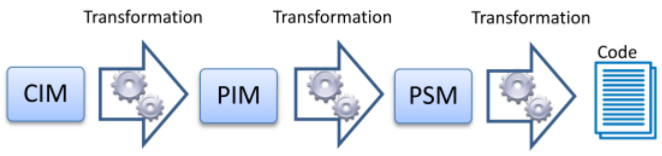


Figure 2.4 Simplified MDA Framework

Platform Independent Model (PIM) is the computation independent model that describes the system hiding details of platform and development technologies. These models present the functional and structural core of the system. Unlike the CIM, PIM design the system in a computational point of view. Metamodels is used for specifying software systems without technical details. Then, it becomes a natural choice for building PIMs.

Platform Specific Model (PSM) is the refinement of the PIM produced by a model-to-model transformation process. It also specifies how that system makes use of a particular platform. Depending on its purpose a PSM may provide enough details to construct a system and to put it into operation.

At first, the process of software development using MDA (Figure 2.4) begins with the definition of the Computation Independent Model (CIM). The CIM can be defined by systems analysts in collaboration with domain experts. Afterwards, the CIM is transformed into a platform-independent model (PIM). The PIM adds information to the CIM, without showing details of the platform being used. Finally, the PIM is transformed into a platform-specific model (PSM) adding details of the target platform. From the PSM the code is generated.

2.4 Eclipse Modeling

The Eclipse Modeling Project is structured into projects that provide several capabilities, such as: abstract syntax development, concrete syntax development, model-to-model transformation, and model-to-text transformation. The term abstract syntax refers to a metamodel and the term concrete sintax to its corresponding form of diagram notation (graphical concrete syntax) or textual notation (textual concrete syntax) (Gronback, 2009).

The Modeling Project provides implementations of industry-standard metamodels and tools for developing models based on those metamodels, many of which conform to published OMG MDA standards. Where standards compliance is not mandatory or when it is necessary a functionality which there is still no implementation, the Modeling project provides alternatives (Gronback, 2009).

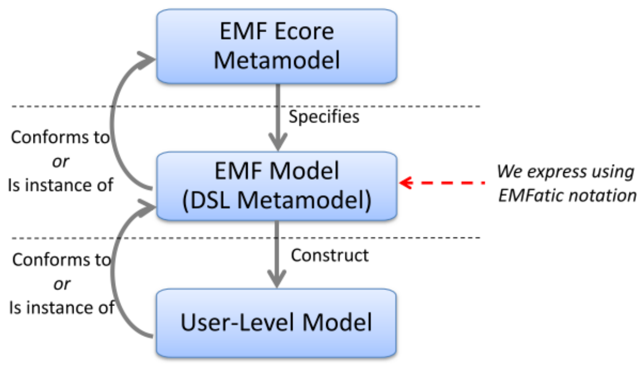

EMF Ecore Metamodel
Specifies
Conforms to or Is instance of
EMF Model (DSL Metamodel)
We express using EMFatic notation
Conforms to or Is instance of
Construct
User-Level Model

Figure 2.5 Relationship between EMF ECore Metamodel (metametamodel), EMF Model (metamodel) and User-Level model (model)

When using Eclipse Modeling Project, the first element of a DSL to be developed is its metamodel, commonly referred to as its abstract syntax. The Eclipse Modeling Framework (EMF) (Steinberg *et al.*, 2008) is used for this task by means of EMF's Ecore models. EMF is a framework used to model a domain and provides an infrastructure able of generating a complete implementation for manipulating basic functionalities of such a modeled domain.

Models provide a language-definition format more robust when compared to traditional approaches such as BNF because a model expressed in terms of Ecore is more expressive since it can have multiple concrete syntaxes to generate textual and graphical editors (Gronback, 2009).

The EMF framework includes a metamodel (EMF's Ecore metamodel), for describing the structure of EMF models that is the basis for constructing user-level models. In Figure 2.5 we see how these concepts are related in their respective levels of abstraction: Zero or more user-level models conforms to an EMF Model (its metamodel). An EMF Model (in our case, a synonym for DSL metamodel) has as its metamodel the EMF's Ecore Metamodel (built-in the EMF Framework). That is, ECore is a metametamodel.

EMF supports several notations to create EMF Models, all of them based on the Ecore metamodel (Steinberg *et al.*, 2008). We use the EMFatic language to define our EMF Models (DSL metamodel).

The Graphical Modeling Framework (GMF) (GMF, 2013) is a model-driven framework that provides the graphical concrete syntax for a DSL and maps to its abstract syntax (metamodel). GMF is widely used to develop graphical Eclipse-based editors for EMF-based DSL languages.

In Figure 2.6 we show the basic usage flow for developing a graphical editor using GMF. The starting point is the definition of a Ecore metamodel that, in our work, is the

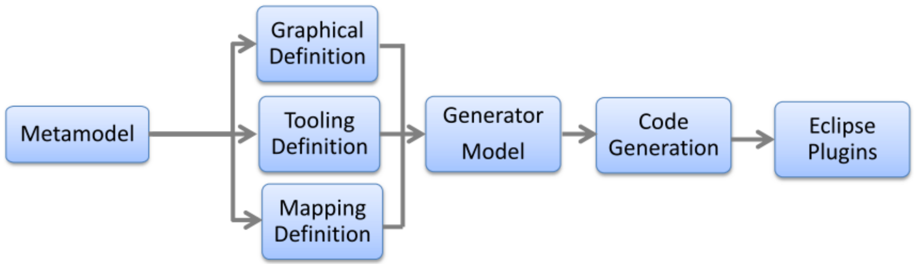


Figure 2.6 An overview of GMF Development Flow (Kolovos *et al.*, 2009a)

corresponding DSL metamodel extracted from a DSL formal specification as will be explained in Section 3. From this metamodel, GMF provides wizards to create additional models related to the graphical concrete syntax: a graphical definition model, a tooling definition model and a mapping model.

The graphical model specifies the shapes that will be used in the editor. The tooling model specifies which tools will be in the editor palette. The mapping model binds the information from the domain model, graphical model and tooling model. The generator model, combines informations of the three models mentioned earlier, is used as input for the code generator.

In terms of MDA, we can consider the Metamodel (Ecore) as Platform Independent Model (PIM) and the Generator Model as Platform Specific Model (PSM), see Figure 2.7. While the Computation Independent Model (CIM) has no correspondents.

Despite its success, as mentioned in (Kolovos *et al.*, 2009a), there are a number of problems related to GMF flow that make implementing an editor atop GMF particularly challenging for potential adopters of Model-Driven Engineering. The built-in wizard provided by GMF generates automatically only basic versions of the tooling, graph and mapping models from the Ecore metamodel itself. For anything beyond simple metamodels it is required that developers manipulate and maintain these complex interconnected models. As a result, implementing a graphical editor with GMF is a hard-working and error prone task even for experienced users.

In order to overcome these shortcomings the EuGENia Tool (Kolovos *et al.*, 2010) was created. EuGENia adopts an approach in which all the additional information that is necessary for implementing a graphical editor is captured by embedding a number of high-level annotations in the Ecore metamodel (Kolovos *et al.*, 2010). From a single metamodel augmented with embedded annotations, EuGENia can automatically produce the required GMF intermediate models (see Figure 2.7) which are necessary in order for GMF to generate a fully-functional graphical editor.

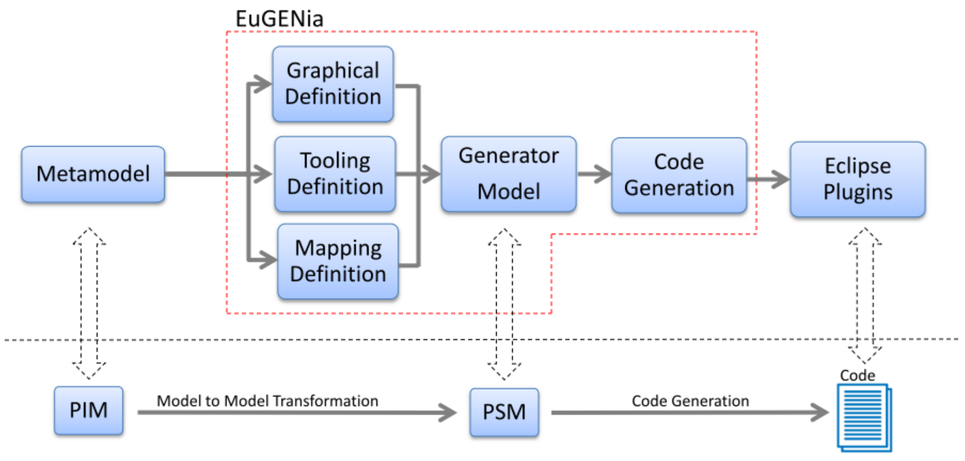


Figure 2.7 GMF Development Flow and its relationship with MDA approach

So, to give a more feature-rich graphical concrete syntax for our DSLs it is needed to provide manually these annotations. This is reasonable given how little can be deduced about the graphical syntax based only in the abstract syntax.

2.5 Constraints

In this work, in addition to GMF usage for the generation of graphical editors from a metamodel, we use the Epsilon platform for management tasks of instances of the metamodel. More specifically, we use the Epsilon Validation Language (EVL) (Kolovos *et al.*, 2009b) to validate the models with respect to its metamodel constraints.

Epsilon is an integrated platform for implementing task-specific languages for interoperable management models (Kolovos, 2008). These languages are used to manage models of different modeling technologies (e.g. EMF, MDR) performing tasks such as model-to-model transformation, code generation, model comparison, refactoring, merging and validation (Kolovos *et al.*, 2013).

The Epsilon Object Language (EOL) (Kolovos *et al.*, 2006) is the core of the platform. Its model navigation and modification facilities are based on OCL. The others task-specific languages of the Epsilon platform extend EOL syntactically and semantically. Through grammar inheritance and reuse of components, the execution mechanisms of task-specific languages only need to define the concepts and logics relevant to the specific domains. As we use EVL, consequently EOL is used too.

EVL has been designed atop the Epsilon platform, and therefore instead of pure OCL, it uses the OCL-based Epsilon Object Language (EOL) as a query and navigation language (Kolovos *et al.*, 2009b).

Although, OCL(-based) languages are commonly used to specify static semantics through invariants defined over the metamodels, it can also be used to specify behavioral aspects through the definition of pre/post-conditions on operations (Gargantini *et al.*, 2009b). EVL goes even further, bringing several improvements in modeling behavioral aspects, such as: support for detailed user feedback, support for warnings/critiques, support for dependent constraints, support for semi-automatically repairing inconsistent model elements and so on (Kolovos *et al.*, 2009b).

However, this does not meet the lack of a static and/or dynamic formal semantics and the capability to formally verify, by means of automatic inference engine and a theorem prover, certain properties about the DSL being specified. That is why instead of using only EVL/EOL we rely on the support of a formal specification language, explained in details in Section 2.1, to be able ensure these essential aspects to our methodology.

2.6 Generating GMF Editors using EuGENia

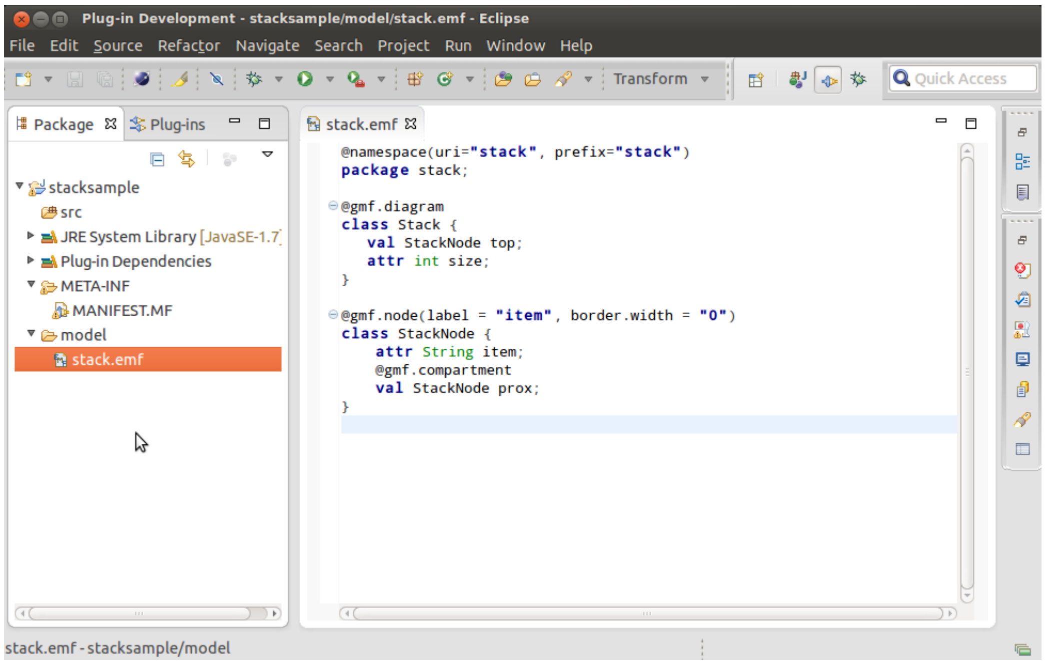
Now, we show how EuGENia can be used to generate a functional GMF editor from a single metamodel. First of all, we create a GMF Project and then define a metamodel, as shown in Figure 2.8. In this case, we define a metamodel for a stack data structure.

This simple metamodel is composed of two classes: *Stack* and *StackNode*. The first represents a stack data structure itself containing a reference to the top element and an integer that represents the size of the stack. The second class represents a stack node, which is responsible for store some data (in this case, a string) and reference the next element of the stack.

As can be observed, this metamodel is somehow related with the stack formal specification presented in Section 2.1. However, we will procrastinate the explanation of this relationship for the Section 3.3.

As mentioned in Section 2.4, EuGENia simplifies the GMF flow. It accomplishes this by automatically generating GMF intermediate models and also considering the use of annotations over the metamodel. These annotations (*@gmf.**, see Figure 2.8) aims at giving a more feature-rich graphical concrete syntax. The complete set of EuGENia annotations can be found in (EuGENia, 2013).

In Figure 2.8, the annotation *@gmf.diagram* in the class *Stack* denotes that it is the root object of the metamodel. The annotation *@gmf.node* in the class *StackNode* indicates that it should appear on the diagram as a node. The parameter *label="item"* indicates that the


Plug-in Development - stacksample/model/stack.emf - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Transform
Quick Access
Package
Plug-ins
stacksample
src
JRE System Library [JavaSE-1.7]
Plug-in Dependencies
META-INF
MANIFEST.MF
model
stack.emf
stack.emf - stacksample/model

```
@namespace(uri="stack", prefix="stack")
package stack;

@gmf.diagram
class Stack {
   val StackNode top;
   attr int size;
}

@gmf.node(label = "item", border.width = "0")
class StackNode {
    attr String item;
    @gmf.compartment
    val StackNode prox;
}
```

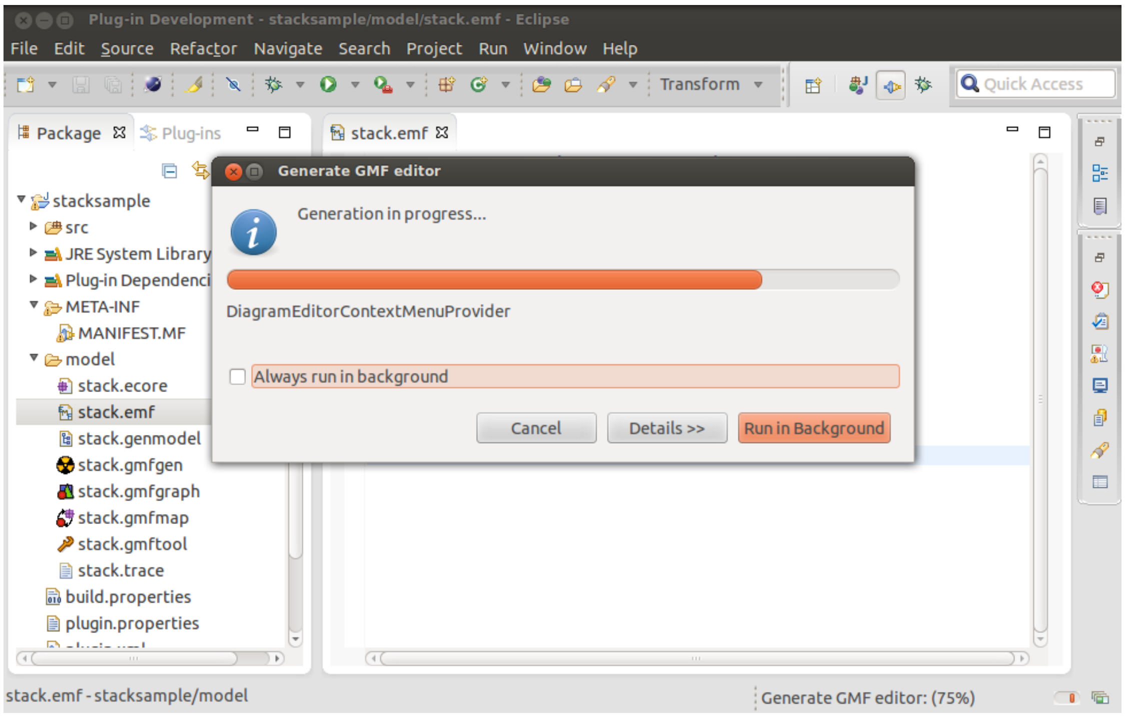
Figure 2.8 Stack Metamodel

value of the *item* should appear as the label of the node and the parameter *border.width="0"* sets the border size of the node to zero. The annotation *@gmf.compartment* defines that the containment reference will create a compartment where model elements that conform to the type of the reference (in this case, *StackNode*) can be placed.

Right-clicking over metamodel file (*stack.emf*) and selecting the option *EuGENia* $\rightarrow$ *Generate GMF editor*, EuGENia tool automatically generates (see Figure 2.9) the graphical (*.gmfgraph*), tooling (*.gmftool*) and mapping (*.gmfmap*) models, needed to implement a GMF editor. Additionally, it triggers by means of EMF/GMF all other tasks related to generation of the editor. We show in Figure 2.10, using a built-in user-friendly viewer, the graphical, tooling and mapping models. In fact, they are XML-based files.

In Figure 2.11 we see the stack GMF editor working. As can be seen, it runs as an Eclipse Application. That is, the GMF editor runs by launching a separate instance of the Eclipse application (*Run As* $\rightarrow$ *Eclipse Application* option). Additionally, we can generate a RCP (Rich-Client Platform) product that allows to generate a stand-alone Eclipse-based application for this GMF editior. This should be explicit in the *rcp* parameter of the *gmf.diagram* annotation: *@gmf.diagram(...,rcp = "true",...).*

As can be evidenced by the annotations that we added to the metamodel, we provided very basic information in order to enhance the graphical concrete syntax of the stack metamodel. Therefore, the resulting editor has a simple graphical interface. In the



Generation in progress...

