Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Jan Calta

Analyzer of Windows Kernel Models

Department of Software Engineering
Advisor: Mgr. Pavel Ježek
Study Program: Computer Science, Software Systems

I would like to thank František Plášil, Jiří Adámek and Jan Kofroň who introduced me to the software verification research area. I thank my advisor Pavel Ježek and especially Tomáš Matoušek for a discussion on issues related to the thesis.

I declare that I have elaborated this master thesis on my own and listed all used references. I agree with lending of this master thesis. The thesis may be reproduced for academic purposes.

In Prague on December 13th, 2007 Jan Calta

Table of Contents

Title: *Analyzer of Windows Kernel Models*
Author: Jan Calta
Department: Department of Software Engineering
Advisor: Mgr. Pavel Ježek
Advisor's e-mail address: jezek@nenya.ms.mff.cuni.cz
Abstract:

The thesis introduces a tool for analyzing models written in the specification language DeSpec and translating them into the Zing modeling language. Resulting models can be verified by the Zing model checker. The DeSpec language is designed primarily to specify the Windows NT kernel driver environment. It makes it possible to abstract this environment in the object-oriented way and it uses temporal logic patterns to capture rules imposed by the Windows kernel on drivers. The Zing language is designed to describe executable concurrent models of software, which can be explored by the Zing model checker. Properties to check are expressed by the assertions. So far, there has been no way to automatically extract a model from DeSpec specification and verify its properties by a model checker. The DeSpec-to-Zing compiler takes a crucial part in this task.

The thesis demonstrates that it is feasible to translate DeSpec specifications into Zing models and that DeSpec is a suitable language for model checking of the Windows kernel driver environment. The introduced analyzer is capable to check correctness of DeSpec specifications and under the constrained conditions given by absence of other necessary tools it is capable to translate a subset of specifications into the Zing model.

Keywords: Windows drivers, compilers, Zing, model checking, software verification

Název práce: Analyzátor modelů jádra OS Windows
Autor: Tomáš Matoušek
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: Mgr. Pavel Ježek
e-mail vedoucího: jezek@nenya.ms.mff.cuni.cz
Abstrakt:

Diplomová práce předkládá nástroj pro analýzu modelů ve specifikačním jazyce DeSpec a pro jejich překlad do modelovacího jazyka Zing. Výsledné modely pak mohou být verifikovány model checkerem Zing. Jazyk DeSpec je navržen především pro specifikaci prostředí, ve kterém pracují ovladače operačních systémů rodiny Windows NT. Umožňuje abstrahovat toto prostředí objektově orientovaným způsobem a používá formule lineární temporální logiky k popisu pravidel, jejichž splnění jádro OS Windows od ovladačů vyžaduje. Jazyk Zing je navržen pro popis vykonavatelných modelů software včetně paralelismu, které mohou být dale zkoumány model checkerem Zing. Vlastnosti k ověření jsou vyjádřeny příkazy assert. Dosud neexistoval způsob, jak automaticky extrahovat ze specifikace v DeSpecu model, který by mohl být formálně verifikován model checkerem. Překladač z DeSpecu do Zingu hraje v tomto úkolu zásadní roli.

Práce ukazuje, že je možné překládat specifikace v DeSpecu do modelů v Zingu a tedy že DeSpec je vhodným jazykem pro model checking cílového prostředí. Uvedený nástroj umožňuje kontrolu správnosti specifikace v DeSpecu a za omezení daných absencí dalších nezbytných nástrojů umožňuje překlad vybrané podmnožiny specifikací do Zingu.

Klíčová slova: ovladače Windows, překladače, Zing, model checking, verifikace software

1. Introduction

Verification of correctness and expected properties is one of the key tasks in software development. It becomes even more apparent while writing concurrent programs, where concurrency is often the source of bugs, which are hard to find and debug. As this problem is crucial for most of industrial and heavily used software, significant effort to find a suitable solution has been made. Stress-testing cannot entirely eliminate this issue and in some applications it is necessary to combine it or replace it with a formal approach. One of the techniques, which has proved to be suitable for this task, is model checking.

1.1. Model Checking

Model checking is the most successful approach that has emerged for verifying requirements. The idea of model checking is as follows: A model of the analyzed environment is made. The requirement imposed on the environment is formulated. A model-checking tool (i.e. model checker) accepts the model and the requirement that the final system is expected to satisfy. After verification, the model checker outputs yes if the given model satisfies given requirements (and verification passed) and generates a counterexample otherwise (verification failed).

The counterexample details why the model does not satisfy the requirement. It is usually demonstrated by an execution path breaking the requirement. Once all errors are discovered and fixed and verification passes, one can be confident about the correctness of the model in all its reachable states. In fact, this ideal state is not always reached because of undecidability (model checker is unable to verify the property in finite time), however, the results of verification are useful even if some of the bugs are not discovered.

The main drawback of this technique is a so-called *state explosion problem*. It refers to an exponentially growing number of model's states with each added parameter. Thus verification becomes more resource-demanding and often even impossible to finish in acceptable time. This problem can be solved by modeling the system on a higher level of abstraction and by providing necessary resources.

1.2. Verification of Windows Drivers

The kernel of Windows NT operating system is more than suitable subject of formal verification. It is very complex and heavily used software and correct interfacing with drivers is the crucial property of the whole kernel-driver environment.

There is a set of rules and guidelines dealing with interaction of drivers and kernel published in Windows Driver Kit (WDK) [1] available for driver authors. However, the rules are described in plain English and a driver developer has no implicit assistance in following them.

Verification of this system requires solution of two issues. Firstly, the model of the environment must be extracted from the system specification and driver and kernel sources. Extraction must focus on problems emerging from the fact that the system is written in the low-level C language and it is difficult to create an appropriate abstraction. Secondly, rules defined in WDK in plain English must be transformed into some form of temporal logic formulae to allow them to be verified by a model checker.

1.3. Modeling Windows Driver Environment

Based on the motivation described above, the specification language DeSpec [2] was designed to make the verification of Windows driver environment possible. This language introduces the object-oriented approach to description of Windows kernel and allows creating models on various levels of abstraction. It supports describing requirements and rules in a form of linear temporal logic (LTL) formulae, more precisely by Temporal Logic Patterns derived from Bandera project [3]. Along with language itself, a specification of Windows driver environment was published in [2]

DeSpec was designed with Zing [4] as target model checker in mind. Zing model checker accepts models written in the Zing modeling language [5], which is an object-oriented language for modeling concurrent software and supports basic level of abstractions like classes, non-deterministic choices, threads and arrays. However the requirements imposed on the verified system can be expressed only by assertions.

With a DeSpec specification of the system to verify on the one hand and Zing model checker on the other, it is necessary to extract the model from the specification and translate it to the Zing language. The resulting model can be verified by the Zing model checker.

1.4. The Thesis

The goal of this thesis is to provide the missing link between DeSpec specification and Zing model checker to enable the formal verification of Windows driver environment. The complete extraction of Zing models of the driver environment from DeSpec specification may require several tools dealing with various issues. Thus the tasks that are crucial for the extraction process must be defined. Above all, the introduced tool should be capable of correct translation of the DeSpec language to Zing. The key features of DeSpec, which are not supported by Zing, must be implemented, particularly those, which allow expressing the requirements to verify.

The following text firstly introduces both the source and target language and analyses the process of the model extraction and translation.

Then the structure and required features of the model extractor are described in chapter 3. In this chapter, all tasks necessary for creating the complete model are mentioned and the individual phases of the extraction are determined. At first, DeSpec specification must be parsed and its inner representation must be built. Then the semantic analysis takes place. It is also necessary to analyze the C sources of the driver and the kernel and complete the model with parts extracted from this input. Another important task is reduction of the model to include only parts relevant to the verified properties. When the model is complete, its transformation to Zing representation takes place. Finally, Zing code is produced.

In chapter 4, the basic analysis of possible approaches to the implementation is made. The development platform and tools are chosen and basic design decisions for the compiler and Zing representation are made.

The thesis continues by the detailed description of the implementation of the compiler in chapter 5. For every phase and task in the process of the model extraction, the key issues are identified and their solution is described. Special effort is made to explain the type analysis, the implementation of DeSpec-specific features and the representation of requirements to verify.

In chapter 6, missing tools, which are required for fully automatic extraction of the complete model, are mentioned. Also unimplemented features of the compiler itself are described there.

Finally, the related work is mentioned and the results of the thesis are summarized.

Appendix A contains the modified grammar of the DeSpec language in the version that is used by the compiler.

Appendix B contains a simplified example of DeSpec class specification and the corresponding Zing model generated by the compiler.

The thesis is accompanied with a CD. The source code of the compiler is stored there along with the executable file. A sample DeSpec specification of the driver environment, which is derived from the specifications published in [2], is also available, together with the corresponding generated Zing model and its executable version, which can be run in Zing model checker.

The CD includes whole package with the compiler, Zing compiler and Zing model checker, so it is possible to run accompanied scripts to create the model from the provided specification and verify it by the model checker. The instructions for running the scripts are included in *readme.html* file, along with the structure of the source code and whole package. Since the source code of the compiler contains also files generated by other tools, the origin of all included source files is explicitly stated there.

2. Verification of Driver Environment Model

Under the given conditions, i.e. with DeSpec specification of the Windows driver environment and Zing as a target model checker, the process of verification of the system comprises several complex steps. This chapter contains short introduction to the DeSpec language and Zing framework and later on, the transformation of the model is described. Finally, the role of the introduced tool in this task is explained.

2.1. DeSpec Language

The DeSpec specification language was designed primarily to describe the Windows driver environment. It supports an object-oriented approach despite the fact that both Windows kernel and drivers are written in the C language. This approach is well founded because the environment simulates an object-oriented design on a specific level, which is limited by means of the C language. With constructs like namespaces, classes, properties and groups and with the support of inheritance, extension and built-in collection types, DeSpec allows writing models of environments straightforwardly and transparently.

DeSpec does not focus only on the high-level abstraction but also allows modeling of threads and concurrency and features constructs inspired by Windows kernel specific concepts and required by the use of the C language. It supports *delegates* for modeling of *ILateBoundDriverRoutines* interface and *function pointer mapping* for modeling of *IEarlyBoundDriverRoutines* interface. These interfaces are designed for passing the driver callbacks to the kernel.

Thanks to *namespaces* and *attributes*, DeSpec allows specifying models on various levels of abstraction and determination of the part of the model that will be subject to verification. Thus it is possible to adjust complexity of the model to make its verification feasible in acceptable time.

The key feature of DeSpec is its extensive support for expressing requirements imposed on the system. There are three concepts for the description of these requirements – *assertions*, *constraints* and *rules*.

Assertion is the most primitive way to describe a required property and can be used as it is usual in common programming languages.

Constraints can be used to assure that some condition holds in interesting places during execution of the model, typically when entering or leaving a method or accessing a field. Some of the constraints backed by DeSpec grammar include for example non-nullity of method arguments and class fields, range constraint or method precondition and postcondition.

Rules are the key means of expressing the requirements on the model and play a crucial role in model checking. Rules allow the use of temporal logic and thus enable the formulation of required properties that cannot be expressed otherwise. Moreover, DeSpec does not require specification of rules directly in LTL but introduces *rule patterns* based on the Bandera toolset. These patterns were designed to describe properties in a well-understood but still precise way. The set of patterns included in DeSpec is universal enough to describe probably all the rules that could be potentially imposed in the Windows driver environment. However, if there is a need to express some property by an unsupported pattern, it will be easy to add it in DeSpec. One of the rule patterns that is often used to

describe requirements is for example *P exists between Q and R*. The same property expressed by LTL formula is $\Box (Q \wedge \neg R \Rightarrow (\neg R \ W \ (P \wedge \neg R)))$[1]. It is obvious that rule patterns are more appropriate to describe the requirements, especially when expressivity of raw LTL is not needed.

Interesting constructs of DeSpec will be mentioned later in sections describing their analysis and implementation in Zing.

2.2. Zing

The Zing framework is developed in Microsoft Research and is divided into 4 parts: a modeling language for expressing concurrent models of software systems, a compiler for translating a zing model into an executable representation of its transition relation, a model checker for exploring the state space of the zing model, and model generators that automatically extract Zing models from programs written in common .NET programming languages.

All components except extractors from .NET languages are important for the process of verification. The Zing model checker has 2 interfaces - a command-line tool (Zinger) and an application with GUI that allows inspecting states of the model (Viewer). The Zing framework in its current state of development supports all features necessary for implementation of DeSpec models and is usable for their formal verification.

The fact that some constructs of the DeSpec language are taken over from the Zing language implies that the implementation of some DeSpec features is quite straightforward. On the other hand, Zing is also an object-oriented language but it lacks some of the typical features, namely inheritance and constructors. Other complications emerge when implementing for example DeSpec delegates, thread static data, structures or built-in collections as Zing does not support these concepts. However, the current version of Zing allows all workarounds necessary for the implementation of DeSpec models.

In the Zing language, types may be either *simple* or *complex*, the primary difference being that complex types are allocated on the heap and simple types are not. Simple types contain enumerations, ranges, structures and the same predefined types as C# except *char* and *string*. Simple type *string* could be useful for implementing internals of models, however it can be replaced by enumerations. Ranges and structures are not fully supported. Complex types include arrays whose sizes can be fixed at the time of allocation, classes and *object* type. An *object* type may be used in place of a strongly-typed declaration. Any complex type reference may be assigned to a variable of the type *object*. Zing does not support typecasts, but an *object* value may be assigned to a strongly-typed variable, which results in a typecast to the target type (if possible). This is used to implement poor polymorphism and DeSpec *is* and *as* operators.

Zing also supports asynchronous calls and synchronization via blocking *select* statement. Statements of different threads can be arbitrarily interleaved unless they are enclosed in an atomic block.

Correct models of the Windows driver environment require some form of non-determinism, which can be achieved by *select* and *choose* statements. Non-determinism

[1] W denoting weak until operator, defined by strong until operator e.g. by equivalence $p \ W \ q = (\Box p) \vee (p \ U \ q)$. $\Box$ is universal time quantifier (always in the future), $\Diamond$ is existential time quantifier (sometime in the future).

leads usually to exponential grow of the explored state space. Sometimes it is useful to reduce the state space by cutting off a trace under a certain condition. The *assume* statement is used for this purpose. It cuts off the current trace if the specified condition does not hold. The *assert* statement can be used to ensure that specified properties hold. During verification of a model by Zing, the model checker failed assumptions are marked but not reported in contrast to failed assertions that cause the failure of the whole verification.

The *assertions* are the only constructs for expressing requirements on the model in Zing. This means that they must be used to implement rule patterns supported by DeSpec. As *assert* statement accepts only common boolean expressions, a workaround implementing features of Linear Temporal Logic must be introduced.

2.3. Extraction of Model

The technologies described in previous chapters satisfy requirements for the verification of the Windows driver environment and DeSpec is even designed directly for this task. However, the transformation of the model to check is still quite complex. The tool that transforms DeSpec specification into the Zing model has to deal above all with the following issues:

First of all, parsing and semantic analysis of DeSpec code must be made. In this task, an extensive support for syntactic sugar in DeSpec must be taken into consideration. This language features many constructs, which are designed to write the specification straightforwardly and on a high level of abstraction. Particular attention is paid to semantic analysis, as it is not possible to restrict the use of advanced constructs on the level of grammar and syntax analysis. This issue is particularly apparent in constructs designed for formulating constraints and rules. Another feature that requires special attention is *attributes*. There are several types of attributes with various meanings and some of them are related to a driver or kernel source code. Thus it is not possible to restrict their formulations by DeSpec grammar and their potential incorrectness has to be checked on the semantic level.

When DeSpec specification is analyzed and its inner representation is made, the remaining inputs must be accepted. For a complete model of the driver environment information about the kernel and the driver is needed. Symbols that are used in DeSpec specification must be extracted from kernel header files, e.g. values of enumerations and constants abstracted by the model. The more complicated task is an analysis of driver source files. The driver part of the model has to include C code of the driver itself. DeSpec does not require inserting appropriate segments of the code into the specification manually. It rather introduces constructs, such as *EarlyBound* attribute and *extracted* label that allows merging of specification and driver code by extractor without the help of a user. It is obvious that quite a complex tool for parsing C code and extracting necessary parts of the model is needed.