DiagramEditorContextMenuProvider

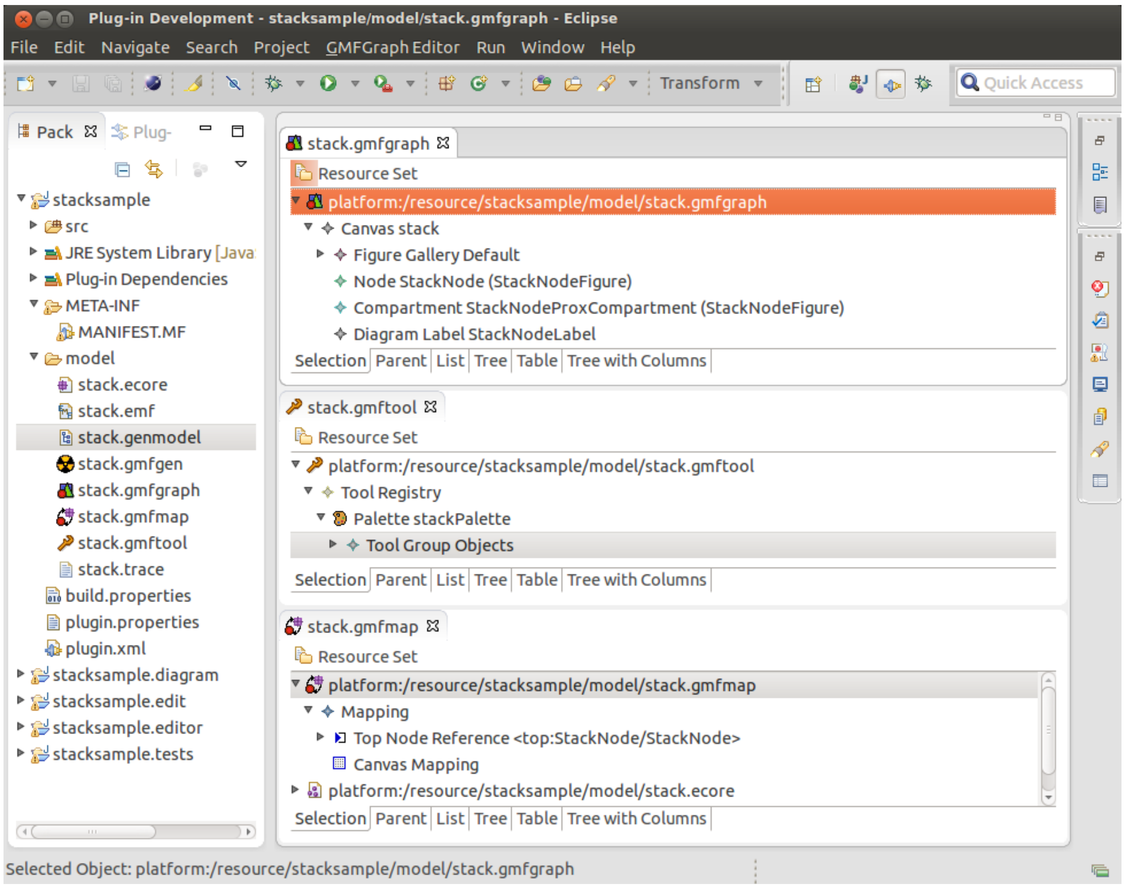
Always run in background

stack.emf - stacksample/model

Generate GMF editor: (75%)

Figure 2.9 Genetating the GMF editor for the stack metamodel

right-hand side of Figure 2.11, we see a basic usage of the editor. First, we added the *"First Item"* node, followed by the *"Second Item"* node. Note that the stack is constructed in an inverted visual order.


Plug-in Development - stacksample/model/stack.gmfgraph - Eclipse

File Edit Navigate Search Project GMFGraph Editor Run Window Help
Transform
Quick Access
Pack
Plug-

```
stacksample
  src
  JRE System Library [Java
  Plug-in Dependencies
  META-INF
    MANIFEST.MF
  model
    stack.ecore
    stack.emf
    stack.genmodel
    stack.gmfgen
    stack.gmfgraph
    stack.gmfmap
    stack.gmftool
    stack.trace
  build.properties
  plugin.properties
  plugin.xml
stacksample.diagram
stacksample.edit
stacksample.editor
stacksample.tests
```

stack.gmfgraph

```
Resource Set
platform:/resource/stacksample/model/stack.gmfgraph
  Canvas stack
    Figure Gallery Default
    Node StackNode (StackNodeFigure)
    Compartment StackNodeProxCompartment (StackNodeFigure)
    Diagram Label StackNodeLabel
```

stack.gmftool

```
Resource Set
platform:/resource/stacksample/model/stack.gmftool
  Tool Registry
    Palette stackPalette
      Tool Group Objects
```

stack.gmfmap

```
Resource Set
platform:/resource/stacksample/model/stack.gmfmap
  Mapping
    Top Node Reference <top:StackNode/StackNode>
    Canvas Mapping
platform:/resource/stacksample/model/stack.ecore
```

Selected Object: platform:/resource/stacksample/model/stack.gmfgraph

Figure 2.10 Graphical (*.gmfgraph*), Tooling (*.gmftool*) and Mapping (*.gmfmap*) models for the stack metamodel

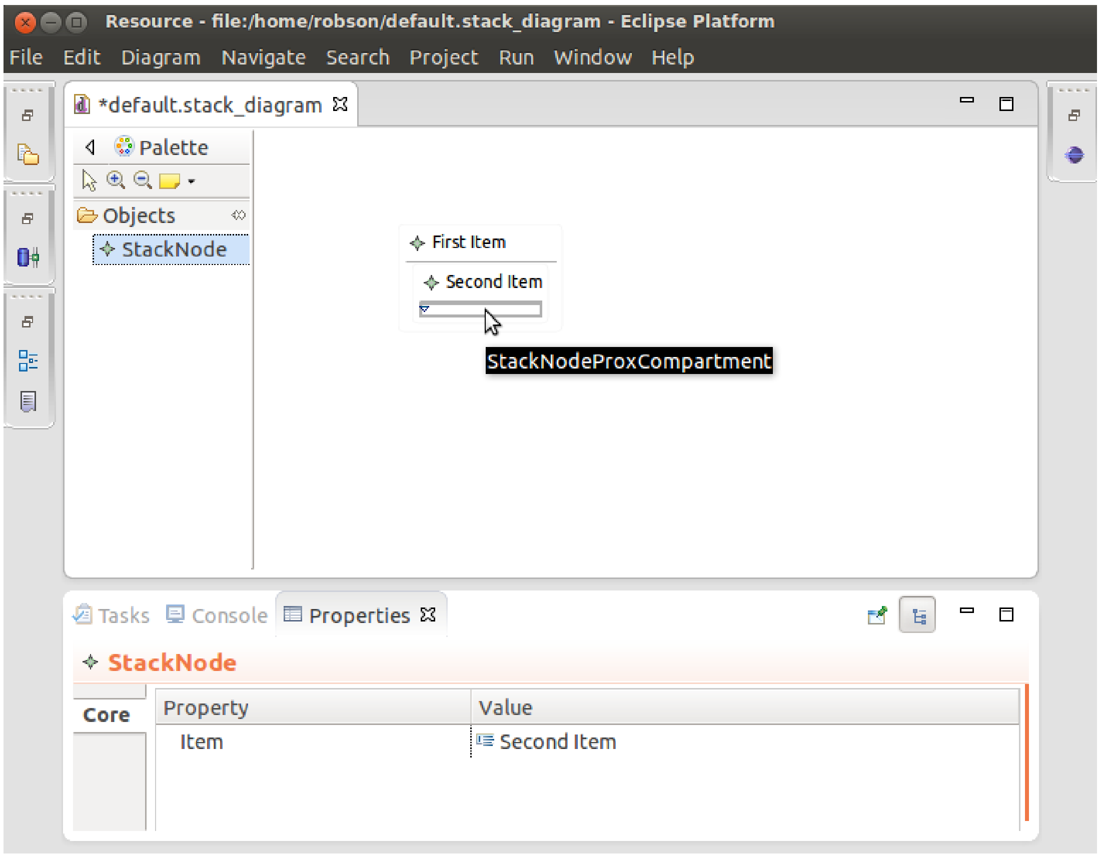


Figure 2.11 Stack GMF Editor

3
Proposed Methodology

In this chapter, we show our proposed methodolody. In Section 3.1, we present the steps of the methodolody that we are proposing. Section 3.2 shows an instantiation of our methodolody and in Section 3.3 we introduce the application of our methodolody.

3.1 Methodology Overview

To better understand our methodology we revisit some key concepts used in this work: A *model*, or *domain model*, is an abstraction that defines and relates a set of concepts within a certain domain. A *metamodel* is also another abstraction, but emphasizing the properties of the model itself. A model is a metametamodel if it is used to define metamodels.

In other words, a metamodel is a model that provides the basis for constructing another model (For instance, the UML metamodel defines the structure that all UML models must have). When a model is expressed in terms of its metamodel, we say that it conforms to or is an instance of its metamodel. Then, models can be categorized into different levels of abstraction.

In Figure 3.1 we see the general idea of the methodology. The starting point (Step A) is to create a DSL formal specification (L) that is composed of a syntax and static semantics (SS_L), and dynamic semantics (DS_L) parts. Therefore, L captures all aspects of the DSL using formal methods. This allows us to formally check desired properties.

The DSL syntax and static semantics (SS_L) are used as the input of an automatic strategy that generates a corresponding metamodel MM and a set of constraints SC_{MM} over MM (Step B). The artifacts MM and SC_{MM} are the primary artifacts to generate a user-friendly constraint preserving front-end for the DSL.

A GUI builder is applied on this metamodel (Figure 3.1) to create the respective DSL GUI editor (Step C). This GUI editor is responsible to manipulate instances of the

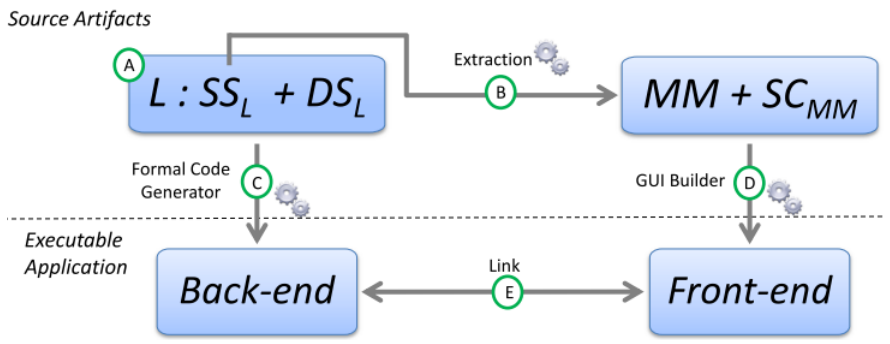


Figure 3.1 Methodology High-level View

DSL in its graphical representation. From the complete DSL formal specification, we propose using a verifiable formal code generator to create a back-end for this DSL (Step D). Finally, we systematically integrate front and back-end (Step E). Therefore, following our proposed methodology one has only to focus on the formal specification.

Before applying the previous steps, we need to choose the right techniques and tools to provide the appropriate technological support. Such choices must follow the requirements:

$Req-1$ The formalism chosen must have tool support able to prove properties and synthesize code (back-end) from a formal specification. Optionally, it can support refinement.

$Req-2$ Automatically extract a metamodel and a set of constrains over this metamodel from a formal specification.

$Req-3$ The target metamodel notation should be supported by modeling tools able to generate graphical editors (front-end) that manipulate instances of this metamodel.

$Req-4$ The constraint language should be able to specify and evaluate constraints on models of the chosen metamodel notation.

$Req-5$ The link between back-end and front-end should be designed in a way that it is transparent for the user of the final application.

□

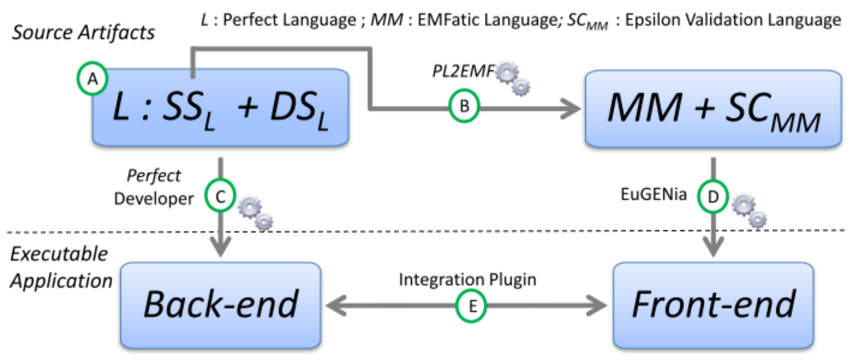


Figure 3.2 Proposed Methodology Instantiated

3.2 Meeting the requirements

In the way that our methodology (Figure 3.1) was designed, it is sufficiently robust to allow the use of different techniques and technologies to meet the established requirements. In this work, we show one of many possible technological instantiations of it as can be seen in Figure 3.2. Now we show the requirements and what has been chosen to meet each one, in this work:

$Req-1:$ to specify a DSL formal specification, prove properties including refinement between versions of the specifications and back-end code generation, we use the *Perfect* Language and the *Perfect* Developer Tool (Escher, 2012).

$Req-2:$ to extract the metamodel itself and its associated constraints, we developed an extractor as an Eclipse Plugin Tool, called *PL2EMF*, using the Spoofax Language Workbench (Spoofax, 2013). The design of the extractor is described in Section 4.1 and its implementation in Section 5.

$Req-3:$ to describe the DSL metamodel we use the Eclipse Modeling Framework by means of EMFatic language to define our EMF Models (DSL metamodel).

$Req-4:$ we use EVL language to specify and evaluate constraints on models of our DSL metamodels (EMF Models).

$Req-5:$ as we chose to use Eclipse Modeling Environment, we establish the link between back-end and front-end by creating an integration plugin, presented in Section 4.2.

```
class Stack ^=
abstract
  var top : StackNode || void;
  var size : int;
  invariant size >= 0;
interface
  function empty: bool ^= ...
  function validate (itm: SItem): bool ^= ...

  schema !push(newItem: SItem) ...
  schema !pop(x!: out SItem)...

  build {}...

  property (x: SItem)...
```

Figure 3.3 Fragment of the Stack Formal Specification

3.3 Methodology Application Overview

To show an overview of how our methodology works, we revisit the example of the Stack Formal Specification (Figure 2.1), presented in Section 2.1. This specification, which we will call L from now on, is the starting point of our methodology. As in L there is no need to refine the semantics into more concrete descriptions, from this level of abstraction we can apply a formal code generator to obtain the executable back-end.

It is worth noting that in this example we present an overview of the steps related to the creation of the *Source Artifacts* (Step A and Step B). In our case study, Chapter 6, we perform all the steps of the methodology.

Taking a closer look in a fragment of L, for instance the class *Stack* as shown in Figure 3.3, we highlight the aspects of this class that captures its static syntax and static semantics (SS_L), using dotted lines, and dynamic semantics (DS_L), using continuous line.

The SS_L aspects that we consider in the extraction process (see Section 4.1 for more details) are: (i) structural features (e.g. classes and their relationships); and (ii) state features (e.g. the abstract variables and invariants). The other *Perfect* elements are not considered because they are related to the business rules.

A metamodel MM and a set of constraints SC_{MM} over M, as detailed in Section 4.1, is extracted from L by using a systematic extraction process that we have established. This process considers as input the syntactical and static semantics SS_L aspects of L (Stack Formal Specification). For our example, the output of this process is shown in Figure 3.4:

```
@namespace(uri="stack",
        prefix="stack")
package stack;

@gmf.diagram
class Stack {
  val StackNode top;
  attr int size;
}

@gmf.node(label = "item",
        border.width = "0")
class StackNode {
  attr String item;
  @gmf.compartment
  val StackNode prox;
}
```

(a) Stack Metamodel

```
context Stack {
  constraint inv1 {
   check: self.size>0
   ...
  }
}
```

(b) Metamodel constraints

Figure 3.4 The complete Stack Metamodel (augmented with embedded EuGENia annotations) and its constraints

a metamodel expressed using EMFatic (3.4(a)) and the constraints using EVL (3.4(b)).

Recall from Section 2.4 (Figure 2.5) where we presented the relationship between EMF ECore Metamodel, EMF Model and User-Level model. Now we revisit this relationship but at the same time we see, on the right-hand side of Figure 3.5, how these concepts apply to our example: several stacks (user-level model) as possible instantiations of the Stack EMF model that conforms to EMF Ecore Metamodel. Additionally, we see the stack constraints acting to restrict the user-level models.

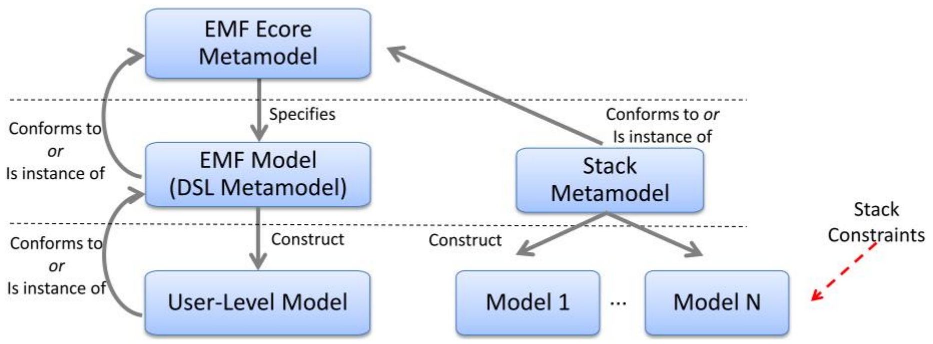

EMF Ecore Metamodel
Specifies
Conforms to or Is instance of
EMF Model (DSL Metamodel)
Construct
Conforms to or Is instance of
User-Level Model
Conforms to or Is instance of
Stack Metamodel
Construct
Model 1
...
Model N
Stack Constraints

Figure 3.5 Abstraction levels between models

4 Methodology Steps

In this chapter, we detail some steps of our methodology by instantiating them. In Section 4.1, we detail how the extraction of metamodel and constraints from a formal specification is performed. In Section 4.2 the link strategy between front and back-end is showed.

4.1 Metamodel Extractor

Recall from Section 3.2 as we met the requirements presented in Section 3.1. Therefore, the aim of the Metamodel Extractor is to obtain a metamodel, expressed in EMFatic, and constraints, in EVL, from a formal specification written in the *Perfect* Language. The extraction is performed by a set of translation rules based on a subset of *P*L grammar.

In this section we show how we designed the *PL2EMF* tool (in Chapter 5, we detail the *PL2EMF* implementation). We present a fragment of the subset of the *P*L grammar used in this work (Section 4.1.1) and some translation rules (Section 4.1.2) using a platform independent notation. In Section 6.2, of our case study (Chapter 6), we exercise these translation rules over a fragment of a *P*L specification.

This grammar and the translation rules are the primary artifacts needed by the Spoofax Language Workbench (Spoofax, 2013)) to generate automatically the PL2EMF as an Eclipse Plugin tool.

4.1.1 Grammar

The features of the *Perfect* specification, related to syntax and static semantics (SS_L), that we consider in our translation are: (i) structural features, that is, classes and their relationships and (ii) state features, that is, the abstract variables and invariants of each

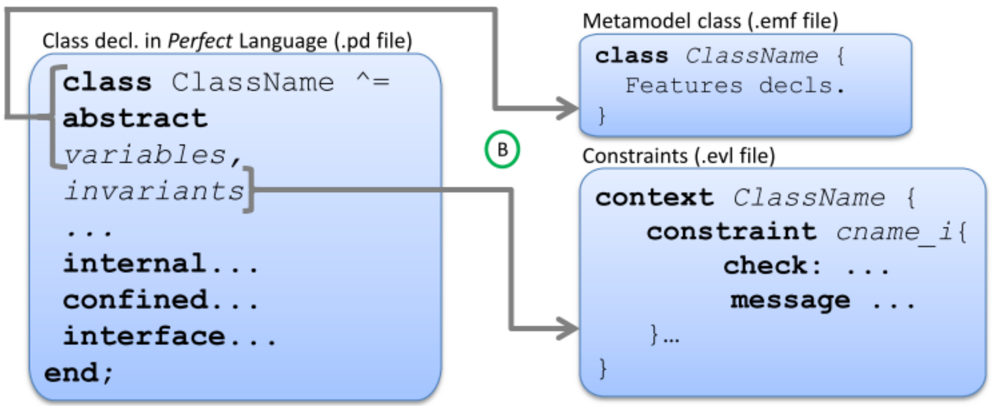


Figure 4.1 Detailed metamodel extraction

class (located in the *abstract* section).

The other *Perfect* constructs are not considered in the extraction because they are related to the dynamic semantics (DS_L) of the specification, like functions and schemas, instead of the metamodel and its constraints. Furthermore, verification conditions (or proof obligations) modeled by the user or those automatically generated by the *Perfect* compiler, such as assertions, properties and operations contracts (pre and post-conditions) are not considered in the extraction process because current MDE environments are not able to prove these kind of conditions. These ignored elements are previously removed by a preprocessing phase.

The translation strategy is depicted in Figure 4.1. Classes in *PL* become classes of the metamodel and their variables, in the ***abstract*** section, become class features. Additionally, the set of class invariants are translated to a set of constraints. The remaining parts of the *Perfect* specification are ignored. Therefore, we consider an abstract grammar for *PL* (see Figure 4.2) tailored to metamodeling needs. The *PL* complete grammar can be found in (Escher, 2012).

According to the grammar in Figure 4.2, a *PL* class is defined with a modifier (*ClassModifier*), a class declaration name (*ClassDeclName*) and an element that marks the beginning ('^=') of the class body (*ClassBody*). A class body can optionally contain an inheritance declaration (***inherits*** *IDENTIFIER*) followed by the keyword ***abstract*** and a list of variables and invariant declarations.

Each variable declaration follows the format ***var*** *IDENTIFIER ':' TypeExp*. A list of invariants can be defined after the keyword ***invariant***. Types can be predefined types (*PredefType*), user defined types (*ClassNameAsType*) or the combination of both (*TypeExpr*).

ClassDeclaration → *ClassModifier ClassDeclName* '^=' *ClassBody* ***end*** ;
ClassModifier → ***deferred*** | ***final*** | *ε*
ClassDeclName → ***class*** *IDENTIFIER*
ClassBody → *AbstractDeclarations* | ***inherits*** *IDENTIFIER AbstractDeclarations*
AbstractDeclarations → ***abstract*** *AbstractMemberDeclarations* ';' | ***abstract***
AbstractMemberDeclarations → *AbstractMemberDeclaration*
| *AbstractMemberDeclarations* ';' *AbstractMemberDeclaration*
AbstractMemberDeclaration → *AbstractVariableDeclaration* | *ClassInvariant*
AbstractVariableDeclaration → ***var*** *IDENTIFIER* ':' *TypeExpr*
ClassInvariant → ***invariant*** *ListOfPredicates*
PredefType → ***bool*** | ***byte*** | ***char*** | ***int*** | ***real***
ClassNameAsType → *IDENTIFIER* | ***from*** *IDENTIFIER*
Type → *PredefType* | *ClassNameAsType*
TypeExpr → *Type* | ***set of*** *Type* | ***seq of*** *Type* | ***bag of*** *Type* | ...
ListOfPredicates → *ExpressionList*
ExpressionList → *Expression* | *ExpressionList* ',' *Expression*
Expression → ... | ***forall*** *IDENTIFIER* '::' *CollExpr* ':-' *Expression* |
exists *IDENTIFIER* '::' *CollExpr* ':-' *Expression*

Figure 4.2 A fragment of the subset of the *P*L grammar used in this work.

4.1.2 Translation Rules

The rules were divided in four groups: *Class declaration*, *Abstract Declarations*, *Corresponding Types* and *Invariants*. For the sake of conciseness, we show one or two rule(s) of each group. The hole set of rules is found in (Silva, 2013).

The translation rules take as parameter (left-hand side of the symbol ▶) one class of a *P*L specification and produces its correspondent metamodel class (right-hand side). The rules are inductively defined on the structure of the syntax given in Figure 4.2 and follow a top-down strategy. All the rules follows the generic structure:

Rule *Number* . $[\ pattern\ ]^{RuleName}$ ▶ *replacement*
proviso *condition*

Number is a numerical identification of the rule. The expression on the left-hand side of the arrow is referred to as the *pattern* (*P*L code input), and the expression on the right-hand side is referred to as the *replacement* (EMFatic/EVL code output). A translation rule is said to be a conditional translation rule whether *condition* is defined, otherwise is said to be an unconditional translation rule. When the rule holds, that is, the *pattern* is found in the input and the *condition* is valid, it is replaced by the expression *replacement*.

Rule 1 is the starting point of the translation and it triggers all the other rules.

Class Declaration The first rule (Rule 1) takes as argument a complete class declaration, whose main elements are its modifier (*CModif*), its name (*CDeclName*), and its body (*CBody*).

Rule 1 . [*CModif CDeclName '^=' CBody* ***end*** *;* $]^{classDecl}$ ▶
[*CModif* $]^{classModif}$ [*CDeclName* $]^{classDeclName}$ [*CBody* $]^{classBody}$ }

Rule 1 is defined in terms of three other rules, where the rule associated to *CModif* (named *classModif*) is not presented here. Rule 3 deals with *CDeclName* producing `class IDENTIFIER` as output. Rules 4 and 5 deals with *CBody*.

Rule 3 . [***class*** *IDENTIFIER* $]^{classDeclName}$ ▶ `class IDENTIFIER`

Rule 4 deals with a non-inherited class and Rule 5 deals with an inherited class producing `extends IDENTIFIER{` as output. Both rules delegate the processing of the abstract declarations (abstract variables and invariants) to Rule 6 or 7.

Rule 4 . [*AbsDecls* $]^{classBody}$ ▶ `{` [*AbsDecls* $]^{abstractDecls}$

Rule 5 . [***inherits*** *IDENTIFIER AbsDecls* $]^{classBody}$ ▶
`extends IDENTIFIER {` [*AbsDecls* $]^{abstractDecls}$

Abstract declarations Rule 6 manages the case when the ***abstract*** section contains a non empty list of variables and invariants declarations, delegating the declarations of abstract members to Rule 8. Otherwise, Rule 7 (not shown) produces an empty output.

Rule 6 . [***abstract*** *AbsMemberDecls* $]^{abstractDecls}$ ▶
[*AbsMemberDecls* $]^{listAbsMemberDecls}$

Rule 8 (not shown) simply iterates over the abstract declarations of *P*L code, delegating to Rules 9 and 10 the remainder translation.

Rule 9 maps a variable declaration in *P*L to a class feature in EMFatic that follows the pattern: *modifiers featureKind emfType id*. For us, we can ignore *modifiers*. The *featureKind* is the kind of the class feature, that can be *attr* (an EAttribute), *val* (an EReference with containment = true) and *ref* (an EReference with containment = false). The *emfType* is the type of the feature and *id* is the identifier.

Rule 9 . [***var*** *IDENTIFIER ':' type* $]^{absMemberDecl}$ ▶ [*type* $]^{emfFeatureKind}$
[*type* $]^{emfEquivType}$ `IDENTIFIER;`

Rule 10 delegates to Rule 14 the processing of the invariants declaration. Rule 14 is the responsible to start the generation of the EVL code, which define the constraints over instances of the resulting metamodel. The *clName* is the name of the class (in *P*L) taken as source of the translation rules.

Rule 10 . $[\ \mathit{absMbDecl}\ ]^{absMemberDecl}$ ► $[\ \mathit{clName\ ListInvs}\ ]^{classConstraints}$
proviso *absMbDecl* = ***invariant*** *ListInvs* $\wedge$ *ListInvs*$\neq \langle\rangle \wedge$ *clName* $\neq \langle\rangle$

Corresponding Types A subset of types in *P*L has equivalents in EMFatic. When the variable in *P*L is of a predefined basic type, Rule 11.1 produces as output the keyword `attr`, as the kind of the correspondent class feature.

Rule 11.1 . $[\ \mathit{type}\ ]^{emfFeatureKind}$ ► `attr`
proviso *type* $\in$ ***{ bool, byte, char, int, real, string }***

To non predefined basic types, such as ***set/seq/bag of*** *type*, user defined class as type and so on, the related rules (not shown) produce as output `ref` or `val` depending on the type.

Rule 12 and Rules 12.x (1≤x≤ 6) are responsible for the translation of basic predefined types in *P*L to their correspondent types in EMFatic. As Rules 12.x are very straightforward we just show Rule 12.1 as an example.

Rule 12. $[\mathit{type}]^{emfEquivType}$ ► $[\mathit{type}]^{predefType}$
proviso *type* $\in$ ***{bool, byte, char, int, real, string}***

Rule 12.1. $[\boldsymbol{bool}]^{predefType}$ ► `boolean`

When we are dealing with user defined classes as types, Rules 13.x produce the correspondent type in EMFatic code. For example, Rule 13.1 applies when a class identifier, preceded or not by the keyword ***from***, is used as type, returning only the class name. In *P*L, the additional keyword ***from*** is necessary to state that a variable of a super class type can receive an instance of an inherited class. In EMFatic this is not necessary.

Rule 13.1 . $[\ \mathit{classID}\ ]^{emfEquivType}$ ► `classID`
proviso *classID is an IDENTIFIER or* ***from*** *IDENTIFIER*

Invariants Rule 14 generates the EVL code that defines the context in which the list of invariants *ListInvs* applies to. This rule creates a context named `clName` (the name of the class to which list of invariants belongs). Its body (invariants) is translated by Rule 15.

Rule 14 . $[\ \mathit{clName\ ListInvs}\ ]^{classConstraints}$ ▶
`context clName {` $[\ \mathit{ListInvs}\ ]^{eqListEVLConstraints}$ `}`

Rule 15 is responsible for iterating over the list of invariants, generating a constraint section named *`cname_i`* to the invariant *inv* and delegating to Rules 16.x the translation of the expression that define the invariant *inv* into equivalent expressions in EVL. The *`cname_i`* is automatically generated, however, this constraint name can be edited by the user to become more meaningful.

Rule 15 . $[\ \mathit{listinv}\ \mathit{frown}\ \langle \mathit{inv} \rangle\ ]^{eqListEVLConstraints}$ ▶
`constraint` *`cname_i`* `{` $[\ \mathit{inv}\ ]^{eqEVLConstr}$ `}`
$[\ \mathit{listinv}\ ]^{eqListEVLConstraints}$

Rule 16.1 takes a universal quantified invariant over a collection of items (in *P*L) and translates it into a correspondent in the EVL notation. In this rule, *x* is an *IDENTIFIER*, *expr* is an *Expression* (a boolean expression involving x) and *collExpr* is an expression that defines a data collection.

Rule 16.1. $[\textbf{\textit{forall}}\ \mathit{x::collExpr\ :\text{-}\ expr}]^{eqEVLConstr}$ ▶
`check:` $[\mathit{collExpr}]^{eqCollInEVL}$`.forAll(x |` $[\mathit{expr}]^{eqExprInEVL}$`)`

The rule for the existential quantifier is very similar to the rule for universal quantifier. Instead of using the forAll() function, we use the exists() function.

4.2 Link Strategy

Recall from Section 2.1 that *Perfect* Developer (PD) is suitable to create a stand-alone system (back-end) but it does not provide mechanisms to easily construct GUI-based systems.

We integrate PD (Section 2.1) with GMF (Section 2.4) to easily create a GUI-based PD system. This integration is accomplished via a plugin that makes the communication between front (GMF) and back-end (PD) transparent to the end user.

This plugin has the following responsibilities: (i) interchange data formats between front and back-end, (ii) receive requests from the GUI to the back-end, (iii) catch back-end

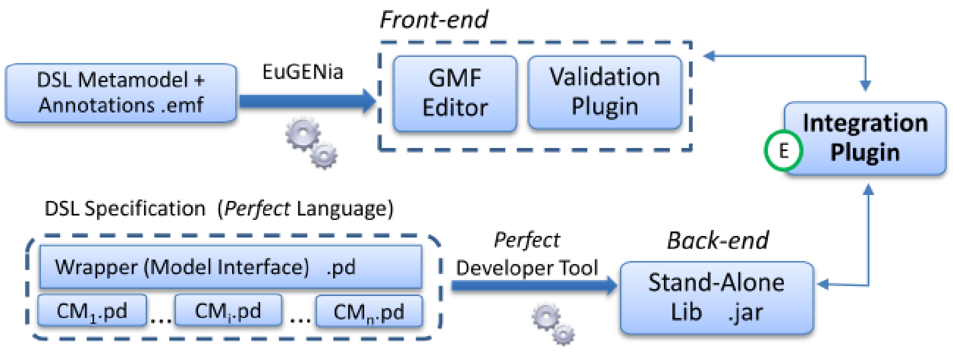

Front-end
DSL Metamodel + Annotations .emf
EuGENia
GMF Editor
Validation Plugin
Integration Plugin
E
DSL Specification (Perfect Language)
Wrapper (Model Interface) .pd
CM1.pd ... CMi.pd ... CMn.pd
Perfect Developer Tool
Back-end
Stand-Alone Lib .jar

Figure 4.3 Detailed Link Strategy

responses and deliver to the GUI, depending on the results the GUI elements can be changed.

The front-end is the part of a system that interacts directly with the user. That is, an interface between the user and the back-end which has the function of processing the user input and adapt it for the back-end. We concentrate all the business-like functionalities in the back-end.

This plugin uses the Epsilon Wizard Language (Kolovos *et al.*, 2013), an extension of EOL. From within GMF editors we can execute Epsilon Wizard Language (EWL) wizards to access underlying elements of models (instances of a metamodel), shown graphically in the GUI, and identify the user's graphical requests.

We see each wizard as an operation that the user can request for the GUI. This request triggers EOL operations that read the part of the model related to the request, manipulating it to deliver the correct data type for the back-end correspondent operation. When the back-end returns the results, these manipulations occur in the reverse order.

Figure 4.3 shows our integration strategy with more detail. From a *Perfect* Specification, we generate, using PD, the back-end as a stand-alone library. Our experience has shown that that the back-end as a library is more appropriate for our purposes. Then, the integration plugin can view this library as one of its own. This makes the interaction easy, since from PD we can generate the back-end using the same target programming language with which the integration plugin has been built.

Some operations of the back-end need to have their preconditions satisfied to be performed correctly. When we call operations with preconditions, outside the PD environment, it is not able to handle violations. Because of this, we need to define a wrapper for these operations. They are called from external applications. They handle any problem that can occur resulting from an incorrect external call. The main goals of the wrapper, is to provide a common access point for all operations provided by a DSL specification and acting as a firewall between the *Perfect* back-end and the environment .

In Section 6.4, we show how the link strategy works in our case study. It is also worth noting that front-end and integration plugin runs on Eclipse platform.

5
PL2EMF Development

In this chapter, we show how we developed the *PL2EMF* tool. In Section 5.1 we present the development environment (Spoofax Language Workbench) that we used to develop the *PL2EMF* tool. In Section 5.2 and Section 5.3 we present the source artifacts that we created to generate our tool by means of the Spoofax Language Workbench. Section 5.4 shows the tool being used.

There are several other Language Workbench, such as XText (Eysholdt and Behrens, 2010), JetBrains MPS (Voelter and Solomatov, 2010). However, we chose use Spoofax due to its support for Stratego Language. Stratego is a language for program transformation. It demonstrated to be very productive to define our translation rules and its facility for manipulating Abstract Syntax Tree (AST).

5.1 Spoofax Language Workbench

Spoofax is a language workbench for developing textual DSLs with full-featured Eclipse editor plugins (Spoofax, 2013). It integrates language processing techniques for parser generation, meta-programming, and IDE development into a single environment (Kats and Visser, 2010).

Using the Spoofax language workbench, a DSL grammar can be written, in a declarative and modular way, using the Syntax Definition Formalism (SDF). Using this single fomalism the complete syntax (lexical and context-free) of a language can be defined and integrated (Heering *et al.*, 1989).

Based on a grammar expressed in SDF grammar, the Spoofax language workbench automatically provides basic editor facilities such as syntax highlighting and code folding, which can be customized using high-level descriptor languages. Using its parser generator tool, a parser can be created from this grammar. This generated parser can be used in an interactive environment, supporting error recovery in case of incorrect syntax or incomplete programs (Kats and Visser, 2010).

We express the semantic definitions, using Stratego language, by means of rewrite rules that provide an integrated solution for analysis, program transformation, code generation rules and more sophisticated editor services such as error marking, reference resolving, and content completion (Kats and Visser, 2010).

Defining transfomations rules by using the Stratego Language, the Spoofax Workbench provides the automatic generation of a functional program transformation infrastructure that is able to perform the transformations we defined.

All of these generated services and infrastructure are integrated with Eclipse. This also allows the application to be delivered to "end developers" via stand-alone plugin.

5.2 Syntax

In Figure 5.1 we show the *P*L grammar we consider in this work (recall from Section 4.1.1) in terms of the SDF language. Aditionally, Figure 5.2 shows the lexical syntax.

We declare the main module of our grammar using SDF in Figure 5.1. In line `01`, we name it as `PerfLang`. A module can import other modules. In this case, we import the `Common` module, (see line 02).

A module can contain a number of sections. The `exports` section in line `03`, is used to define the syntactic aspects (visible to other modules that import it). In this case, we have: start symbols (line `04`) and context-free syntax (line `06`).

All syntax, both lexical and context-free, is defined by productions, respectively in `lexical syntax` section (declared constructs like literals, identifiers and so on) and `context-free syntax` section (constructs like operators, statements and so on). In the module `PerfLang` (Figure 5.1) we declare the context-free syntax (starting in line `06`) and in the module `Common` (Figure 5.2) we declare the lexical syntax (starting in line `03`).

Productions, in SDF, have the form $a_1 ... a_n -> a_0$, where $a_1 ... a_n$ is a sequence of strings such that when they match, they produce the symbol a_0. That is, productions take a list of symbols and produce another symbol. The terminology of terminal and non-terminal is not very suitable for SDF, since only single characters are terminals and almost everything else is a non-terminal. For this reason, every element of a production is called a symbol (SDF, 2013).

SDF includes a declarative disambiguation construct to uniquely identify a symbol in the abstract syntax: the `cons(n)` annotation, where `n` uniquely identifies a symbol Kats *et al.* (2010). It also provides several regular expression operators to simplify common patterns that appear in defining productions in order to reduce the effort and enchance expressivity in defining grammars. In line `16` of Figure 5.1 the declaration `ClDeclaration*` means zero or more symbols `ClDeclaration`. That is, it is allowed to declare zero or more *P*L classes in one file. The declaration `{Predicate ","}+`, line `30`, means one or more symbols `Predicate` separated by `","`.

In Figure 5.2 we show the `Common` module. This module focuses on the lexical aspects of the syntax definition. In this module we define the sections: lexical syntax (line `03`), lexical restrictions (line `15`) and context-free restrictions (line `18`).

In line `14` we show how identifiers are formed in *P*L: the first character must be a letter or an underscore, followed by zero or more letters, numbers or underscores.