Another important task is slicing of the model. Even if an extensive and detailed specification of the whole environment can be provided, the complexity of the resulting Zing model and the verification depends mainly on checked properties and a selected level

of abstraction. To enable this flexibility, it is necessary to analyze which parts of the model are relevant for the particular verification process. DeSpec also provides means for influencing this analysis by the user. It is possible to enforce inclusion of specific constraints into the model by *CheckConstraints* attribute. Before the extraction process starts, the user is supposed to select a specific namespace to check and thus a specific level of abstraction can be chosen. The contents of the resulting model influence the extraction of C source files. It is also possible that there are some parts of the extracted model that are never used by the specific driver so the slicing should be applied again at the end of the model extraction to further reduce its state space. The required analysis can be made by performing a slicing algorithm described in [6].

After a particular part of the model is determined and analyzed, it must be transformed into Zing model, i.e. the inner representation of the model must be translated into Zing language. Despite of the fact that DeSpec uses number of constructs taken over form Zing, there are key features specific to DeSpec that cannot be implemented in Zing in a trivial way.

The concept of inheritance is not supported by Zing but use of this relationship in DeSpec specification of Windows driver environment is well-founded. Windows kernel simulates kind of polymorphism by overlapping structures and there must be a possibility to abstract this behavior in the specification. However, not all features provided by inheritance are simulated by the kernel so it is sufficient to provide only specific necessary workarounds for transformation to Zing. Implementation of polymorphism relies on implicit typecasts made by Zing runtime when assigning an *object* value to strong-typed variable.

Other abstractions useful for specifying the driver environment are DeSpec delegates (analogous to function pointer types in C) that have no counterpart in Zing language. The model of delegates expects a pointer-to analysis to determine a set of functions, which the specified function pointer can point to. In the context of kernel-driver environment it means that driver source code must be analyzed to find out which functions can be bound to a particular delegate. For every DeSpec delegate its value from the set of applicable functions is tracked throughout the model and invocations of delegate are replaced with dispatch to an appropriate function.

A crucial task is representation of temporal logic formulae supported by DeSpec in Zing language that support only assertions. Expressing requirements via temporal logic patterns is one of the main features of DeSpec and makes verification of the system comfortable for users. Under these conditions, a suitable Zing representation of automata equivalent to specified rules must be found and a mechanism for their transitions must be implemented.

To cope with the issues described above a following set of tools must be developed:

At first a tool for parsing and analyzing specifications in DeSpec language should be produced. Next, an extractor of the model from the analyzed specification, kernel header files and driver source code is needed. On this extracted model a slicing algorithm should be performed to determine the part of the model to check. To complete the task, a translator of the model into Zing language is necessary.

It is apparent that the key component is a compiler with DeSpec front-end and Zing back-end. Operations implemented by C code extractor and slicer must be preformed on inner representation of the model after the semantic analysis.

2.4. Tasks for Compiler

During the development it proved to be infeasible to implement whole set of necessary tools in the scope of this thesis. Rather, it was decided to focus on the semantic analysis of DeSpec, which is needed for all the other steps. Some tasks even seem to be suitable as a topic for another master thesis.

However, there are certain subsets of DeSpec language that allows describing a specification, which does not require the other tools to finish the extraction. Only analyzer of DeSpec and translator to Zing can perform complete transformation of such model. With some workarounds and help of user it is possible to create such specification, analyze it with the implemented tool and translate it into Zing model. Analysis of C source files as well as basic model has to be made manually. With this motivation in mind, an attempt to implement a simplified Zing back-end and produce a working compiler was made.

Main part of the introduced compiler is focused on the front-end and DeSpec analysis. The back-end translating DeSpec model into Zing language rather proves that the proposed approach to modeling Windows driver environment is feasible and that DeSpec can be successfully used for model-checking of such systems. However, implementation of the other tools is required to process a full-fledged verification of the environment.

3. Structure of Model Extractor

For extracting a Zing model from DeSpec specification of Windows driver environment, Windows kernel header files and driver source code, 4 main tasks must be solved: Semantic analysis of DeSpec code to ensure its correctness, analysis and extraction of relevant C code to complete the model, slicing to reduce the state space of the model and translation of the model into Zing language. Structure of the model extractor is determined by these 4 steps. In following sections organization of tasks and development is described.

3.1. DeSpec Front-End

For processing of DeSpec specification the typical compiler approach [7] is applied. The analysis is divided into 3 levels: lexical, syntax and semantic.

3.1.1. Lexical Analysis

The analysis of tokens is made by a lexer generated from DeSpec lexical grammar. There is no need to implement the lexer by hand as lexer generators are available. However, the lexer generator should be chosen with respect to target language of the compiler implementation.

3.1.2. Syntax Analysis

The syntax analysis is made by a parser that is generated from DeSpec syntactic grammar. Same as for lexer, there are tools for generating parsers available. A chosen generator should produce parsers that are able to interface with the generated lexer.

During the process of parsing, an inner representation of the specified model is built. It has a form of *Abstract Syntax Tree* (AST) [8]. The structure of AST nodes is designed in such a way that there is no need to transform the generated AST to another intermediate form during the processing of a model. All necessary operations on the model can be performed easily through the generated AST.

Changes in DeSpec, which turn out to be desirable during the development of extractor, are reflected in its syntax grammar.

3.1.3. Semantic Analysis

During semantic analysis, the compiler has to check the semantic rules for using DeSpec constructs as described in [2]. This also requires a complete static type analysis. Whole task is completed by several passages through the AST. During traversing the AST, operations necessary to support further translation into Zing are performed.

3.1.4. Built-In Types

DeSpec specification of built-in types is actually a specification of templates, which cannot be used directly in the model. Instead, parameterized instances of these generic templates should be created. It is necessary to find all references to various built-in types and supply the model with specification of instances of required templates. This must be completed before type analysis to enable the mapping of built-in type references to their declarations.

3.1.5. Eliminating Compile-Time Constructs

DeSpec features several constructs that are designed to make specifications more readable and make ideas behind them clear, e.g. *groups*, *namespaces* or *extension* mechanism. All of them must be eliminated during the extraction of the model and their meaning must be represented in a different way. Complex semantic of these features implies that their implementation by other means is not trivial.

3.2. Kernel and Driver Code Analysis

A specification of Windows driver environment must be merged with C code of the Windows kernel and a driver to be verified. Kernel header files must be inspected for extracting symbols referenced by the model. For example abstractions of kernel enumeration and constants must be supplied with appropriate values. As for the driver part, extraction is more complicated and requires e.g. a pointer-to analysis of function pointer types and merging of method abstractions in DeSpec with bodies of driver functions in C. Complete source code of a verified driver is required to complete the model.

3.3. Determination of Resulting Model

Only relevant parts of DeSpec specification and C code should be included in the extracted model to limit its state space. These parts are determined by two means – DeSpec *namespaces* and slicing algorithm. Before the model extraction and its verification a set of rules to verify should be chosen. According to chosen set, user can select a namespace with model on desired level of abstraction. During the extraction slicing must be performed on the model to determine, which parts influence checking of selected rules and constraints, and what parts of C code are to be merged with the model. The slicing algorithm must respect *CheckConstraints* attributes and include abstractions marked with this attribute into resulting model even if they are not directly connected with verified rules.

3.4. Zing Back-End

Since the target language of the compiler is a high-level modeling language, there is no need to perform operations on an inner representation of code typical for compilers to low-level or binary code. Main task for the back-end is implementation of DeSpec specific features by means common to both DeSpec and Zing, generating automata for corresponding rules and emitting Zing code acceptable by the Zing compiler.

3.4.1. Implementation of Modeling Features

As Zing does not support some concepts provided by DeSpec, appropriate workarounds must be introduced.

Inheritance is one of such features and its implementation requires an analysis of classes involved in this relationship. Involved classes must be provided with an additional mechanism that dispatches dynamic access to inherited members. Compile-time access to inherited declarations is provided during type analysis in the front-end of the compiler. Moreover, methods for implementing *is* operator must be generated and added into every class involved in inheritance. Other features, like overriding, are not required by DeSpec.

Zing does not support *properties*, which are heavily used not only in DeSpec specifications. Firstly use of properties improves specification simplicity and readability.

Secondly properties are used by the compiler for implementation of some constraints, as properties allow controlled access to data in contrary to fields. Access to properties can be simulated by generating and invoking corresponding methods.