*P*L has a set of keywords that are not allowed to be used as identifiers. However, in SDF, the keywords (e.g. class, inherits) are not automatically preferred over identifiers. Thus, ambiguity can happen. So, we use SDF *reject productions* (lines 05 and 06) to define explicitly that these keywords are not allowed as identifiers. Reject productions define that all derivations of a symbol for which there is a reject production are forbidden SDF (2013).

SDF supports constructs to define in a declarative way that certain kinds of derivations are not allowed; this is also known as disambiguation filters. In line `16`, there is an example of this: a lexical restriction that specifies that an identifier cannot be followed by a character that is allowed in an identifier. Similarly, in line `19`, we define that a keyword cannot be followed by a character that is allowed in an identifier.

5.3 Translation Rules

In Figure 5.3 we show how the translation rules, previously presented in Section 4.1.2, are defined using the Stratego transformation language. The translation rules take a *P*L specification based on grammar, Figures 5.1 and 5.2, and produce as output its corresponding metamodel and constraints.

Basic transformations are defined using conditional term rewrite rules that are combined with strategies to control the application of rules. With Stratego, basic transformation rules can be defined separately from the strategy that applies them. This allows that they can be understood independently Stratego (2013). The rules defined using Stratego language act over the AST of a *P*L specification.

```
module PerfLang
 imports Common
exports
 context-free start-symbols
  Classes
 context-free syntax
  ClDeclaration* -> Classes {cons("Classes")}
  ClassModifier ClassDeclName "^=" ClassBody "end" ";"
                    -> ClassDeclaration {cons("ClassDecl")}
  "final"    -> ClassModifier {cons("ClModifFinal")}
  "deferred" -> ClassModifier {cons("ClModifDeferred")}
  ""         -> ClassModifier {cons("NoClModif")}
  "class" ID -> ClassDeclName {cons("ClassDeclName")}
  AbstractDeclarations         -> ClassBody {cons("ClBodyA")}
  "inherits" ID AbstractDeclarations -> ClassBody {cons("ClBodyB")}
  "abstract" AbsMbrDecl* -> AbstractDeclarations {cons("AbsMbrsDecls")}
  "var" ID ":" TypeExpr ";" -> AbsMbrDecl {cons("AbsVarDecl")}
  "invariant" ListOfPredicates ";" -> AbsMbrDecl {cons("ClInv")}
  BasType -> PredefType {cons("PredefType")}
  ID        -> ClassNameAsType {cons("ClAsTyp")}
  "from" ID -> ClassNameAsType {cons("ClAsTyp")}
  PredefType  -> Type
  ClassNameAsType -> Type
  ...
  "set" "of" Type -> SetOfType {cons("SetOfType")}
  ...
  Type     -> TypeExpr
  SetOfType -> TypeExpr
  ...
  {Predicate ","}+ -> ListOfPredicates {cons("ListOfPredicates")}
  Expression       -> Predicate {cons("Predicate")}
  ForAllExpr   -> Expression {cons("Expression")}

  "forall" BndVarDecl  ":-" Expression
              -> ForAllExpr {cons("ForAllExpr")}
  ...
  ID "::" CollExpr -> BndVarDecl {cons("BoundVarDecl")}
  ...
```

Figure 5.1 *P*L grammar expressed in SDF: `main` module

```
module Common
exports
 lexical syntax
  [a-zA-Z\_][a-zA-Z0-9\_]* -> ID
  Keyword -> ID {reject}
  BasType -> ID {reject}
  class -> Keyword
  inherits -> Keyword
  ...
  bool -> BasType
  ...
  "~" -> NOT
  "==>" -> IMPLIES
  ...
 lexical restrictions
  ID -/- [a-zA-Z0-9\_]
  ...
 context-free restrictions
  class inherits -/- [A-Za-z0-9\_]
  ...
```

Figure 5.2 *P*L grammar expressed in SDF: `common` module

The AST is expressed in terms of an Annotated Term Format (ATerms, for short). ATerm is a structured representation generated after a parser reads the input text (a PL specification) and turns it into abstract syntax tree. Given the grammar using SDF (Figures 5.1 and 5.2), this parser is generated automatically by the Spoofax Workbench.

A basic unconditional rewrite rule, using Stratego language, has the following form: `r: t1 -> t2`, where `r` is the rule name, `t1` is the left-hand side and `t2` the right-hand side term pattern. The rule `r` applies to a term `t` when the pattern `t1` matches `t`, resulting in the instantiation of `t2`. Conditional basic rules have the form: `r: t1 -> t2 where s`, where `s` is the condition. We also use rules of the form:

```
r:
  t1 -> t2
  with
    x := y;
    ...
```

Using the `with` clause, instead of `where`, variables can be assigned and other rules can be invoked.

Our rules produce its output by string interpolation, using the `$[ ... ]` brackets, to construct the metamodel as text fragments. Variables can be inserted using brackets without a dollar: `[ ... ]` (e.g. line `08` of Figure 5.3). String interpolation allows the combination of text with variables. Any indentation used is preserved in the end result, except the indentation leading up to the quotation.

Rules written in the Stratego language (Figure 5.3) do not necessarily have a one to one (1-1) relation with the rules expressed in Section 4.1.2 using a platform independent notation. For example, in Figure 5.3, the rule defined in line `16`, represents Rules 6, 7 and 8 defined in Section 4.1.2. Whereas the rule defined in line `01` represents only the Rule 1.

The rules are divided in four groups: *Class declaration*, *Abstract Declarations*, *Corresponding Types* (not shown) and *Invariants*. For conciseness, we show only one (or two) rule(s) of each group. See Silva (2013) for all the rules.

Class Declaration Rule 1 (line `01`) starts the translation and it triggers all the other rules. It represents a complete class declaration, whose main elements are its modifier (`modif`), its name (`name`), and its body (`body`).

Rule 1 triggers other three rules, where the rule associated to `modif` is not presented here. Rule 3 (line `08`) deals with *name* producing `class` *`class_id`* as output. Rules 4 (line `10`) and 5 (line `13`) deals with `body`.

Rule 4 deals with a non-inherited (not shown) and Rule 5 with an inherited class producing `extends ID {` as output. Both rules delegate the processing of the abstract declarations (abstract variables and invariants) to Rule 6 (line `16`).

```
01:  to-emfatic:
02:   ClassDecl(modif,name,body) ->  $[ [modif'] [name'] [body'] } ]
03:   with
04:     body'   := <to-emfatic> body;
05:     name'   := <to-emfatic> name;
06:     modif'  := <to-emfatic> modif
07:
08:  to-emfatic: ClDeclName(x) -> $[class [x]]
09:
10:  to-emfatic: ClBodyA(decls) -> $[{ [decls'] ]
11:      with decls' := <to-emfatic> decls
12:
13:  to-emfatic: ClBodyB(c, decls) -> $[ extends [c] { [decls'] ]
14:      with decls' := <to-emfatic> decls
15:
```

Abstract declarations Rule 6 (line `16`) iterates over the abstract declarations of *P*L code, delegating to other rules (e.g. Rule 9) the remainder translation.

```
to-emfatic:
 ClassDecl(modif,name,body) ->  $[ [modif'] [name'] [body'] } ]
 with
   body'  := <to-emfatic> body;
   name'  := <to-emfatic> name;
   modif' := <to-emfatic> modif

to-emfatic: ClDeclName(x) -> $[class [x]]

to-emfatic: ClBodyA(decls) -> $[{ [decls'] ]
    with decls' := <to-emfatic> decls

to-emfatic: ClBodyB(c, decls) -> $[ extends [c] { [decls'] ]
    with decls' := <to-emfatic> decls

to-emfatic: AbsMbrsDecls(d*) -> $[ [d'*] ]
    with d'* := <to-emfatic> d*

to-emfatic: AbsVarDecl(x, t) ->
  $[
    [f] [t'] [x];
   ]
  with
    f  := <feature-kind> t;t' := <to-type> t

feature-kind: f -> x
 where
 switch !f
   case !f => PredefType(t) : x := "attr"
   ...
 end

to-type: PredefType(t) -> x
where
 switch !t
  case "bool"    : x := "boolean"
  ...
 end

to-type: ClAsType(t) -> t

to-evl: ClInv(LstPredicates(p*)) ->
  $[context CLASS_NAME {
    [p'*]
  }]
  with p'* := <to-evl> p*

to-evl: Predicate(Expression(e)) ->
  $[
    constraint CONSTRAINT_NAME {
       check: [e']
       message: "Put an error message here."
    }
   ]
  with e' := <to-expr> e

to-expr: ForAllExpr(BoundVarDecl(it, coll),
    Expression(expr)) -> $[ [coll'].forAll([it] | [expr']) ]
  with
    coll' := <to-expr> coll; expr' := <to-expr> expr
```

Figure 5.3 Translation Rules expressed using Stratego Language

Rule 9 (line `19`) maps a variable declaration in *P*L to a class feature in EMFatic that follows the pattern: *modifiers featureKind emfType id*. For us, we can ignore *modifiers*. The *featureKind* is the kind of the class feature, that can be *attr* (an EAttribute), *val* (an EReference with containment = true) and *ref* (an EReference, containment = false). The *emfType* is the type of the feature and *id* is the identifier.

Rule 14 (line `42`) is the responsible to start the generation of the EVL code, which define the constraints over instances of the resulting metamodel. The *`class_id`* is the name of the class (in *P*L) taken as source of the translation rules.

```
16:  to-emfatic: AbsMbrsDecls(d*) -> $[ [d’*] ]
17:      with d’* := <to-emfatic> d*
18:
19:  to-emfatic: AbsVarDecl(x, t) ->
20:    $[
21:      [f] [t’] [x];
22:     ]
23:    with
24:      f  := <feature-kind> t;t’  := <to-type> t
25:
26:  feature-kind: f -> x
27:   where
28:   switch !f
29:     case !f => PredefType(t) : x := "attr"
30:     ...
31:   end
32:
33:  to-type: PredefType(t) -> x
34:  where
35:   switch !t
36:    case "bool"    : x := "boolean"
37:    ...
38:   end
39:
40:  to-type: ClAsType(t) -> t
41:
```

Invariants Rule 14 generates the EVL code that defines the context in which the list of invariants `LstPredicates(p*)` applies to.

For each invariant, Rule 15 (line `48`) is responsible for generating a constraint section named *`CONSTRAINT_NAME`* to the invariant *inv* and delegating to Rules 16.x the translation of the expression that define the invariant *inv* into equivalent expressions in EVL.

Rule 16.1 (line `57`) represents an universal quantified invariant over a collection of items (in *P*L) and translates it into a correspondent in the EVL notation. In this rule, `x` is an item of the collection `coll`, `expr` is an boolean `Expression` involving `x`.

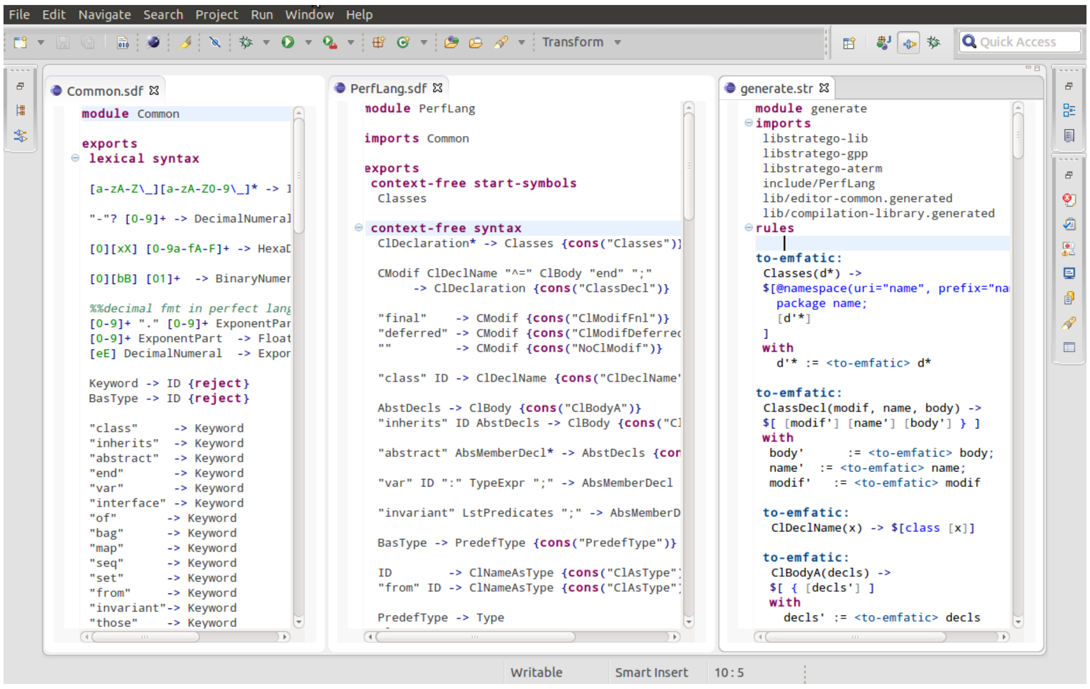
```
42:  to-evl: ClInv(LstPredicates(p*)) ->
43:    $[context class_id {
44:      [p'*]
45:    }]
46:    with p'* := <to-evl> p*
47:
48:  to-evl: Predicate(Expression(inv)) ->
49:    $[
50:      constraint CONSTRAINT_NAME {
51:         check: [inv']
52:         message: "Put an error message here."
53:      }
54:     ]
55:    with inv' := <to-expr> inv
56:
57:  to-expr: ForAllExpr(BoundVarDecl(x, coll),
58:      Expression(expr)) -> $[ [coll'].forAll([x] | [expr']) ]
59:    with
60:      coll' := <to-expr> coll; expr' := <to-expr> expr
```

5.4 Tool

In Figure 5.4, we show the artifacts, previously presented in Section 5.2 and Section 5.3, used as input for the Spoofax Workbench to generate our translation tool. On the left-hand side and in the middle of Figure 5.4 we show the grammar modules. On the right-hand side we see the rules. After Spoofax generates the *PL2EMF* tool, we now can use to it translate a *P*L formal specification given as input to generate the corresponding metamodel and constraints. For example, in Figure 5.5 (left-hand side), we show the *PL2EMF* tool using as input the Fault Tree formal specification presented in Chapter 6.

As mentioned in Chapter 3, only the syntax and static semantics (SS_L) of a DSL L declarations are considered in the translation. The removal of the *P*L elements that are not considered in the extraction is responsibility of a preprocessing phase.

On the right-hand side of Figure 5.5, we can see the abstract syntax tree of our *P*L



```
module Common

exports
 lexical syntax

 [a-zA-Z\_][a-zA-Z0-9\_]* -> I

 "-"? [0-9]+ -> DecimalNumeral

 [0][xX] [0-9a-fA-F]+ -> HexaD

 [0][bB] [01]+  -> BinaryNumer

 %%decimal fmt in perfect lang
 [0-9]+ "." [0-9]+ ExponentPar
 [0-9]+ ExponentPart  -> Float
 [eE] DecimalNumeral  -> Expor

 Keyword -> ID {reject}
 BasType -> ID {reject}

 "class"     -> Keyword
 "inherits"  -> Keyword
 "abstract"  -> Keyword
 "end"       -> Keyword
 "var"       -> Keyword
 "interface" -> Keyword
 "of"       -> Keyword
 "bag"      -> Keyword
 "map"      -> Keyword
 "seq"      -> Keyword
 "set"      -> Keyword
 "from"     -> Keyword
 "invariant"-> Keyword
 "those"    -> Keyword
```

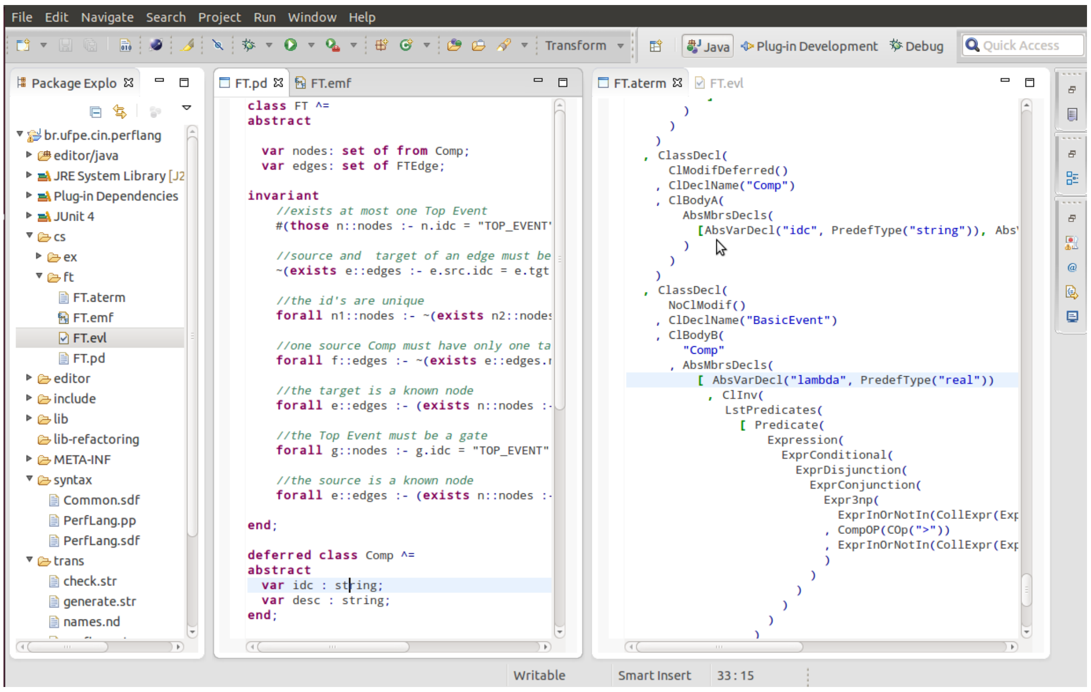
```
module PerfLang

imports Common

exports
 context-free start-symbols
  Classes

 context-free syntax
  ClDeclaration* -> Classes {cons("Classes")}

  CModif ClDeclName "^=" ClBody "end" ";"
       -> ClDeclaration {cons("ClassDecl")}

  "final"    -> CModif {cons("ClModifFnl")}
  "deferred" -> CModif {cons("ClModifDeferrec
  ""         -> CModif {cons("NoClModif")}

  "class" ID -> ClDeclName {cons("ClDeclName'

  AbstDecls -> ClBody {cons("ClBodyA")}
  "inherits" ID AbstDecls -> ClBody {cons("Cl

  "abstract" AbsMemberDecl* -> AbstDecls {con

  "var" ID ":" TypeExpr ";" -> AbsMemberDecl

  "invariant" LstPredicates ";" -> AbsMemberD

  BasType -> PredefType {cons("PredefType")}

  ID         -> ClNameAsType {cons("ClAsType"
  "from" ID -> ClNameAsType {cons("ClAsType"

  PredefType -> Type
```

```
module generate
imports
 libstratego-lib
 libstratego-gpp
 libstratego-aterm
 include/PerfLang
 lib/editor-common.generated
 lib/compilation-library.generated
rules
    |
to-emfatic:
 Classes(d*) ->
 $[@namespace(uri="name", prefix="na
   package name;
   [d'*]
 ]
 with
   d'* := <to-emfatic> d*

to-emfatic:
 ClassDecl(modif, name, body) ->
 $[ [modif'] [name'] [body'] } ]
 with
  body'      := <to-emfatic> body;
  name'  := <to-emfatic> name;
  modif'   := <to-emfatic> modif

 to-emfatic:
  ClDeclName(x) -> $[class [x]]

 to-emfatic:
  ClBodyA(decls) ->
  $[ { [decls'] ]
  with
    decls' := <to-emfatic> decls
```

Figure 5.4 Fault Tree SDF grammar modules and Stratego rules

expressed using the Annotated Term Format (ATerms). The notation ATerms is the format adopted by Spoofax for representation of abstract syntactic trees. Recall from Section 5.3 that rules defined using the Stratego language also act over ATerms of a *PL* specification.

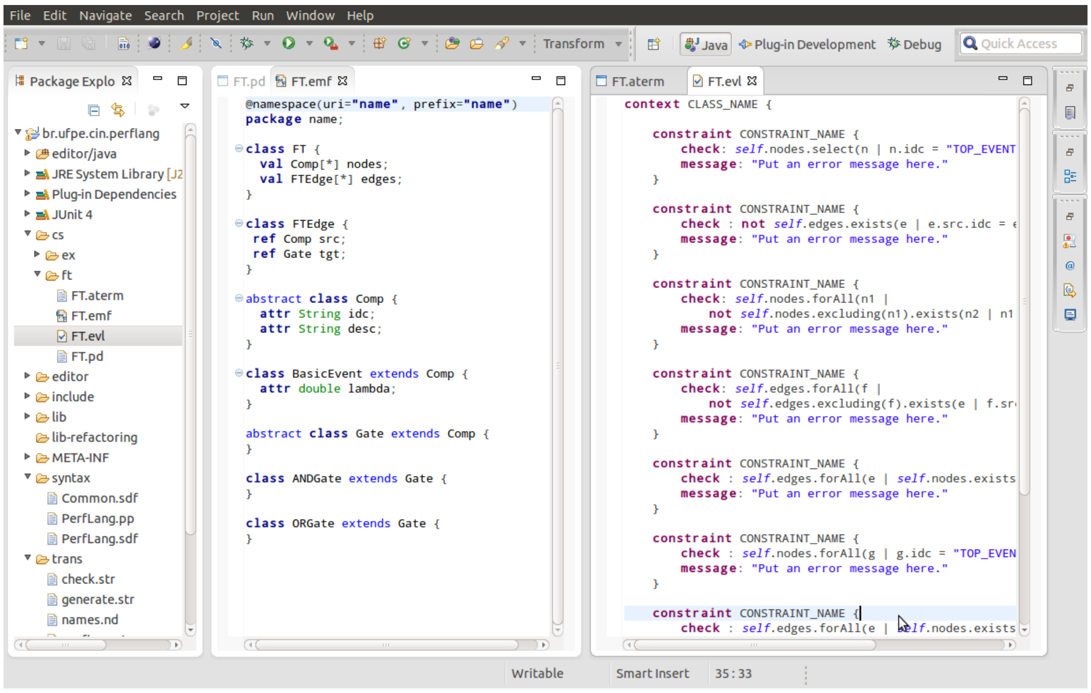
Figure 5.6 shows the metamodel and constraints resulting from the application of the translation rules over the FT formal specification. It is important noting that the developer (specifier) can add manually EuGENia annotations in the generated metamodel to improve its graphical concrete syntax. These annotations are responsible to give, for example, the shapes of a circle and an arrow to *BasicEvent* and *FTEdge* class, respectively. See our case study in Chapter 6. A significant name for the constraints, `CONSTRAINT_NAME`, is also a manual task.



```
class FT ^=
abstract

  var nodes: set of from Comp;
  var edges: set of FTEdge;

invariant
    //exists at most one Top Event
    #(those n::nodes :- n.idc = "TOP_EVENT'

    //source and  target of an edge must be
    ~(exists e::edges :- e.src.idc = e.tgt

    //the id's are unique
    forall n1::nodes :- ~(exists n2::nodes

    //one source Comp must have only one ta
    forall f::edges :- ~(exists e::edges.r

    //the target is a known node
    forall e::edges :- (exists n::nodes :-

    //the Top Event must be a gate
    forall g::nodes :- g.idc = "TOP_EVENT"

    //the source is a known node
    forall e::edges :- (exists n::nodes :-

end;

deferred class Comp ^=
abstract
  var idc : string;
  var desc : string;
end;
```

```
          )
        )
      )
    , ClassDecl(
        ClModifDeferred()
      , ClDeclName("Comp")
      , ClBodyA(
          AbsMbrsDecls(
            [AbsVarDecl("idc", PredefType("string")), Abs'
          )
        )
      )
    , ClassDecl(
        NoClModif()
      , ClDeclName("BasicEvent")
      , ClBodyB(
          "Comp"
        , AbsMbrsDecls(
            [ AbsVarDecl("lambda", PredefType("real"))
            , ClInv(
                LstPredicates(
                  [ Predicate(
                      Expression(
                        ExprConditional(
                          ExprDisjunction(
                            ExprConjunction(
                              Expr3np(
                                ExprInOrNotIn(CollExpr(Exp
                              , CompOP(COp(">"))
                              , ExprInOrNotIn(CollExpr(Exp
                              )
                            )
                          )
                        )
                      )
                    )
```

Figure 5.5 Fault Tree Formal Specification and its Abstract Syntax using ATerms



```
@namespace(uri="name", prefix="name")
package name;

class FT {
  val Comp[*] nodes;
  val FTEdge[*] edges;
}

class FTEdge {
 ref Comp src;
 ref Gate tgt;
}

abstract class Comp {
  attr String idc;
  attr String desc;
}

class BasicEvent extends Comp {
  attr double lambda;
}

abstract class Gate extends Comp {
}

class ANDGate extends Gate {
}

class ORGate extends Gate {
}
```

```
context CLASS_NAME {

    constraint CONSTRAINT_NAME {
        check: self.nodes.select(n | n.idc = "TOP_EVENT
        message: "Put an error message here."
    }

    constraint CONSTRAINT_NAME {
        check : not self.edges.exists(e | e.src.idc = e
        message: "Put an error message here."
    }

    constraint CONSTRAINT_NAME {
        check: self.nodes.forAll(n1 |
            not self.nodes.excluding(n1).exists(n2 | n1
        message: "Put an error message here."
    }

    constraint CONSTRAINT_NAME {
        check: self.edges.forAll(f |
            not self.edges.excluding(f).exists(e | f.sr
        message: "Put an error message here."
    }

    constraint CONSTRAINT_NAME {
        check : self.edges.forAll(e | self.nodes.exists
        message: "Put an error message here."
    }

    constraint CONSTRAINT_NAME {
        check : self.nodes.forAll(g | g.idc = "TOP_EVEN
        message: "Put an error message here."
    }

    constraint CONSTRAINT_NAME {
        check : self.edges.forAll(e | self.nodes.exists
```

Figure 5.6 Generated FT metamodel and constraints

6 Case Study

In this chapter, we present the development of a Fault Tree Analysis Tool following our proposed methodology. In Section 6.1 we present a brief overview concerning Fault Trees (FT for short) and our FT formal specification, from which we generate the tool back-end automatically using the *Perfect* Developer Tool. In Section 6.2 we exercise our translation rules over a fragment of the FT formal specification. In Section 6.3 we present the extracted metamodel and respective constraints, from which the resulting FT GMF front-end is generated by the GMF tools. In Section 6.4 we show how we linked front and back-end. Additionally, in Section 6.5 we give an overview about formal verification proof aspects of the Fault Tree formal specification. The complete FT specification can be found in Appendix A and the analysis tool can be found in (Silva, 2013).

6.1 FT Overview and Formal Model

A Fault Tree (FT) (NASA, 2002) is a kind of combinatorial model commonly used to find how an undesired event of interest (called the top event) might be caused by some combination of other undesired events (failures). Relationships among events are described by *AND*- (Figure 6.1(b)) and *OR-Gates* (Figure 6.1(c)). An undesired event usually represents a state of the system that is critical from a safety or reliability point of view. Basic events (Figure 6.1(a)) fail spontaneously according to some failure rate. Events represented by gates depend on basic events and/or other gates. Basic events do not have inputs (they are independent events). Gates have an arbitrary number (at least one) of inputs (dependent events). See Figure 6.1.

Mathematically, an FT is a directed acyclic graph (a tree), where each vertex is an FT component. Components can be basic events or gates. Gates can be *AND*- or *OR*- logical ports. We declare the class *Component* as ***deferred*** to allow dynamic binding, see Figure

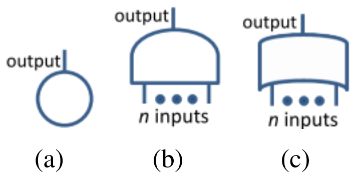


Figure 6.1 Components of a FT

6.2. Variables of type *Component* can receive any instance from any of its descendant classes. In the definition of *Component*, the functions *idc* and *desc* have the same name of their class attributes to allow public access.

The class *Gate* specializes the class *Component*, without any additional attributes or operations (Figure 6.2). It contains two constructors (not shown) that calls the *Component*'s constructors. *AND-Gates*, and similarly *OR-Gates*, are just a specialization of *Gate* and consequently of *Component*.

We assume that components represented by basic events have an exponentially distributed life time. Then, its failure probability at time t is computed by using the function

$$probf(t) = 1 - e^{-\lambda t}, t \geq 0$$

where λ is its failure rate and $e \approx 2.718281828$. As we can see, the class *BasicEvent* extends *Component* by adding the attribute *lamba* (λ) that holds its failure rate. Additionally, a constant definition (E) is declared to represent the e constant and the function $probf$ is translated according to PL notation.

An edge, $e = (src, tgt)$, denotes that the output of a *source* component is an input to a *target* component. In the class *FTEdge*, left-hand side of Figure 6.3, we have one attribute with type ***from*** *Component* and other with type ***from*** *Gate*. This means these variables can be assigned to any descendants of *Component* and *Gate*, respectively. The same is valid for *nodes:* ***set of from*** *Component*, in the *FT* class. This says that *nodes* can contain an arbitrary number of elements (descendants of) *Component*.

Looking at the *FT* class (right-hand side of Figure 6.3) we can see the schema *addFTNode*. It changes the state of the system by adding a new node *n*, passed as parameter, if it was not already in the set of *nodes* (*~checkFTNode(n)*). Additionally, if node *n* is the top event (*n.idc = "TOP_EVENT"*), it must be a gate (*n* ***within from*** *Gate*) and there must not be another event labeled as top event (function *existsTOPEventId*, without input parameters). If this precondition holds, the postcondition of this operation states that the attribute *nodes*! (after the operation) is equal to itself before the operation,

```
deferred class Component ^=
abstract
  var idc : string;
  var desc : string;
interface
  operator = (arg);
  function idc;
  function desc;
  build {!idc: string}
   post desc! = “”;
end ;
class ANDGate ^= inherits Gate
interface
  build {idEv: string}
   inherits Gate{idEv};
end ;

class ORGate ^= inherits Gate
interface
  build {idEv: string}
   inherits Gate{idEv};
end ;
```

```
deferred class Gate ^=
    inherits Component
  ...
end ;

class BasicEvent ^=
    inherits Component
abstract
  const E : real ^= 2.718281828;
  var lambda : real;
  invariant lambda > 0.0 ;
interface
  function lambda;
  function probf(t:int):real
  pre t>=0
   ^=(1 - E^(-(lambda*t)));
    assert result >=0;
  build {idEv: string, lb: real}
  pre lb > 0.0
    inherits Component{idEv};
  post lambda! = lb;
end ;
```

Figure 6.2 Fault Tree Components: AND-Gate, OR-Gate and Basic Event

```
class FTEdge ^=
abstract
  var src: from Component;
  var tgt: from Gate;
interface
  function src;
  function tgt;
  build {!src: from Component,
    !tgt: from Gate};
end ;
```

```
class FT^=
abstract
  var nodes: set of from Component;
  var edges: set of FTEdge;
invariant
 //list of invariants (shown below)...
interface
  schema !addFTNode (n: from Component)
   pre ~checkFTNode(n),
  (n.idc = “TOP_EVENT”
   ==> ~existsTOPEventId &
    (n within from Gate))
   post nodes! = nodes.append(n);
...
end ;
```

Figure 6.3 The *FTEdge* definition (left-hand) and a fragment of the main class, *FT*, of the formal specification (right-hand)

nodes, appended with the new node *n*. Any instance of the FT model must satisfy the following invariants (the list is not exhaustive):

inv1 There is a unique top event

*#(**those** n::nodes :- n.idc = "TOP_EVENT") <= 1*

inv2 Source and target of an edge must be different. That is, it is not allowed self-loops

*~(**exists** e::edges :- e.src.idc = e.tgt.idc)*

inv3 The id's are unique

***forall** n1::nodes :- ~(**exists** n2::nodes.remove(n1) :- n1.idc = n2.idc)*

inv4 One source component must have only one target component

***forall** f::edges :- ~(**exists** e::edges.remove(f) :- f.src.idc = e.src.idc)*

As the main goal of this work is to present our proposed rigorous methodology (Chapter 3), we assume simplifications concerning the failure probability calculation of the top event: (i) in *AND-Gates*, input events are considered to be independents and (ii) in *OR-Gates*, their input events are mutually exclusive.

Then, traversing the FT, and making multiplications (AND-gates) and sums (OR-gates) we can calculate the top event failure probability (indeed an approximation) of a fault tree. The function definition that makes this calculation can be found in Appendix A. For a complete description about Fault Tree Quantitative Analysis, please see (SAE, 1996) and (NASA, 2002).

6.2 Exercising the Translation Rules

Now we show an example of how the rules we presented in Section 4.1.2 work together to produce the correspondent EMFatic/EVL code from the Fault Tree formal specification expressed in *P*L. We choose the class named *Component* to illustrate how the translation rules work. For this class, constraints in EVL are not produced as output because they do not have invariants defined in its body.

Recall from Section 4.1.2 that Rule 1 is the starting point of the translation and it triggers all the other rules. It takes as argument the complete *Component* class declaration and breaks it into three main elements: the class modifiers, the class name and its body. Rule 1 delegates, respectively, these elements to the Rules 2, 3 and 4.

Rule 1. [***deferred class*** *Component* ^= ***abstract***
var *idc : string;* ***var*** *desc : string;* ***end;***]^{classDecl} ▶
[***deferred***]^{classModif} [***class*** *Component*]^{classDeclName}
[***abstract var*** *idc : string;* ***var*** *desc : string;* ***end;***]^{classBody} ' } '

Rule 2 translates the class modifier ***deferred*** in P*L* to its corresponding in EMFatic.

Rule 2. [***deferred***]^{classModif} ▶ `abstract`

Rule 3. [***class*** *Component*]^{classDeclName} ▶ `class Component`

Rule 4 deals with a non-inherited class. It delegates the processing of the abstract declarations (abstract variables and invariants) to Rule 6 (non-empty ***abstract*** section) or 7 (empty ***abstract*** section) . In this context, Rule 4 delegates to Rule 6.

Rule 4 . [***abstract*** ***var*** *idc : string;* ***var*** *desc : string;*]^{classBody} ▶
{ [***abstract*** ***var*** *idc : string;* ***var*** *desc : string;*]^{abstractDecls}

As partial result of the previous rules applications we have:

[***deferred*** ***class*** *Component* '^=' ***abstract***
var *idc : string;* ***var*** *desc : string;* ***end*** *;*]^{classDecl} ▶
`abstract  class Component {`
[***abstract*** ***var*** *idc : string* ; ***var*** *desc : string;*]^{abstractDecls}
`}`

Rule 6 applies when the ***abstract*** section contains a non-empty list of statements (variables and invariants declarations), delegating this list to Rule 8. The Rule 8 iterates over each statement, delegating each statement, in this case, to Rule 9 the remainder translation. Applying Rule 6 and Rule 8 (twice) we obtain:

Rule 6. [***abstract var*** *idc : string;* ***var*** *desc : string;*]^{abstractDecls} ▶
[***var*** *idc : string*; ***var*** *desc : string*;]^{listAbsMemberDecls}

Rule 8. [***var*** *idc : string;* ***var*** *desc : string*;]^{listAbsMemberDecls} ▶
[***var*** *idc : string*]^{absMemberDecl} [***var*** *desc : string*;]^{listAbsMemberDecls}

[***var*** *idc : int;* ***var*** *desc : string*;]^{listAbsMemberDecls} ▶
[***var*** *idc : string*]^{absMemberDecl} [***var*** *desc : string*]^{absMemberDecl}

```
class FTEdge {
  ref Component src;
  ref Gate tgt;
}
abstract  class Component {
  attr String idc;
  attr String desc;
}
class FT {
  val Component[*] nodes;
  val FTEdge[*] edges;
}
```

```
context FT {
  constraint inv1 { check : self.nodes.
   select(n | n.idc = "TOP_EVENT").size() <= 1}
  constraint inv2 { check :  not self.edges.exists(e |
     e.src.idc = e.tgt.idc) }
  constraint inv3 {check: self.ftnodes.forAll(n1 |
    not self.ftnodes.excluding(n1).
        exists(n2 | n1.idc = n2.idc))}
  constraint inv4 {
   check : self.edges.forAll(f |
     not self.edges.excluding(f).exists(e |
      f.src.idc = e.src.idc))}
}
```

Figure 6.4 The FT EMF model (left-hand) with some constraints (right-hand)

Rule 9 maps the ***var*** *idc : string* variable declaration in PL to a class feature in EMFatic. The same happens with ***var*** *desc : string* variable declaration (not shown).

Rule 9. $[\textbf{\textit{var}}\ \textit{idc : string}]^{absMemberDecl}$ ▶
$[\textit{string}]^{emfFeatureKind}$ $[\textit{string}]^{emfEquivType}$ `idc;`

With the application of Rules 11.1 and 12 we complete the transformation:

$[$***deferred class*** *Component '^=' **abstract***
var *idc : string;* ***var*** *desc : string;* ***end****;*$]^{classDecl}$ ▶

```
abstract class Component {
  attr string idc;
  attr string desc;
}
```

6.3 FT GMF Editor

By applying the translation rules (briefly presented in Section 4.1.2) with the aid of our tool *PL2EMF*, on our FT specification of Section 6.1, we obtain automatically the metamodel and constraints showed in Figure 6.4. From this metamodel, we apply EuGENia that automatically generates the graphical editor.

An *FTEdge* references (***ref***) the source and target components. A Fault Tree contains (***val***) zero or more components and edges. On the right-hand side of Figure 6.4 we have the invariants *inv1*, *inv2*, *inv3* and *inv4* translated to EVL.

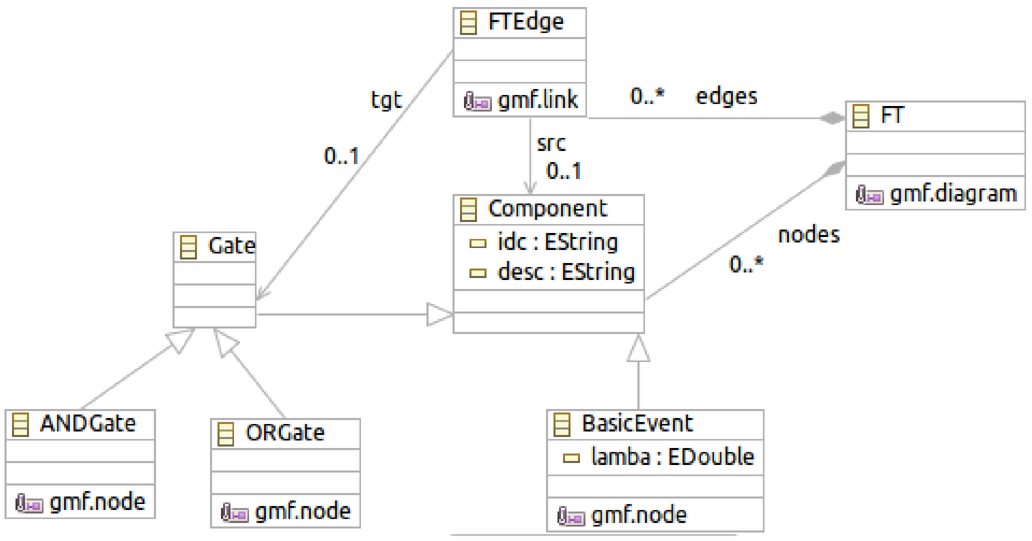


Figure 6.5 Diagram of the FT EMF model (FT Metamodel)

For readability purposes, in Figure 6.5 we show the FT metamodel (previously presented —see Figure 6.4— using a textual notation) using the standard visual notation of class diagrams. In this diagram we can see the structural relationships and its cardinality between the elements of a Fault Tree.