Use of various expressions is quite restricted by Zing grammar, compared with DeSpec and also common programming languages like C#. Especially occurrences of method invocations and assignment expressions are limited. Even if these limitations propagated into DeSpec grammar probably would not cause any problem in writing specifications, it is not suitable to transfer them to users, especially if they expect typical functionality from such basic language constructs. Complex expressions that are not allowed by Zing grammar must be turned into equivalent segments of statements and simplified expressions acceptable by Zing.

Implementation of delegates relies on a pointer-to analysis of driver source code provided by the C code extractor. A dispatch mechanism based on the results of the analysis must be generated and added into the model.

3.4.2. Implementation of Constraints and Rules

As Zing provides none of the constructs for expressing constraints supported by DeSpec, workarounds using assertions must be introduced. Constrained fields are transformed into properties. Constrains related to methods and properties are expanded to assertion statements.

For implementation of rules equivalent automata must be generated at first. Their transition methods must be added into involved classes. The actual value of a property expressed by a rule and represented by an automaton can be changed from various places of the model during its execution. A mechanism for transition of appropriate automata from these places must be generated. A routine that checks final states of the automata must be added at the end of model execution.

3.4.3. Emitting Zing Code

When all DeSpec specific features are implemented by constructs common to both DeSpec and Zing, Zing code of the resulting model can be generated. A dumping routine must be provided for every node of AST present in the resulting model. Some branches of AST that were generated by the compiler during the extraction are not valid parts of the extracted model and must be cut off.

Built-in collection types cannot be specified by DeSpec on necessary level of detail and can be represented by AST nodes only on very high level of abstraction. Their real functionality can be expressed only in Zing by its own built-in types. Thus transformation from DeSpec to Zing can be made only in this step and without appropriate representation in DeSpec. However, intended functionality is known from [2].

4. Approaches to Implementation

An analysis of possible approaches to implementation of the DeSpec-to-Zing compiler is in this case quite simple and straightforward.

.NET development platform is suggested in the assignment of the thesis and seems to be the most suitable choice. Since C# was chosen as a language of implementation, it necessary to find lexer and parser generators that produce outputs in the same language or at least in any .NET language. A GPPG parser generator [9] proved to be a suitable tool for generating DeSpec parser. GPPG takes a Bison/Yacc-style grammar specification with semantic actions coded in C# and produces an LALR(1) parser. However it does not include a lexer generator so a standalone tool must be used. A CsLex lexer generator is such a tool that works well with GPPG. It accepts a Lex-like input specification and produces a C# output. Recently a GPLEX lexer generator was developed by authors of GPPG. This tool is designed to be used with GPPG and it would be probably suitable replacement for CsLex, if necessary.

As the goal of the thesis is translation of DeSpec language into Zing, the typical approach to the implementation of a compiler should be taken. A choice whether AST nodes will be so-called *smart* or *dumb* objects should be made. Since Zing is the only intended target language, there should be no problems with smart nodes, i.e. with nodes represented by classes with rich functionality. On the other hand, the concept of dumb nodes with few or no methods makes a design of a compiler more comprehensible and keeps code with related functionality at the same place. An ideal solution seems to be use of dumb nodes and implementation of required functionality in visitors. However, in case that implementation of some task seems to be more suitable in specific nodes, there is no reason to avoid that.

Compilers are referenced as ideal examples for use of *visitor* design pattern [10] and it is apparent that in case of translation of DeSpec to Zing it holds as well.

A specified model is represented by AST nodes, which correspond to DeSpec constructs, during whole process of extraction. Implementation of DeSpec features, especially in the back-end of the compiler, requires generating of other nodes, adding new branches to the AST and replacing old ones. Usually, new nodes should be at least partially based on those to be replaced. Implementation of *prototype* design pattern described in [10] and providing nodes with *clone* method significantly simplifies generating of additional code.

On if the key issues of translating DeSpec into Zing is representation of DeSpec temporal rules. Since the rules follow temporal patterns, they can be expressed in LTL formulae. Raw LTL formulae can be represented by Büchi automata. The appropriate (potentially non-deterministic) automaton can be constructed by algorithm described in [11] and then transformed into a deterministic minimal state automaton. Nevertheless, with regard to a specific set of rule patterns and character of the models, it is also possible to construct an automaton for each DeSpec rule pattern manually. Automata representing the rules can be driven by events triggered during execution of the model. At the end of the execution their states can be checked by assertions. Thus it is possible to implement DeSpec rule patterns by Zing *assert* statements.

5. Implementation of Compiler

5.1. Generating Lexer

A lexer necessary for providing a DeSpec parser with tokens is generated by CsLex tool from lex-like DeSpec lexical grammar defined in [2]. One change was made in the lexical structure. Compiler needs to create additional instances of DeSpec constructs during the model extraction, among others named enumerations, classes, members and variables. To avoid potential conflicts with names used in the original specification, a unique prefix must be reserved for identifiers generated by compiler. Two underscores ("__") are chosen for this prefix. Regular expressions standing for identifiers in lexical grammar are modified to enforce this restriction.

Conflicts of identifiers in DeSpec specification with Zing keywords are solved during emitting of Zing code by inserting the same prefix.

5.2. Generating Parser

A parser processing DeSpec code is generated by GPPG tool from DeSpec syntactic grammar. GPPG accepts grammars written in Yacc/Bison style [11] and allows defining a custom semantic value type by `%valuetype` and `%union` directives, as well as a custom location type.

This is a C# pseudo-union for transferring semantic values:

```
%union
{
  public string str;
  public int n;
  public object obj;
}
```

The field of *string* type is designed for holding string literals and identifiers, the *int* field contains integer literals and enumeration values and *object* field holds instances of AST nodes created during parsing. The union is flexible enough to transfer any semantic value or AST node. In case that it is necessary to transfer two objects at a time, e.g. in rule *AttributesAndModifiers : Attributes Modifiers*, an instance of generic class *Pair<F, S>* can be used for them and itself can be assigned to *obj* field of the union.

A default location-information class contains fields for both start and end position and fit the needs of error reporting. The only necessary information that is not automatically provided by the parser is name of processed file. For error reporting purposes, *Location* structure with name of file and position in file is added to every node of AST.

GPPG allows to provide additional code of parser in Yacc-like way directly in `%{prologue%}` section of input grammar file or in separate C# file by declaring parser class *partial*. This is used for adding a set of fields necessary for building an AST tree to the generated parser. Most of them are C# dictionaries for registering of DeSpec declarations and representing current context of processed node. Counters for generating unique names of anonymous constructs are also added to the generated parser.

The grammar production rules are usually supplemented with piece of C# code implementing appropriate semantic actions. These routines are usually triggered after the

completion of the rule. In that case they gather semantic values passed from a lower level and generate an AST node corresponding to the semantics of the rule. The result of this processing is then passed to a superior rule.

Sometimes an initialization is required before descending into a particular part of a rule. E.g. the rule for *class declaration* has following form:

```
ClassDecl :
AttributesAndModifiers ‘class‘ T_IDENTIFIER Inherits ‘{‘ MemberDecls
‘}‘
```

Example 1: Production rule for class declaration

Before parsing of member declarations of a class (`MemberDecls`), it is necessary to reset some context-related fields of the parser. During parsing of the declarations, these fields are filled with data necessary for generating the class declaration AST node after the completion of the rule.

However, most of functionality is moved to C# source files in order to keep grammar file simple and readable. It is also more comfortable to work with and debug pure C# code than code mixed with grammar rules and inserted into the implementation of a parser. Main purpose of the code segments in grammar file is generating of instances of AST nodes and building the AST from them.

5.3. Abstract Syntax Tree

The input specification is represented by an abstract syntax tree (AST) constructed during its parsing. Individual constructs of DeSpec language are represented by different node types and its occurrences in the parsed specification have their counterparts in the nodes of the constructed AST. Every type of node is represented by a specific class and relations between similar node types are expressed by inheritance.

5.3.1. Support for Model Extraction

All types of nodes are derived from an abstract base class *Node* (its simplified structure is described in Figure 1). This class has no counterpart in DeSpec grammar and contains only members that are required by the compiler from all specific node types to process a specification. It includes information about the location of corresponding segment of code in DeSpec specification and bindings to parent nodes in AST.

A key member for whole process of model extraction is *Accept* method. Being a part of *visitor* design pattern it allows traversing the AST by visitors that implement specific functionality in their *Enter/Return* callbacks.