Figure 6.6(a) presents the FT Analysis Tool obtained, following our rigorous proposed methodology. On the left-hand side we have a palette, where we can select components (gates and basic events) and connection edges.

6.3.1 Calculating Top Event Failure Probability

Consider a fault-tolerant multiprocessor computer system with multiple shared memory modules connected by a bus (NASA, 2002). The system is operational if at least one processor (in a total of three), one memory module (in a total of two) and the bus are operational. This system is modeled by the FT presented on the right-hand side of Figure 6.6(a).

We assume that the failure time for each component is exponentially distributed. Then, let $\lambda_P = 0.00125$, $\lambda_M = 0.00643$ and $\lambda_B = 0.00235$ be the failure rate of each processor, memory and bus, respectively. From this, the failure time distribution function of a processor, for example, is given by

$$probf(t)_P = 1 - e^{-0.00125t}, t \geq 0$$

To calculate the Top Event Failure Probability using our FT Analysis Tool, we right-click in the drawing blank area with the mouse and select the option *Wizards* → *Calculate Top Event Failure Probability*. As result, the input dialog of Figure 6.6(b) is presented where the user can provide the time (t). Suppose that we provide 10.0 as input for t. After clicking on the OK button, the failure probability of the FT Top Event is calculated and presented as can be seen in Figure 6.6(c). This calculation happens in operations located in the back-end part of the tool. These operations are called by means of the integration plugin that links the back and front-end of the FT Analysis Tool.

6.3.2 Validating the Fault Tree

The work reported in (Maciel *et al.*, 2011) presents a system in which software application that can read, write and modify the content of the storage device *Disk1*. The system periodically replicates the produced data of one storage device (*Disk1*) in two storage replicas (*Disk2* and *Disk3*) to allow recovering in case of data loss or data corruption. The system is also composed of one *Server* and *Hub* that connects the *Disk2* and *Disk3* to the server (Figure 6.7(a)). The system is considered to have failed if it is not possible to read, write or modify data on *Disk1* and if no data replica is available. Hence, if *Disk1* or the *Server* or the *Hub*, or either replica storages are faulty, the system fails. The respective FT is presented on the right-hand side of the Figure 6.7(b).

We use this model to show how constraints validation works through the front-end. Note that we added two extra edges: one extra egde from the gate that represents the top event *SysFail* to itself (Figure 6.7(b)) and other from the basic event that represents the *Hub* to the gate that represents the failure of replicas. Selecting the option *Validate* in the *Edit* menu, we get an error message saying that the invariants *inv2* (source and target of an edge must be different) and *inv4* (one source component must have only one target components) (Figure 6.7(b)). This small example shows that the invariants that hold in the FT formal specification, also hold in the GUI created from its metamodel.

We checked other invariant violation possibilities and all of them matched the expected results as characterized in the *PL* formal specification.

6.4 FT Tool: linking front and back-end

In Figure 6.8, we show how the link of the FT tool was implemented. On the left-hand side we see the library, *backendftree.jar* from the FT formal specification, generated automatically by PD. On the top of the right-hand side we see the EWL wizard responsible to interact with the GUI user that requests a FT failure probability. Wizards access underlying elements of a graphical FT model and make calls to adapters (in this case, for class `AdapterFT.java`) that holds an instance of the back-end.

The adapter represents the external environment that interacts with the back-end interface. This back-end interface we call: *wrapper*. The wrapper provides a common access point for all operations provided by the DSL specification.

6.5 Formal Verifications

For the Fault Tree specification, PD generated 71 verification conditions, from which 70 were confirmed automatically.

The unproven condition is the post-assertion ***assert result*** *>= 0* in the function *probf()* of the *BasicEvent*, shown in Figure 6.9. This post-assertion specifies an additional property that the *probf()* return are expected to hold. Within the post-assertion, the predefined identifier ***result*** refers to the result of the function. Each expression in the post-assertion must refer to ***result***.

Recall from Section 6.1 that this function represents the failure probability at time t of a FT basic event (using mathematical notation: $probf(t) = 1 - e^{-\lambda t}, t \geq 0$ where λ is its failure rate and $e \approx 2.718281828$).

In Figure 6.10, we show the report generated by PD of the unproven verification condition (or proof obligation) related to $prof()$ function. For each unproven condition the PD report shows: (i) the kind of the verification condition generated: in this case, “Post-assertion valid”; (ii) suggestions to the specifier to add in the specification to help the prover to prove the condition; (iii) the goal tried to prove: *0.0 ≤ (1-(E^-(self.lambda*t)))* (iv) the reason why it was not proved, in this case: exausted rules; and so on.

From Figure 6.10, we can see that the prover stopped at the expression

$$(2.71828\hat{} - (self.lambda * t)) \leq 1.0)$$

But we can easily see that it is valid because from $self.lambda > 0$ (class invariant) and $t \geq 0$ (precondition of $probf()$ function. So, the multiplication $self.lambda * t$ always produces a number greater than zero. For a better understanding, we show below this condition, using mathematical notation:

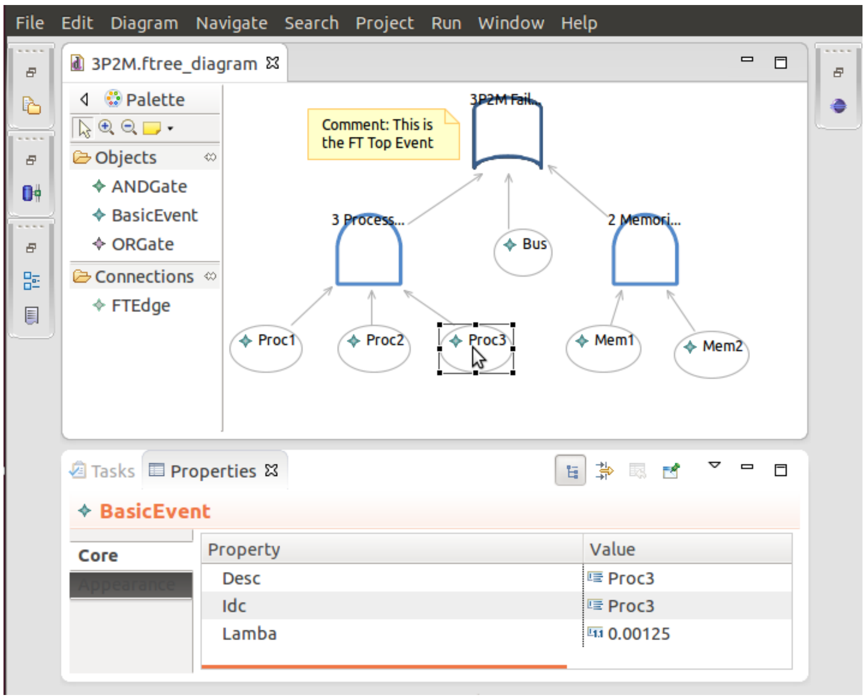
$$\frac{1}{e^{\lambda t}} \leq 1.0 \; where \; \lambda > 0.0, t \geq 0$$

We can also see the behavior of this function graphically in Figure 6.11. The expression $exp(x)$ means e^x. To our case, consider the parameter $x, x \geq 0$ as $x = \lambda t$. In the figure we can see that the function $1/e^x$ approach 0 (zero) as x approaches infinity and when $x = 0$ its result is 1 (one).

In Figure 6.12 we see an example of a successful simple proof. This verification condition checks one of the preconditions of the exponentiation *P*L built-in operator (ˆ) is satisfied it: checks whether the base of the exponentiation is greater than or equal to zero.

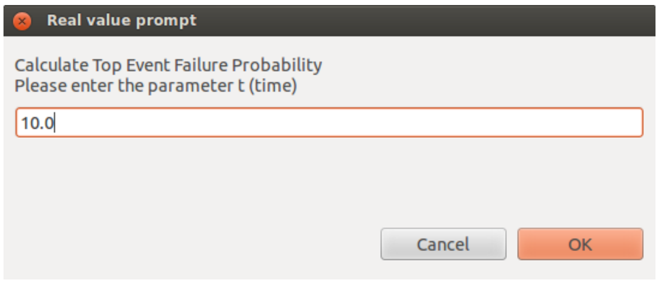
In Figure 6.13, we show a little bit more elaborated proof of a verification condition. The condition goal is to check whether the *BasicEvent* class invariant, $lambda > 0.0$ is satisfied by *BasicEvent*'s constructor (***build***).

It can be seen in Figure 6.13, the verification condition (or proof obligation) was generated in line 27:. This means that PD generated this proof obligation due to the piece of specification located in this line. This states that the new value of *lambda* (*lb* if the precondition of ***build*** is satisfied) must also satisfy the class invariant.



Property	Value
Desc	Proc3
Idc	Proc3
Lamba	0.00125

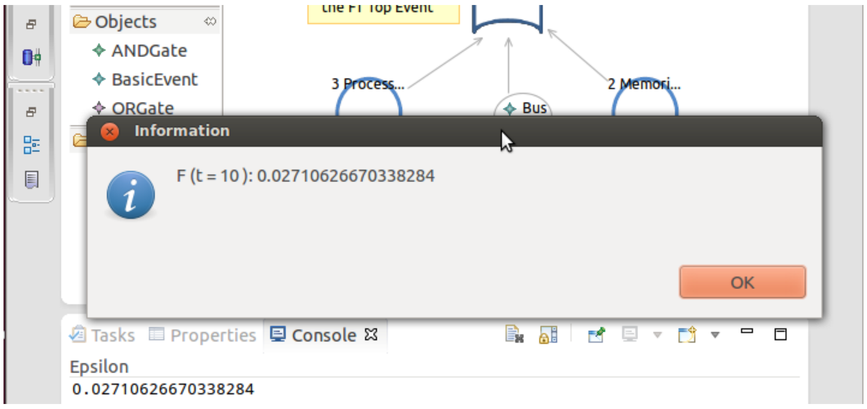
(a)



Calculate Top Event Failure Probability
Please enter the parameter t (time)

10.0

(b)

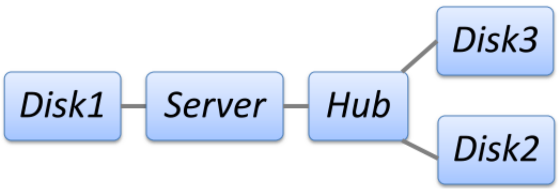


F (t = 10): 0.02710626670338284

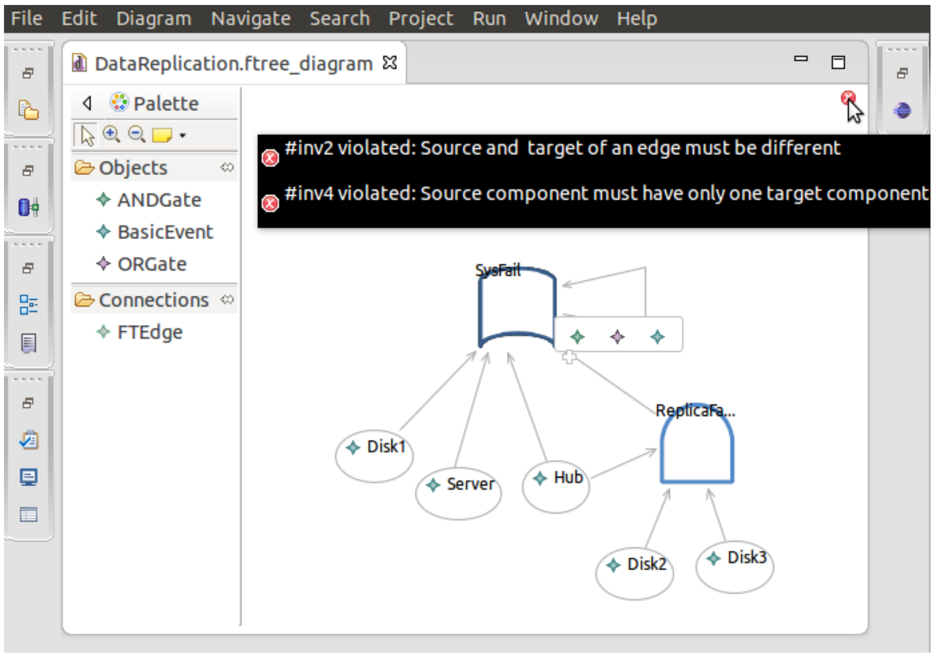
```
Epsilon
0.02710626670338284
```

(c)

Figure 6.6 FT Analysis Tool used to model a fault-tolerant multiprocessor computer system



(a)

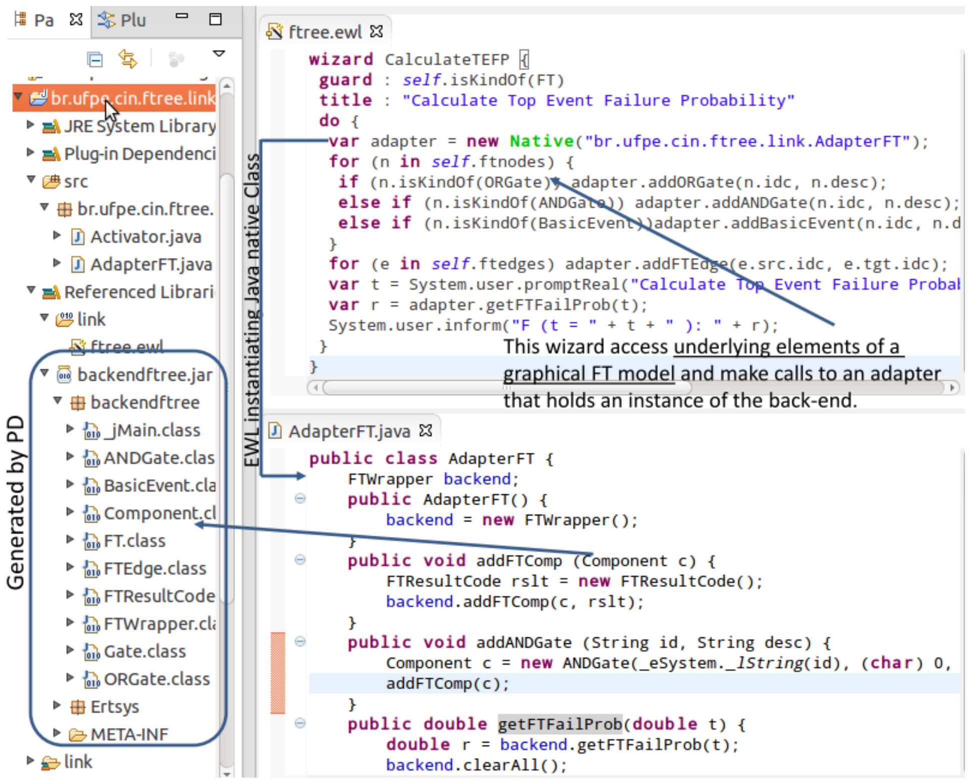


#inv2 violated: Source and target of an edge must be different

#inv4 violated: Source component must have only one target component

(b)

Figure 6.7 FT Analysis Tool used to model a data replication system



```
wizard CalculateTEFP {
 guard : self.isKindOf(FT)
 title : "Calculate Top Event Failure Probability"
 do {
  var adapter = new Native("br.ufpe.cin.ftree.link.AdapterFT");
  for (n in self.ftnodes) {
   if (n.isKindOf(ORGate)) adapter.addORGate(n.idc, n.desc);
   else if (n.isKindOf(ANDGate)) adapter.addANDGate(n.idc, n.desc);
   else if (n.isKindOf(BasicEvent))adapter.addBasicEvent(n.idc, n.d
  }
  for (e in self.ftedges) adapter.addFTEdge(e.src.idc, e.tgt.idc);
  var t = System.user.promptReal("Calculate Top Event Failure Probal
  var r = adapter.getFTFailProb(t);
  System.user.inform("F (t = " + t + " ): " + r);
 }
}
```

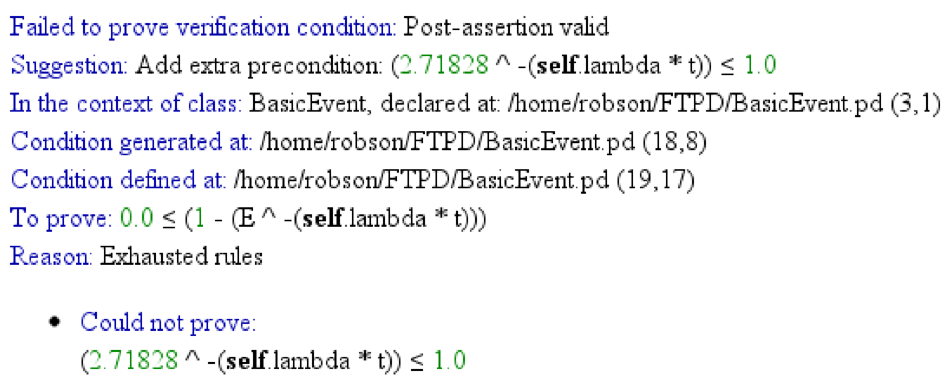
This wizard access underlying elements of a graphical FT model and make calls to an adapter that holds an instance of the back-end.

```
public class AdapterFT {
    FTWrapper backend;
    public AdapterFT() {
        backend = new FTWrapper();
    }
    public void addFTComp (Component c) {
        FTResultCode rslt = new FTResultCode();
        backend.addFTComp(c, rslt);
    }
    public void addANDGate (String id, String desc) {
        Component c = new ANDGate(_eSystem._lString(id), (char) 0,
        addFTComp(c);
    }
    public double getFTFailProb(double t) {
        double r = backend.getFTFailProb(t);
        backend.clearAll();
```

Figure 6.8 FT Tool Link plugin

```
Line | class BasicEvent ^=
     |     inherits Component
     | abstract
     |   const E : real ^= 2.718281828;
     |   var lambda : real;
10:  |   invariant lambda > 0.0 ;
     | interface
     |   ...
     |   function probf(t:real):real
     |   pre t>=0.0
18:  |     ^=(1 - E^(-(lambda*t)))
19:  |     assert result >=0.0;
     | ...
     |   build{idEv: string, lb: real}
     |   pre lb > 0.0
     |     inherits Component{idEv};
27:  |   post lambda! = lb;
     | end;
```

Figure 6.9 Basic Event class fragment



Failed to prove verification condition: Post-assertion valid
Suggestion: Add extra precondition: (2.71828 ^ -(**self**.lambda * t)) ≤ 1.0
In the context of class: BasicEvent, declared at: /home/robson/FTPD/BasicEvent.pd (3,1)
Condition generated at: /home/robson/FTPD/BasicEvent.pd (18,8)
Condition defined at: /home/robson/FTPD/BasicEvent.pd (19,17)
To prove: 0.0 ≤ (1 - (E ^ -(**self**.lambda * t)))
Reason: Exhausted rules

- Could not prove:
  (2.71828 ^ -(**self**.lambda * t)) ≤ 1.0

Figure 6.10 PD Report: verification condition over *probf* function not proved

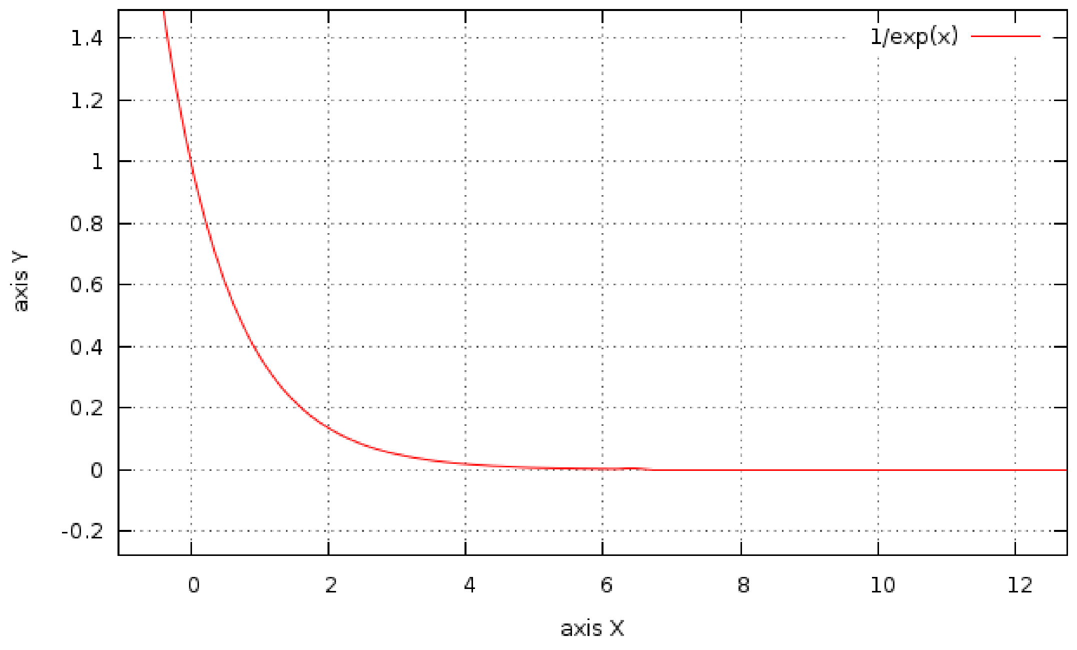
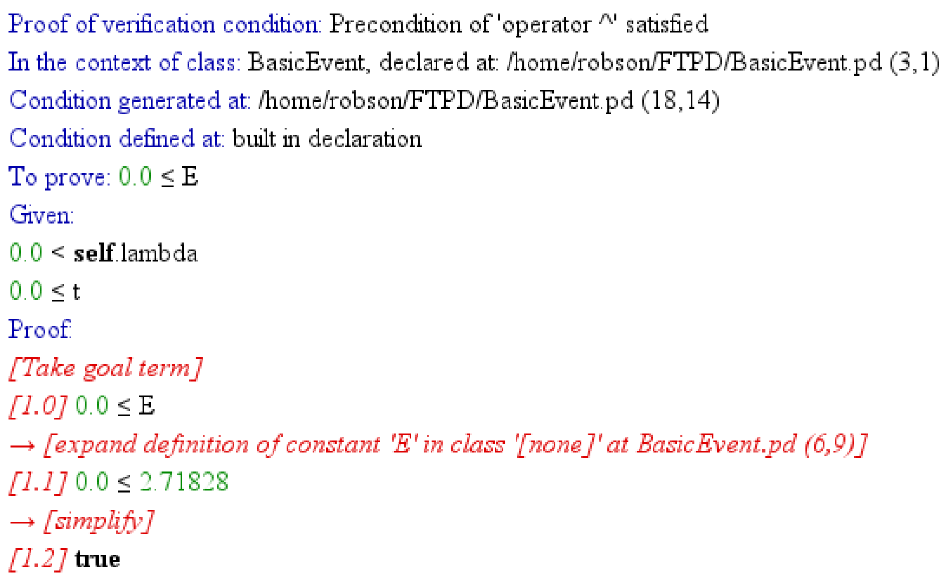
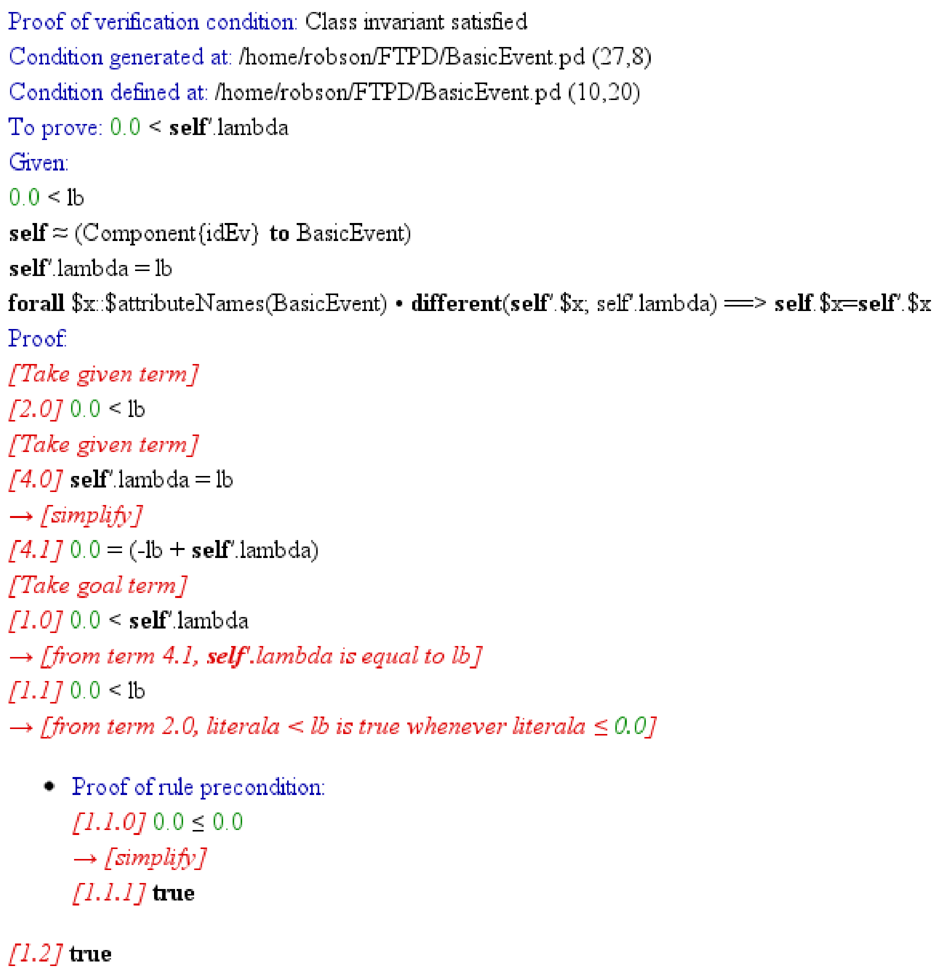


Figure 6.11 Graph of $1/e^x$ function



Proof of verification condition: Precondition of 'operator ^' satisfied
In the context of class: BasicEvent, declared at: /home/robson/FTPD/BasicEvent.pd (3,1)
Condition generated at: /home/robson/FTPD/BasicEvent.pd (18,14)
Condition defined at: built in declaration
To prove: 0.0 ≤ E
Given:
0.0 < **self**.lambda
0.0 ≤ t
Proof:
[Take goal term]
[1.0] 0.0 ≤ E
→ *[expand definition of constant 'E' in class '[none]' at BasicEvent.pd (6,9)]*
[1.1] 0.0 ≤ 2.71828
→ *[simplify]*
[1.2] **true**

Figure 6.12 PD Report: a verification condition over exponentiation *P*L built-in operator



Proof of verification condition: Class invariant satisfied
Condition generated at: /home/robson/FTPD/BasicEvent.pd (27,8)
Condition defined at: /home/robson/FTPD/BasicEvent.pd (10,20)
To prove: 0.0 < **self'**.lambda
Given:
0.0 < lb
self ≈ (Component{idEv} **to** BasicEvent)
self'.lambda = lb
forall $x::$attributeNames(BasicEvent) • **different**(**self'**.$x, self'.lambda) ==> **self**.$x=**self'**.$x
Proof:
[Take given term]
[2.0] 0.0 < lb
[Take given term]
[4.0] **self'**.lambda = lb
→ *[simplify]*
[4.1] 0.0 = (-lb + **self'**.lambda)
[Take goal term]
[1.0] 0.0 < **self'**.lambda
→ *[from term 4.1, **self'**.lambda is equal to lb]*
[1.1] 0.0 < lb
→ *[from term 2.0, literala < lb is true whenever literala ≤ 0.0]*

- Proof of rule precondition:
  [1.1.0] 0.0 ≤ 0.0
  → *[simplify]*
  [1.1.1] **true**

[1.2] **true**

Figure 6.13 PD Report: a verification condition over *BasicEvent*'s constructor

7
Conclusion

This work proposed a methodology for creating GUI-based formally verified tools through the combination of metamodel-based GUI generators (an approach that follows MDE principles) with executable back-ends automatically generated from formal specifications.

As to instantiate our methodology it is necessary to make some design decisions we defined a set of requirements to help Software Engineers to take such decisions.

We presented an instantiation of our methodology using the *Perfect* Language to specify formal specifications, the *Perfect Developer* Tool to prove properties about the specification and generate automatically the back-end. To generate the front-end we used metamodel-based Eclipse modeling tools.

To guarantee that the front-end connects correctly with the back-end, we defined a set of translation rules that derive metamodels and constraints from formal specifications. We developed a tool, called *PL2EMF*, as an Eclipse Plugin by means of the Spoofax Language Workbench. This implementation involves (i) the definition of the complete *P*L syntax using SDF formalism and the implementation in Stratego Language of translation rules to extract automatically from *P*L specifications (ii) metamodels written in EMFatic and (iii) constraints written in EVL.

Based on this instantiation we developed a case study to illustrate our methodology. First, we formalized a simplified version of the Fault Tree model. From this, following our methodology, we created a simplified version of a formally verified GUI-Based Fault Tree Analysis Tool. This tool brings benefits which include: (i) the easiness to build huge Fault Tree models; (ii) validate its structure against constraints derived from formal invariant; and (iii) calculate, from formally verified generated code, the failure probability of the Fault Tree top event.

We investigated the soundness of our translation rules by exercising the GUI checking whether the invariants and preconditions that hold in the FT formal specification, also hold in the GUI created from its metamodel. With respect to completeness, our rules are complete with respect to the *Perfect* language subset we used here. But we need to prove that as well.

7.1 Related works

The work (Gargantini *et al.*, 2009a) presents the benefits of integrating Formal Methods (FM) with MDE software development and discusses how they can be used to (partially or completely) overcome the disadvantages of each solution taken separately. It proposes an approach, called *in-the-loop*, to integrate both worlds, showing its experience on integrating Abstract State Machine (ASM) formal method, used to provide semantics, with EMF. In the *in-the-loop* the application of the MDE to the FM occurs before the application of the FM to the MDE. From the first activity, the FM will be endowed with a set of modeling artifacts which can be used in the second activity to automatize (meta-)model transformations and apply suitable tools for formal analysis of models. However, it is not provided automatic synthesized code from the formal semantics as well as the metamodel is not obtained automatically from a formal specification.

In (Gargantini *et al.*, 2009c), which is already an evolution of the work reported in (Gargantini *et al.*, 2009a), a formal semantic framework is introduced for the definition of the semantics (possibly executable) of metamodel-based languages. Using metamodelling principles, several techniques are proposed to show how the ASM formal method can be integrated with current metamodel engineering environments to endow metamodel-based languages with precise and executable semantics. The use of to semantic framework is exemplified by applying the proposed techniques to the OMG metamodelling framework for the behavior specification of the Finite State Machines provided in terms of a metamodel.

Instead of as (Gargantini *et al.*, 2009c) proposes to provide semantics to a metamodel-based language, e.g. using ASM formal method, in a metamodeling environment, our work proposes a different way: the dynamic semantics is provided by any formal method that satisfy our requirements defined in Chapter 3 (*P*L is used in this work but it could be any other). This dynamic semantics is bound to a metamodel expressed in a metamodel-based language (in this work we uses EMFatic but could be another) by means of a set of transformation rules. In fact, these rules were responsible for generating the metamodel.

The work (Jiang and Wang, 2012) proposes a formal representation of the structural semantics of Domain-specific metamodeling language (DSMML). It works on structural semantics due to (i) its importance and (ii) research on it is not so extensive and deep as behavioral semantic. The structural semantics of DSMML describes static semantic constraints between metamodeling elements, focusing on the static structural properties. It is argued that there are much typical work on formalization of modeling language. That is, (semi-)informal language formalised and verified using some formal methods. But, without considering formalization of metamodeling language and automatic translation from metamodels to the corresponding formal semantic domain. In contrast, our work does not focus on formalizing metamodeling languages, but we guarantee that our metamodels, generated automatically from DSLs formal specifications, conforms to its properties formally verified.

The work (Moller *et al.*, 2008) presents how CSP-OZ can be integrated with UML and Java in the design of distributed reactive system. The advantages of such an integration lies in the rigor of the formal method and in checking adherence of implementations to models. The integration starts by generating a significant part of the CSP-OZ from the UML model. From this specification, properties can be verified. This CSP-OZ specification is also the basis for generating JML contracts (complemented by CSP_{jassda}) for the final implementation. Tools for runtime checking are used to supervise the adherence of the final Java implementation to the generated contracts. Large parts of the integration approach are automated. However, as the development of tools that support the approach were not the focus, they are only prototypes. In contrast to our work that aims to use existing tools and develop new tools to support our proposed methodology as automatic as possible.

The work reported in (Di Ruscio et al., 2006) presents, a practical and generic solution to define the precise dynamic semantics of DSLs by means of an experiment where Abstract State Machines (ASMs) are used to give the dynamic semantics of Session Programming Language (SPL). SPL is a DSL defined for the development of telephony services over the Session Initiation Protocol (SIP). This experiment is performed in the context of a MDE framework called AMMA (Atlas Model Management Architecture). However, unlike our proposed methodology, in this work, there is no focus on formal proofs of properties over the dynamic semantics of a DSL.

7.2 Future Work

As future work we intend to prove the soundness and completeness of the translation rules defined in this work. Another goal is to instantiate our methodology by using different formal methods and MDE approaches and tools. For each instantiation of the methodology, we intend to apply it to build several GUI-Based formal tools for different DSLs. We also wish to test the consistency between front-end and back-end creating and manipulating test-models (instances of a DSL) in this tool.

A controlled experiment to check whether our proposal is much productive than dedicated software engineers creating a corresponding tool and applying a testing campaign to attest the tool's correctness.

We also intend to apply our proposed methodology in real case studies of our industrial partners. We also intend to investigate how our methodology behaves on developing tools for textual DSLs.

References

Bézivin, J. (2005). On the unification power of models. *Software and System Modeling*, **4**(2), 171–188.

Boehm, B. W. (1981). *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, NJ.

Chalin, P. and Rioux, F. (2008). JML runtime assertion checking: Improved error reporting and efficiency using strong validity. In J. Cuéllar, T. S. E. Maibaum, and K. Sere, editors, *FM*, volume 5014 of *Lecture Notes in Computer Science*, pages 246–261. Springer.

Cok, D. and Kiniry, J. (2005). Esc/java2: Uniting esc/java and jml. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer Berlin Heidelberg.

Di Ruscio et al., D. (2006). A practical experiment to give dynamic semantics to a dsl for telephony services development. Technical report, Laboratoire d'Informatique de Nantes-Atlantique (LINA).

Escher (2012). Escher Verification Studio v5.0 (academic license.) eschertech.com. acessed oct/2012.

Esterel Technologies (2012). SCADE Suite Product. http://www.esterel-technologies.com/products/scade-suite/. acessed jan/2012.

EuGENia (2013). EuGENia Annotations Manual. eclipse.org/epsilon/doc/articles/eugenia-gmf-tutorial. Acessed: Jan/2012.

Eysholdt, M. and Behrens, H. (2010). Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 307–309, New York, NY, USA. ACM.

Favre, J.-M. (2004). Towards a Basic Theory to Model Model Driven Engineering.

Fowler, M. (1996). *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Menlo Park, CA.

Gargantini, A., Riccobene, E., and Scandurra, P. (2009a). Integrating formal methods with model-driven engineering. In *Software Engineering Advances, 2009. ICSEA '09. Fourth International Conference on*, pages 86–92.

Gargantini, A., Riccobene, E., and Scandurra, P. (2009b). A semantic framework for metamodel-based languages. *Automated Software Engineering*, **16**(3-4), 415–454.

Gargantini, A., Riccobene, E., and Scandurra, P. (2009c). A semantic framework for metamodel-based languages. *Automated Software Engg.*, **16**(3-4), 415–454.

GMF (2013). Graphical Modeling Framework Project (GMF). http://www.eclipse.org/modeling/gmp/. Acessed: Jan/2013.

Gronback, R. C. (2009). *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, Upper Saddle River, NJ.

Heering, J., Hendriks, P. R. H., Klint, P., and Rekers, J. (1989). The syntax definition formalism sdfreference manual. *SIGPLAN Not.*, **24**(11), 43–75.

Jackson, E. K., Levendovszky, T., and Balasubramanian, D. (2011). Reasoning about metamodeling with formal specifications and automatic proofs. In *MoDELS*, pages 653–667.

Jiang, T. and Wang, X. (2012). Research on metamodels consistency verification based on formalization of domain-specific metamodeling language. *Journal of Shanghai Jiaotong University (Science)*, **17**(2), 171–177.

Kats, L. C. L. and Visser, E. (2010). The Spoofax Language Workbench: rules for declarative specification of languages and IDEs. In W. R. Cook, S. Clarke, and M. C. Rinard, editors, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, pages 444–463, Reno/Tahoe, Nevada. ACM.

Kats, L. C. L., Visser, E., and Wachsmuth, G. (2010). Pure and declarative syntax definition: Paradise lost and regained. In *Proceedings of Onward! 2010*. ACM.

Kleppe, A. G., Warmer, J., and Bast, W. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Kolovos, D., Paige, R., and Polack, F. (2006). The Epsilon Object Language (EOL). In A. Rensink and J. Warmer, editors, *Model Driven Architecture Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 128–142. Springer Berlin Heidelberg.

Kolovos, D., Rose, L., Paige, R., and Polack, F. (2009a). Raising the level of abstraction in the development of GMF-based graphical model editors. In *Modeling in Software Engineering, 2009. MISE '09. ICSE Workshop on*, pages 13 –19.

Kolovos, D., Rose, L., García-Domínguez, A., and Paige, R. (2013). *The Epsilon Book*.

Kolovos, D. S. (2008). *An Extensible Platform for Specification of Integrated Languages for Model Management*. Ph.D. thesis, Department of Computer Science, University of York.