Another feature that can be required from all types of nodes is cloning of their instances. *Clone* method returns the deep copy of a node. The only field whose copy is shallow is a reference into the table of DeSpec type declarations. Changes in nodes representing type declarations are supposed to propagate to all their references. This would not hold if a declaration in table of types changed and these references pointed to cloned instances.

Node types representing constructs common to both DeSpec and Zing also must be able to emit their representation in Zing language at the end of model extraction. *Dump* method

serves for this purpose. Inheritance relationships between node classes express similarity of specific DeSpec constructs rather then their inclusion in both Zing and DeSpec grammar. This property is not caught in the hierarchy of node types by any means so dumping method is included in every type of node as all of them inherit from the base node class.

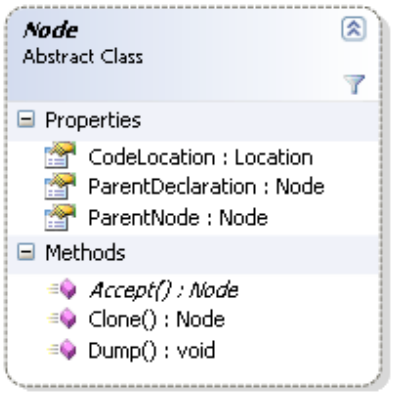

Figure 1: Structure of base class for AST nodes

In contrast with base node class, a class representing particular DeSpec construct must contain some additional members. Fields corresponding to individual parts of the construct are included and non-default constructor that accepts values of these parts must be provided. In some cases methods with added functionality can be also included. It does not conform to the concept of dumb AST objects but it is a simple solution with no drawbacks. An example of such case is *ApplyAttributes* method that processes attributes assigned to some DeSpec constructs.

Most of node types can be instantiated quite straightforwardly just after the corresponding DeSpec construct is parsed. When the appropriate syntax rule is completed, a constructor of the node type is called and subnodes held in syntactic value unions are passed as arguments. The newly created node is then propagated up to its superior rule by the parser. When parsing of this superior rule is completed, the process is repeated on that level, and so on.

However, there are some DeSpec constructs that require special treatment when their node types are instantiated and incorporated into the AST.

One of them is DeSpec class declaration represented by *ClassDecl* node type. This class contains several dictionaries for declared members, rules, structures, etc. to keep declarations of different kind separated and easily accessible. The constructor of ClassDecl class expects these separated collections as arguments. That is the reason why declarations created in the context of a parsed class must be stored in parser's collections. When class declaration rule is completed, filled collections are used for construction of the ClassDecl node.

A different approach is taken when creating a *FieldDecl* node, which represents a DeSpec field declaration. In DeSpec it is possible to declare more variables of the same type together in one field declaration (here declaration means a line of code terminated by semicolon). However, the type of declared fields is not known until whole rule is completed. Thus a list of field names and initializers must be maintained during parsing of the field declaration construct. When leaving the rule, list of real *FieldDecl* nodes must be

created and passed to a superior rule, where it will be added to other field declarations of the parent class.

DeSpec allows distribution of a namespace declaration in the specification much like C#. Therefore, one namespace can be entered end exited repeatedly during parsing. Since DeSpec does not support declaration of partial class, namespaces are the only constructs that have to cope with this problem. Even if this feature is not explicitly used in a specification, at least implicit *Default* namespace can be divided by another namespace declaration. Unlike the other nodes, the one representing namespace declaration is created at the start of namespace syntax rule, of course only if the namespace has not been already created at some other place. When the rule is completed, contents of the namespace declaration are just added to existing namespace node.

Except building of AST, there is another task that can be completed during parsing. DeSpec supports 5 modifiers that can be applied on various constructs – *static, synthetic, abstract, base* and *readonly*. If some construct contains an applicable modifier, it must be propagated to all nested relevant constructs. As information about modifiers is necessary for most of the tasks in the model extraction, it is necessary to do the propagation as soon as possible.

It would be possible to implement a special visitor that would do the job but this is not necessary. Instead, correct modifiers are passed as arguments already when creating a node. 2 fields are added to the parser, one holding or-combination of current propagating modifiers and one being a stack that stores these combinations for corresponding scopes. When entering DeSpec class declaration or structure declaration, a copy of current propagating modifiers is saved on the stack and they are combined with modifiers applied on this declaration. When creating a node within a class or structure declaration that accepts modifiers, they are combined with currently propagating ones before passing them to constructor. When leaving the declaration, the former value of the propagating modifiers is loaded from the stack. Thus in every scope all applied modifiers are known before the construction of nodes starts. It is possible that particular combination of propagating modifiers is not applicable to a specific construct. Nevertheless this is not a problem because those flags that are not applicable are never checked for presence.

Most of the tasks in model extraction process are solved by appropriate visitors when traversing the constructed AST. All visitors implement *IVisitor* interface (Figure 2) and their traversing is driven by *Accept* methods of AST nodes.

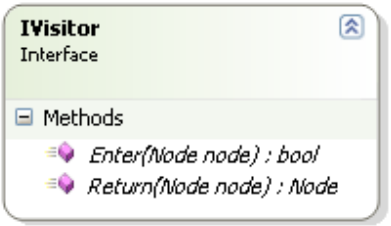

Figure 2: IVisitor interface

Original visitor design pattern proposes only one method for visiting objects called simply *Visit*. However, for working with an AST it is desirable to support conditional

traversing to avoid visiting branches that are not relevant for a particular visitor. Moreover, for some tasks it is more suitable to perform them when descending the tree and for other the opposite direction is more appropriate. Both issues are solved by replacing *Visit* method with *Enter* and *Return* methods.

A visitor's *Enter* method is invoked by a node at the beginning of *Accept* (when the visitor is descending the tree and enters a node). If *Enter* returns true, descent can continue by calling *Accept* on its child nodes. At the end of node's *Accept* method (when visitor is ascending the tree and leaves the node) visitor's *Return* method is invoked.

Visitors implement both methods actually only by a dispatch based on the type of a node passed as argument. When the type is determined, the appropriate override is called.

5.3.2. Hierarchy of Nodes

Since the rest of this text will mainly discuss implementation of specific DeSpec language constructs, a brief introduction to some of their counterparts in AST follows.

Specification node is the root of every generated AST. As such, it has an important role in driving the process of model extraction and hence the structure of this class is rather specific. The fact that DeSpec specification is actually a list of namespaces is represented by *Namespaces* dictionary. Nothing else can be declared on the level of specification, since all declarations out of explicit namespaces are included into the implicit *Default* namespace. Partial namespaces and implicit *Default* namespace require creating *Namespace* nodes on a higher level. As the only node above namespaces is the specification, it is implemented by its *EnterNamespace* and *ExitNamespace* methods and *CurrentNamespace* member. Other data members serve the needs of the compiler.

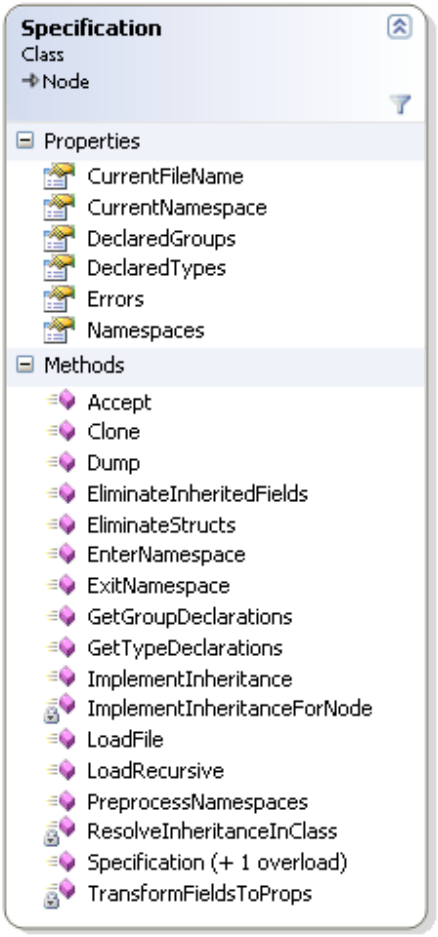

Figure 3: Selected members of Specification node

As can be seen on Figure 3, *Specification* node does not conform to the notion of dumb object. *Specification* class contains some additional properties and number of methods that are used for processing of the model. The key member for extracting of models is *DeclaredTypes* dictionary of all class, structure, union, enumeration, range and delegate declarations present in the specification. This dictionary represents global type table of the compiler and is required by most of visitors. Another important member is *Errors* list holding semantic errors accompanied with *CurrentFileName* member for error reporting purposes.