Kolovos, D. S., Paige, R. F., and Polack, F. A. C. (2009b). Rigorous methods for software construction and analysis. chapter On the evolution of OCL for capturing structural constraints in modelling languages, pages 204–218. Springer-Verlag, Berlin, Heidelberg.

Kolovos, D. S., Rose, L. M., Abid, S. B., Paige, R. F., Polack, F. A. C., and Botterweck, G. (2010). Taming EMF and GMF using model transformation. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I*, MODELS'10, pages 211–225, Berlin, Heidelberg. Springer-Verlag.

Leavens, G. T. and Cheon, Y. (2006). Design by contract with JML,.

Maciel, P. R. M., Trivedi, K. S., Jr., R. M., and Kim, D. S. (2011). *Dependability Modeling. Performance and Dependability in Service Computing: Concepts, Techniques and Research Directions*. Hershey, Pennsylvania.

Martin, J. and Odell, J. J. (1995). *Object-Oriented Methods: A Foundation*. Prentice Hall.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Comput. Surv.*, **37**(4), 316–344.

Miller, J. and Mukerji, J. (2003). MDA guide version 1.0.1. Technical report, Object Management Group (OMG).

Moller, M., Olderog, E.-R., Rasch, H., and Wehrheim, H. (2008). Integrating a formal method into a software engineering process with uml and java. *Form. Asp. Comput.*, **20**(2), 161–204.

NASA (2002). *Fault Tree Handbook with Aerospace Applications*. NASA Office of Safety and Mission Assurance, Washington, DC.

OMG (2013). Object constraint language (OCL). `http://www.omg.org/spec/OCL/2.3.1/PDF`. acessed jan/2013.

SAE (1996). Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems. Aerospace Recommended Practice ARP4761, Society of Automotive Engineers (SAE), Warrendale, PA.

Schmidt, D. C. (2006). Guest editor's introduction: Model-Driven Engineering. *Computer*, **39**, 25–31.

SDF (2013). Stratego/XT Manual. Chapter 6. Syntax Definition in SDF. `http://hydra.nixos.org/build/5114850/download/1/manual/chunk-chapter/tutorial-sdf.html`. Acessed: May/2013.

Silva, R. (2013). GUI-based DSL Formal Tools Project. `cin.ufpe.br/~rss7/mscproj/`.

Spoofax (2013). The Spoofax Language Workbench v1.1. `http://spoofax.org/`.. Acessed: Feb/2013.

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2008). *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2. edition.

Stratego (2013). Stratego/XT Manual. Part III. The Stratego Language. `http://hydra.nixos.org/build/5114850/download/1/manual/chunk-chapter/stratego-language.html`. Acessed: May/2013.

Truyen, F. (2006). The Fast Guide to Model Driven Architecture - The basics of Model Driven Architecture.

Voelter, M. and Solomatov, K. (2010). Language modularization and composition with projectional language workbenches illustrated with MPS. In M. van den Brand, B. Malloy, and S. Staab, editors, *Software Language Engineering, Third International Conference, SLE 2010*, Lecture Notes in Computer Science. Springer.

Appendix

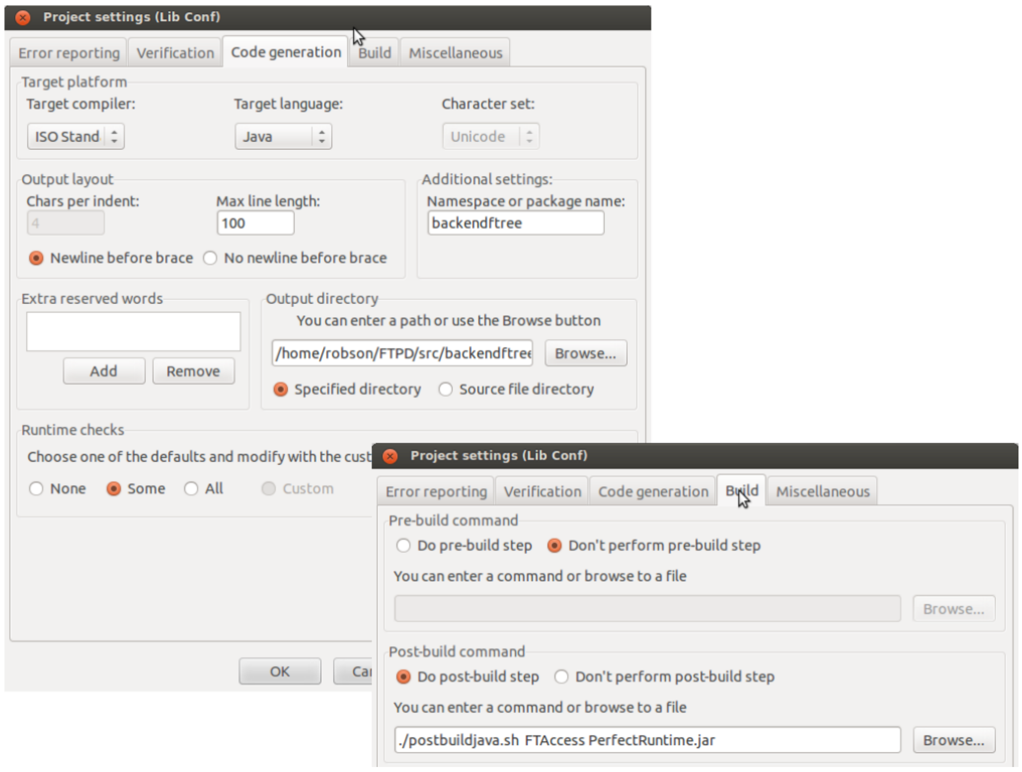
A Fault Tree Formal Specification

A.1 Specification in *Perfect* Language

```
deferred class Component ^=
abstract
  var id : string;
  var desc : string;
interface
  operator = (arg);
  function id, desc;

  build{!id: string}
    post desc! = "";

  build{!id: string, !desc: string};
end;

deferred class Gate ^= inherits Component
interface
  build{idEv: string}
    inherits Component{idEv};

  build{idEv: string, descc: string}
    inherits Component{idEv, descc};
end;

class ANDGate ^= inherits Gate
interface
  build{idEv: string}
    inherits Gate{idEv};

  build{idEv: string, descc: string}
    inherits Gate{idEv, descc};
end;

class ORGate ^= inherits Gate
interface
  build{idEv: string}
```

```
    inherits Gate{idEv};

  build{idEv: string, descc: string}
    inherits Gate{idEv, descc};
end;

class BasicEvent ^= inherits Component
abstract
  //Euler's Number
  const E : real ^= 2.718281828;
  //failure rate
  var lambda : real;

  invariant lambda > 0.0;

interface
  function lambda;

  function probf(t:real):real
  pre t >=0.0
    ^= (1 - E^(-(lambda*t)))
  assert result >= 0.0;

  build{idEv: string, lb: real}
  pre lb > 0.0
    inherits Component{idEv}
  post lambda! = lb;

  build{idEv: string, descc: string, lb: real}
  pre lb > 0.0
    inherits Component{idEv, descc}
  post lambda! = lb;
end;

class FTEdge ^=
abstract
  var src: from Component;
  var tgt: from Gate;
interface
  function src, tgt;
  build{!src: from Component,
!tgt: from Gate};
end;

class FT ^=
abstract

  var ftnodes: set of from Component, ftedges: set of FTEdge;

  invariant
    forall n1::ftnodes :-
~(exists n2::ftnodes.remove(n1) :- n1.id = n2.id),
    #(those n::ftnodes :- n.id = "TOP_EVENT") <= 1,
    forall g::ftnodes :- g.id = "TOP_EVENT" ==> (g within from Gate),
    forall e::ftedges :- (exists n::ftnodes :- n.id = e.src.id),
    forall e::ftedges :- (exists n::ftnodes :- n.id = e.tgt.id),
    ~(exists e::ftedges :- e.src.id = e.tgt.id),
    forall f::ftedges :-
  ~(exists e::ftedges.remove(f) :- f.src.id = e.src.id );

    //more invariants can be added here
interface
  ghost operator =(arg);

  function checkFTNode (n: from Component): bool
  ^= exists c::ftnodes :- c.id = n.id;

  function srcCompNotUsed (e: FTEdge): bool
  ^= ~(exists f::ftedges :- f.src.id = e.src.id);

  function existsTOPEventId:bool
  ^= exists c::ftnodes :- c.id = "TOP_EVENT";

  schema !addFTNode (n: from Component)
  pre
      //the new node is not known
      ~checkFTNode(n),
      //there must exists only one node called
      //"TOP_EVENT" and it must be a gate
      (n.id = "TOP_EVENT" ==> ~existsTOPEventId &
      (n within from Gate))
  post
      //add the new node in the set of ftnodes
      (ftnodes! = ftnodes.append(n));

  schema !addFTEdge(e: FTEdge)
  pre
      //the elements of the edge must be known
      checkFTNode(e.src), checkFTNode(e.tgt),
      //no cicle to itself
      e.src.id ~= e.tgt.id,
      //components cannot be shared
      srcCompNotUsed(e)
  post
      (ftedges! = ftedges.append(e));

  function getFTFailProb(t:real):real
  pre t >= 0.0
  ^= ( let topevs ^= those n::ftnodes :- n.id = "TOP_EVENT";
       ([#topevs = 1] : compute(that topevs, t, #ftnodes),
       []: -1.0)//indicates that there is top event
     );

  function compute(n: from Component, t:real, h: int) : real
  pre  t >= 0.0
  decrease h
  satisfy result >= 0.0
  via
```

```
    let childs ^= (for those e::ftedges :-
    e.tgt.id = n.id yield e.src).opermndec;
    if
       [n within ANDGate]:
var tot: real != 1.0;
loop
var j: nat != 0;
change tot
keep tot' >= 0.0 & j' >= 0
until j' = #childs
decrease #childs - j';
     tot! * compute(childs[j], t, h-1);
     j!+1;
end;
value tot;

       [n within ORGate]:
var tot: real != 0.0;
loop
var j: nat != 0;
change tot
keep tot' >= 0.0 & j' >= 0
until j' = #childs
decrease #childs - j';
     tot! + compute(childs[j], t, h-1);
     j!+1;
end;
value tot;

[n within BasicEvent]:
  value (n is BasicEvent).probf(t);

        []:value 0.0; //unreachable

     fi;
   end;

  schema !clearAll
    post
ftnodes! = set of from Component{},
ftedges! = set of FTEdge{};

  build{}
    post
ftnodes! = set of from Component{},
ftedges! = set of FTEdge{};
end;


class FTResultCode ^=
enum
  NODE_ADDED,
  NODE_ALREADY_EXISTS,
  TOP_EVENT_ALREADY_EXISTS,
  TOP_EVENT_MUST_BE_GATE,
  SRC_NODE_DOES_NOT_EXISTS,
  TGT_NODE_DOES_NOT_EXISTS,
  EDGE_SRC_EQUALS_TO_TGT,
  EDGE_ALREADY_EXISTS,
  EDGE_TGT_IS_NOT_A_GATE,
  EDGE_ADDED
end;

class FTWrapper ^=
abstract
  var ft: FT;

interface
  ghost operator =(arg);

  build{}
    post ft! = FT{};

  schema !clearAll
    post ft!clearAll;

  schema !addFTComp(c: from Component, rslt!: out FTResultCode)
    post (
  //node already in FT so don't try to add it
  [ft.checkFTNode(c)]: rslt! = FTResultCode NODE_ALREADY_EXISTS,
  //an event identified as top event has already been added
  [c.id = "TOP_EVENT" & ft.existsTOPEventId]:
      rslt! = FTResultCode TOP_EVENT_ALREADY_EXISTS,
  //the top event must be a gate. it can not be a basic event
  [c.id = "TOP_EVENT" &  ~(c within from Gate)]:
      rslt! = FTResultCode TOP_EVENT_MUST_BE_GATE,
  //otherwise, the node is added successfuly
  []: ft!addFTNode(c) & rslt! = FTResultCode NODE_ADDED
);

  schema !addFTEdge(e: FTEdge, rslt!: out FTResultCode)
    post (
  [~ft.checkFTNode(e.src)]: rslt! = FTResultCode SRC_NODE_DOES_NOT_EXISTS,
  [~ft.checkFTNode(e.tgt)]: rslt! = FTResultCode TGT_NODE_DOES_NOT_EXISTS,
  [e.src.id = e.tgt.id]: rslt! = FTResultCode EDGE_SRC_EQUALS_TO_TGT,
  [~ft.srcCompNotUsed(e)]: rslt! = FTResultCode EDGE_ALREADY_EXISTS,
  []: ft!addFTEdge(e) & rslt! = FTResultCode EDGE_ADDED
);


  function getFTFailProb(t:real):real
  pre t > 0.0
    ^= ft.getFTFailProb(t);

end;
```



Project settings (Lib Conf)

Error reporting | Verification | Code generation | Build | Miscellaneous

Target platform
Target compiler: ISO Stand
Target language: Java
Character set: Unicode

Output layout
Chars per indent: 4
Max line length: 100
[✓ Newline before brace] No newline before brace

Additional settings:
Namespace or package name: backendftree

Extra reserved words
Add Remove

Output directory
You can enter a path or use the Browse button
/home/robson/FTPD/src/backendftree Browse...
[✓ Specified directory] Source file directory

Runtime checks
Choose one of the defaults and modify with the cust
None [✓ Some] All Custom

OK Ca

Project settings (Lib Conf)

Error reporting | Verification | Code generation | Build | Miscellaneous

Pre-build command
Do pre-build step [✓ Don't perform pre-build step]
You can enter a command or browse to a file
Browse...

Post-build command
[✓ Do post-build step] Don't perform post-build step
You can enter a command or browse to a file
./postbuildjava.sh FTAccess PerfectRuntime.jar Browse...

A.2 Back-end generation parameters

In Figure A.2 we show the parameters needed to be setted in *Perfect* Developer v5.0 to instruct it to generate the the back-end in Java Language as a .jar library.

B Fault Tree Metamodel and Constraints

B.1 Metamodel

```
@namespace(uri="ftree", prefix="ftree")
package ftree;

@gmf.diagram(rcp = "true")
class FT {
   val Component[*] ftnodes;
   val FTEdge[*] ftedges;
}

@gmf.link(source="src", target="tgt", target.decoration="arrow")
class FTEdge {
  ref Component src;
  ref Gate tgt;
}

abstract class Component {
  attr String idc;
  attr String desc;
}

@gmf.node(label = "desc",
  figure = "ellipse")
class BasicEvent extends Component {
attr double lamba;
}

abstract class Gate extends Component {
}

@gmf.node(label = "desc",
  figure="figures.ANDGateFigure",
  label.icon="false",
  size="15,15")
class ANDGate extends Gate {
}
```

```
@gmf.node(label = "desc",
  figure="figures.ORGateFigure",
  label.icon="false",
  size="15,15")
class ORGate extends Gate {
}
```

B.2 Constraints

```
context FT {
constraint inv1 {
check: self.ftnodes.forAll(n1 |
not self.ftnodes.excluding(n1).exists(n2 | n1.idc = n2.idc))
message {
var msg = "#inv1 violated: The identifiers of the components must be unique\n";
System.user.inform(msg);
return msg;
}
}

    constraint inv2 {
check: self.ftnodes.select(n | n.idc = "TOP_EVENT").size() <= 1
message {
var msg = "#inv2 violated: Must exists at most one Top Event\n";
System.user.inform(msg);
return msg;
}
}

constraint inv3 {
check : self.ftnodes.forAll(g | g.idc = "TOP_EVENT" implies g.isKindOf(Gate))
message {
var msg = "#inv3 violated: The Top Event must be a gate\n";
System.user.inform(msg);
return msg;
}
}

constraint inv4 {
check : self.ftedges.forAll(e | self.ftnodes.exists(n | n.idc = e.src.idc))
message {
var msg = "#inv4 violated: The source is must be a known node\n";
System.user.inform(msg);
return msg;
}
}

constraint inv5 {
check : self.ftedges.forAll(e | self.ftnodes.exists(n | n.idc = e.tgt.idc))
message {
var msg = "#inv5 violated: The target is must be a known node\n";
System.user.inform(msg);
return msg;
}
}

constraint inv7 {
check : not self.ftedges.exists(e | e.src.idc = e.tgt.idc)
message {
var msg = "#inv2 violated: Source and  target of an edge must be different\n";
System.user.inform(msg);
return msg;
}
}

constraint inv8 {
check: self.ftedges.forAll(f |
not self.ftedges.excluding(f).exists(e | f.src.idc = e.src.idc))
message {
var msg = "#inv4 violated: Source component must have only one target component\n";
System.user.inform(msg);
return msg;
}
}

}
```

C
Fault Tree Link

C.1 Wizard for calculanting FT Failure Probability

```
wizard CalculateTEFP {
 guard : self.isKindOf(FT)
 title : "Calculate Top Event Failure Probability"
 do {
  var adapter = new Native("br.ufpe.cin.ftree.link.AdapterFT");
  for (n in self.ftnodes) {
    if (n.isKindOf(ORGate))
    adapter.addORGate(n.idc, n.desc);
    else
    if (n.isKindOf(ANDGate))
    adapter.addANDGate(n.idc, n.desc);
    else
    if (n.isKindOf(BasicEvent))
    adapter.addBasicEvent(n.idc, n.desc, n.lamba);
  }

  for (e in self.ftedges)
   adapter.addFTEdge(e.src.idc, e.tgt.idc);
  var t = System.user.promptReal("Calculate Top Event Failure Probability\n"+
                  "Please enter the parameter t (time)");

  var r = adapter.getFTFailProb(t);

  System.user.inform("F (t = " + t + " ): " + r);
 }
}
```

C.2 AdapterFT.java

This adapter is ....

```
package br.ufpe.cin.ftree.link;
```

```
import java.util.HashMap;
import java.util.Map;

import Ertsys._eSystem;
import backendftree.ANDGate;
import backendftree.BasicEvent;
import backendftree.Component;
import backendftree.FTEdge;
import backendftree.FTResultCode;
import backendftree.FTWrapper;
import backendftree.Gate;
import backendftree.ORGate;

public class AdapterFT {
FTWrapper backend;
Map<String, Component> cMap;

public AdapterFT() {
backend = new FTWrapper();
cMap = new HashMap<String, Component>();
}

public void addFTComp (Component c) {
FTResultCode rslt = new FTResultCode();
backend.addFTComp(c, rslt);
//check status report of the back-end: rslt.
}

public void addFTEdge(String idSrc, String idTgt) {
FTResultCode rslt = new FTResultCode();
FTEdge e = new FTEdge(cMap.get(idSrc), (Gate)cMap.get(idTgt));
backend.addFTEdge(e, rslt);
}

public double getFTFailProb(double t) {
double r = backend.getFTFailProb(t);
backend.clearAll();
cMap.clear();
return r;
}

public void addBasicEvent (String id, String desc, double lambda) {
Component c = new BasicEvent(_eSystem._lString(id), (char) 0,
                _eSystem._lString(desc), (char) 0, lambda);
addFTComp(c);
cMap.put(id, c);
}

public void addANDGate (String id, String desc) {
Component c = new ANDGate(_eSystem._lString(id), (char) 0,
                _eSystem._lString(desc), (char) 0);
addFTComp(c);
cMap.put(id, c);
}

public void addORGate (String id, String desc) {
Component c = new ORGate(_eSystem._lString(id), (char) 0,
           _eSystem._lString(desc), (char) 0);
addFTComp(c);
cMap.put(id, c);
}
}
```