All other methods except *Node* methods' overrides trigger or implement some phases of model analysis and extraction and they will be described later.

Namespace nodes provide access to other DeSpec declarations and as soon as the type table is filled, they are needed only to check visibility of DeSpec members and declarations during the type analysis. Namespaces do not model any property or feature of a specified environment and thus they are not preserved by any means in the resulting Zing model. Their main purpose is dividing specifications into parts with various levels of abstraction.

The highest nodes in AST hierarchy that have counterparts in DeSpec code are representations of DeSpec declarations. As all declarations have some common features, a base class for declaration nodes exists. Members of this *Decl* class (Figure 4) reflect the fact that every DeSpec declaration can be marked with some modifiers and attributes. Combinations of applicable modifier flags and attributes are specific for each type of declaration, however *Conditional* attribute[2] can be applied on all of them.

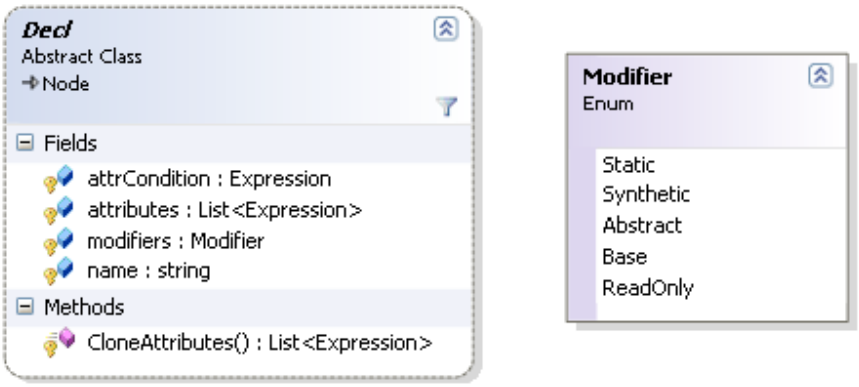

Figure 4: Abstraction of declaration and applicable modifiers

Another common property of all declaration types is that they must have a name. Even if DeSpec grammar allows anonymous rule declarations, some identifier is required by compiler for their implementation. For anonymous rules, names are generated during parsing. These autogenerated names start with “lambda” prefix. The same mechanism is used for anonymous namespaces.

DeSpec declaration types are divided in two groups. Declarations from the first group can be used on the namespace level[3] and those from the second group must be nested in some declaration from the first group. The first group is represented by *SpecDecl* abstract class and the second one by *MemberDecl* abstract class. Neither class adds new members to the parent *Decl* class and their only purpose exploiting the possibilities of polymorphism. It is often useful to maintain a collection of either only global declarations or only member declarations and *SpecDecl* and *MemberDecl* classes make this separation easy. The group of global declarations types consists of class, enumeration, range and delegate. The group of member declarations consists of field, property, method, structure, group and rule. The hierarchy of respective AST node types is depicted on Figure 5.

[2] This attribute assures inclusion of the declaration into the resulting model only under a certain condition. It is designed for model reduction process.
[3] Declarations of classes and delegates are even restricted only to the namespace level, they cannot be nested.

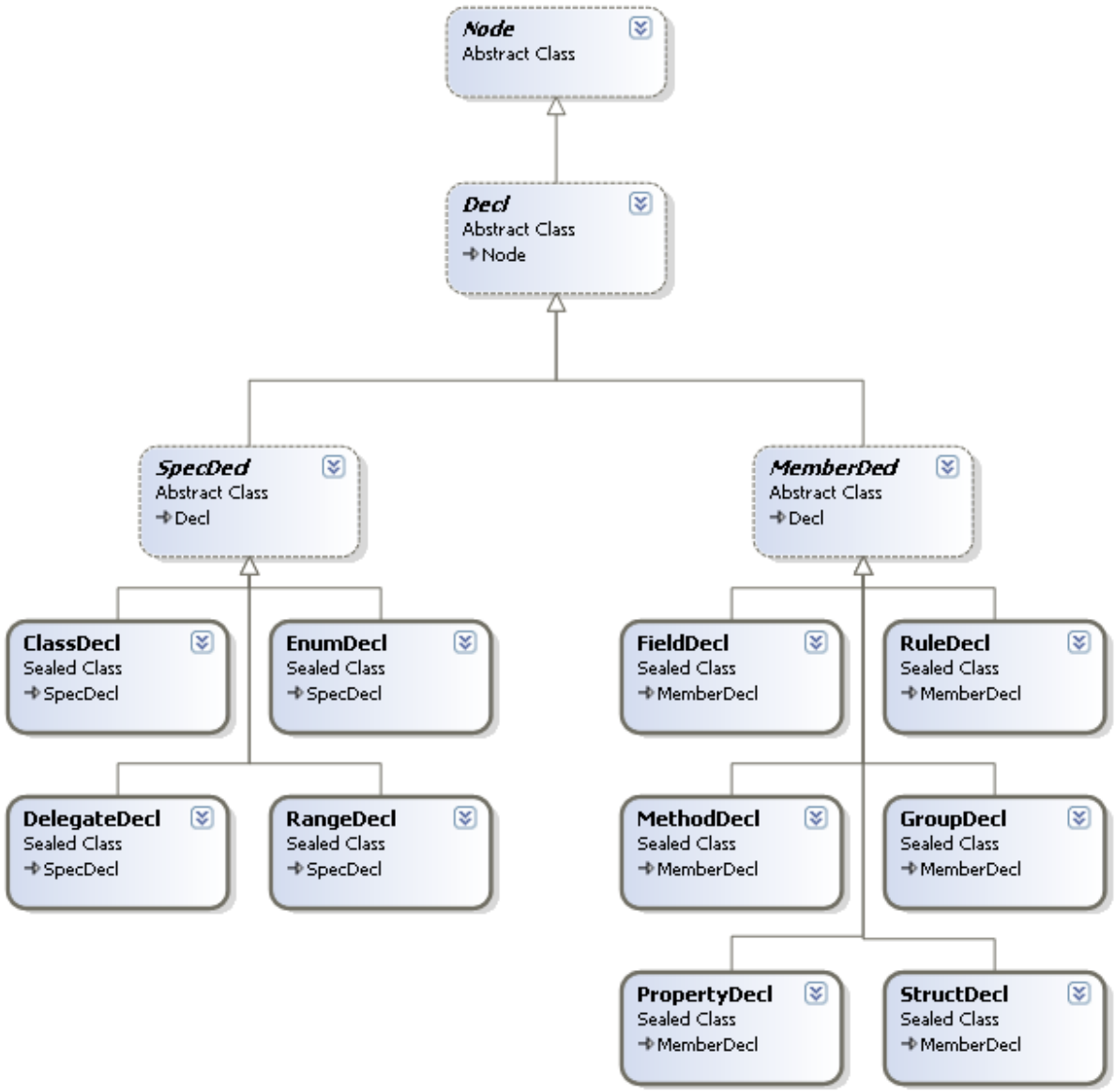

Figure 5: Hierarchy of declaration nodes

Main drawback of this classification is that it cannot be used for recognizing declarations of types, since structure declaration defines a type but it can only be nested in a class declaration. Another level of inheritance and abstract classes could be added for distinguishing e.g. between global declarations, which can appear only on the namespace level and those, which can also be nested, etc. However, such detailed classification would be utilized only very rarely by the compiler.

Nodes in lower layers of AST can represent vide variety of DeSpec constructs, which generally have nothing or very little in common. This is given by different constructs used for specifying content of individual declaration types. Classes representing these constructs will be described later in appropriate places, if necessary.

The lowest layers of AST show a kind of uniformity again. This is due to the fact that the longest paths in the tree usually end within a method body or rule declaration. Thus, nodes closest to the leaves of AST represent usually expressions, especially member accesses and literals. All expression node types are derived from common base class *Expression* (Figure 6). This abstract class introduces members holding the type information.

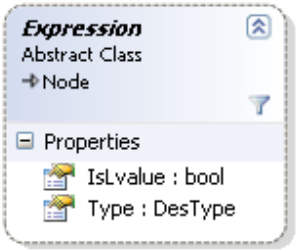

Figure 6: Base class for expressions

Value of *IsLvalue* member says, whether the expression can be assigned to. This property is determined by the kind of expression. Only *MemberAccess*, *SpecialAccess* and *ElementAccess* expressions can be potentially assigned to, if no other restrictions are applied. In case of *SpecialAccess* expression, which represents occurrences of *this*, *result*, *value* and *thread* keywords, value of this member is dependent on the concrete keyword. As a consequence, *IsLvalue* cannot be a static member and is resolved in constructors of corresponding classes.

In contrary, type of an expression is almost always unknown in the time of construction due to the fact that type information is incomplete during parsing. The only exception is instantiation of *Literal* class, because its type is given by its value. For the other cases, the type analysis must be run to correctly set the value of *Type* member. Until the type analysis, type information represented by this member is incomplete. Type hierarchy and type analysis is described in detail in chapter 5.8.

General idea about AST nodes hierarchy can be gained from Figure 7. It shows a sample branch of the tree ending in an expression included in a statement of a method. Vertical arrows represent inclusion of a subnode with double arrows denoting subnodes, which represent items from a collection. Horizontal arrows indicate inheritance relationship. To keep the sample clear, only necessary node types are included. To show a real branch consisting exclusively from instances of non-abstract classes, further inheritance bindings would have to be depicted.

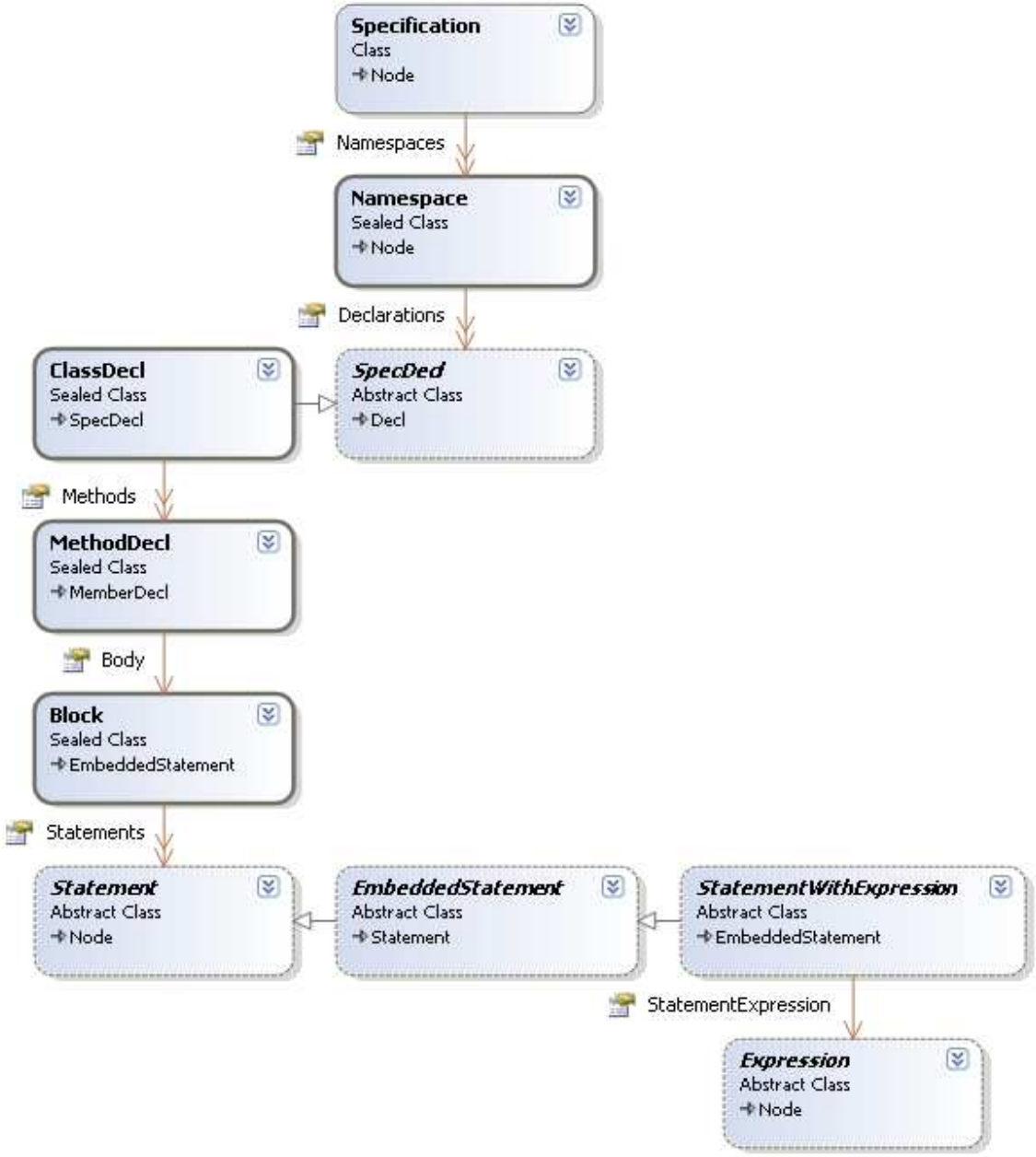

Figure 7: Sample AST branch

5.3.3. Child-Parent Bindings

During the model extraction, visitors often need a broader context when processing some nodes, i.e. information about ancestors of the visited node is required. This context can be obtained by two mechanisms.

Firstly, when descending the AST, a visitor can in its *Enter* method save the information about visited node on the top of its special stack. With such a stack for every node type, which the visitor is interested in, it is possible to determine at any moment the

closest ancestor of a specific node type as well as the others on the path to the AST root. For example, if visitor needs to know during the visit of statements declaring local variables, what is the most nested parent *Block* statement, an additional stack accepting *Block* nodes is declared in visitor's class. Every time when visitor enters a *Block* node, it pushes the entered node on the stack. When leaving a *Block*, it pops the stack. Thus, the closest ancestor of *Block* type is always accessible on the top of the stack. With one stack common for all node types, it would be possible to keep track of traversing the AST, since the stack would contain all nodes on the path to the AST root. However, a convenient implementation of such stack would require a common base class of all visitors, which would work with the stack in its *Enter* and *Return* methods. Moreover, this stack does not allow inspecting ancestors of nodes, which are not on the current branch.

More straightforward and convenient way, how to determine an ancestor of visited node, is to set up bindings from child nodes to their parents. Every node type inherits two members designed for these bindings from the *Node* base class. *ParentNode* holds information about the closest node of any type on the path to the AST root. *ParentDeclaration* points to the closest structure, union, class or node declaration. Due to these bindings, every node knows its parent and thus it is possible to effectively inspect whole path to the AST root from an arbitrary node.

The parent bindings are established by *ParentingVisitor* in an early phase of specification processing. The parser could also do the job, but determining parent declarations would be more complicated. Moreover, the implementation via *ParentingVisitor* allows updates of the bindings later in the process of extraction. This feature is very useful for the compiler, as it does not need to keep bindings correct when changing the AST. After a phase involving critical operations, *ParentingVisitor* is run on the modified AST and repairs inconsistencies.

5.4. Processing of Namespaces

Namespaces are used in DeSpec specifications to separate the models on different levels of abstraction. Before the processing of a specification starts, a namespace containing a model with desired level of detail is supposed to be selected. This selection determines which parts of the specification are to be extracted. Namespaces can be involved in a refinement relation to achieve code reuse. The original DeSpec specification proposes three types of refinement – *inclusion*, *replacement* an *extension*. A concept of extension involves also classes and enumerations and is not currently supported, as it requires quite complicated merging of code and detailed type analysis. The inclusion of another namespace can be achieved via *using* clause. *Refine* clause causes merging of the two involved namespaces and replacement of declarations from the refined one with declarations with the same names from the refining one. All declarations on the global level of a specification together compose so-called *Default* namespace. The *Default* namespace is implicitly included in every other namespace.

Selection of a namespace with a model to extract is not implemented. The main purpose of the namespaces is reduction of model's state space, which should be performed by a slicing tool. As this tool is not implemented yet, support for namespace selection is not necessary. It is easy to enforce the selection of a desired namespace manually, because

currently the *Default* namespace is always processed. Hence, it suffices to move desired top-level model class out of its original namespace.

Namespaces are not preserved in the resulting model. To complete the namespace containing a model to extract, the refining operations must be preformed in a kind of preprocessing. *Specification* class declares *PreprocessNamespaces* method, which performs this preprocessing as soon as the specification is parsed. For every namespace its namespace to refine is found, merging is performed and then the *Default* namespace is included in the result by adding appropriate *using* clause.

The namespace to refine must be found by specification object, because a namespace itself has no access to the other namespaces. In contrary, refinement itself is implemented recursively by *Refine* method of the *Namespace* class. At first, the refined namespace is cloned and processed by *NamespaceDereferenceVisitor*. This visitor simply turns all references to the original namespace into references to the refining namespace. Then *using* clauses of both namespaces are combined and eventually the declarations are merged and *refines* clause is deleted. During merging, declarations that are not specified in the refining namespace are inserted into it. Since the *NamespaceDereferenceVisitor* updated the references in these declarations, they are correct in the new context.

Inclusion of *Default* namespace by using clause simulates implicit global access to the declarations in this namespace from the other namespaces without the need of using corresponding prefix. Contrary to refinement, redeclaring of specifications from an included namespace is not allowed in the including namespace.

Although inclusion of a namespace in another one should result in the incorporation of the included namespace into the including one, this is not actually necessary. Since the *using* clauses are preserved, they can be used in the type analysis when the corresponding declarations are looked for. At the end, during emitting Zing code, the content of an included namespace can be dumped as it is, because the incorporation would cause no changes in it.

5.5. Providing Built-In Collections

The built-in collection types are not generated solely by compiler. Their generation is based on the specification of collection templates, which is supplied with the compiler and which is a mandatory part of every specification.

These templates can be parameterized with a DeSpec type and thus allow to define specific collection types (actually instances of these templates), which can be used in the specification. A specific template instance is defined simply by its occurrence in the specification on any place where a name of type can be used. The specification must be explored and all instances of collection templates must be found. For every type, which is used as the parameter of the template, a new corresponding collection type is generated and integrated into the specification. It is included in the type table and thus during the type analysis, all occurrences of a particular template instance are identified as valid DeSpec types.

A new instance of *ArrayList* collection template can be introduced in the specification e.g. by declaring a variable of this type:

```
ArrayList<int> integerArray = new ArrayList<int>(10);
```

or also by using it as a type of method parameter:

```
static synthetic int IndexOfSignaled(
                                ArrayList<DispatcherObject>! objects)
```

Necessary built-in collection types are provided by *BuiltInTypesVisitor*. When a reference to a parameterized collection type is found in the specification, it is checked, whether this type has not been already generated. If not, a new collection type, which is based on the appropriate template, is declared and added to the specification. Type of its items is fixed by the parameter of the template. It is also necessary to generate an appropriate instance of *Array* template parameterized with the same type, as this collection type is used for the *items* member representing underlying low-level collection and a proxy to Zing array. Implementation of built-in high-level collection types by instances of *Array* template and notion of Zing array proxy are described in section 5.11.6. Finally, references in the generated class declaration must be corrected *ReplaceTypeVisitor*.

```
synthetic class ArrayListOfT
{
   ArrayOfT items;

   ArrayListOfT(int size)
   {items = new ArrayOfT(size);}

   void Resize(int index,int size)
   {
         ArrayOfT new_items = new ArrayOfT(size);
         ...
   }

   void Add(T newValue)
   {...}

   T Remove()
   {...}

   void RemoveAt(int index)
   {
         ArrayOfT new_items = new ArrayOfT(items.Count-1);
         ...
   }

   int IndexOf(T item)
   {...}

   T this[int i]
   {
         get {...}
         set {...}
   }

   int Count
   {
         get {...}
         set {...}
   }
}
```

Example 2: ArrayList template

There is a simplified DeSpec specification of built-in *ArrayList* template in Example 2. Its instances are backed by *Array* collection with the same underlying type. *BuiltInTypesVisitor* starts deriving a collection type parameterized with e.g. *int* type from this template in its *CreateType* method.

This class declaration is cloned and renamed to *ArrayList<int>* to match the identifier, by which it is referred to from the rest of the specification. Then all occurrences of *T* placeholder for the type parameter are replaced with the *int* type by the *ReplaceTypeVisitor*. The same process is applied for replacing references to the original type (*ArrayListOfT*) with those addressing the generated type (*ArrayList<int>*).

After that, the *Array<int>* type is required for the items member of the newly created class. It can happen that this type has been already generated during deriving of an instance of another template parameterized also with *int*. If not, the same deriving process is applied to get the required declaration. In the end, references to *ArrayOfT* are replaced with *Array<int>* in the declaration of *ArrayList<int>* class by the *ReplaceTypeVisitor*. Thus, a correct DeSpec class declaration is generated, which can be transformed into Zing in the same standard way as any other DeSpec class. Its functionality relies solely on Zing implementation of *Array<int>* class, which is described in section 5.11.6.

5.6. Processing of Groups

The *group* construct is a mean for code reuse as well as for an abstraction in a specific context. The group declaration is basically a set of methods that can be interchanged in a specific context.

As for code reuse, it allows declaring a number of methods while providing only one common body. This feature can be used when modeling a common behavior of methods, while their different names and signatures must be preserved. DeSpec language specification also proposes a possibility of merging the common body with extensions, which are specific for the individual methods. Nevertheless, this feature is quite limited, since the extension mechanism is not currently supported.

The group declaration can be used also to abstract from differences between grouped methods, which are not important in a specific context, especially in rules. In such a case, the name of the group can be used as a placeholder for the names of the included methods. Semantics of group invocation is dependent on the place where it is used. When the name of a group is used in an invocation expression, it means that any of the included methods can be called at that place. The target is chosen randomly. When the group invocation is used as in a boolean rule expression, it is actually expanded into the conjunction of calls of all grouped methods. Thus, quantification over the methods with an existence quantifier is introduced. A group can be also used to parameterize a quantified rule. It allows to express that rule must hold for all methods from the group.

The group construct is syntactic sugar and has no counterpart in the resulting model. As such, it must be eliminated during an early phase of model extraction, in a kind of preprocessing. Groups are eliminated by individual declaring of all included methods and by expanding the expressions referring to them. This elimination is performed by a set of visitors.

At first, all group declarations from all class declarations are gathered into one table for further use by the visitors. Since group declaration allows including not only method names but also names of other groups, these lists must be expanded to contain only names of methods before the elimination starts. *GroupExpandingVisitor* is designed for this task. When expanding the lists of grouped methods, it is suitable to use the gained information to provide correct declarations for these methods at the same time.

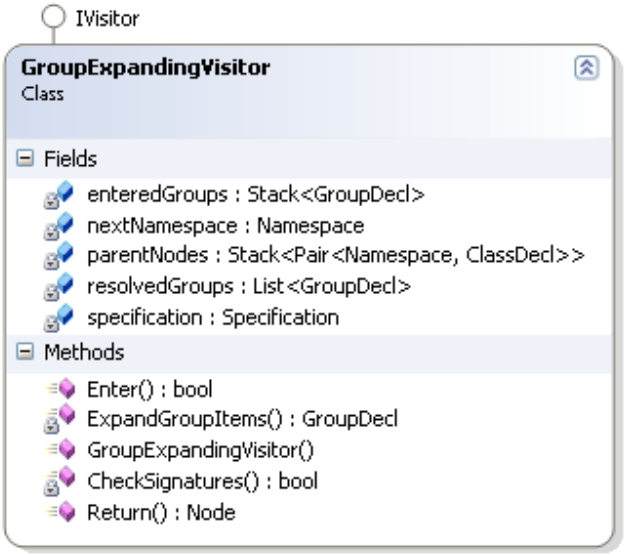

Figure 8: GroupExpandingVisitor class

Traversing of this visitor is not quite typical, mainly due to the possibility to recursively include other groups in the group declaration. The descent of the *GroupExpandingVisitor* is limited to the level of group declarations. When leaving a *group* node, all its items are inspected by *ExpandGroupItmes* method. If some of them stands for another group, which is not resolved yet, the visitor is redirected to process this group first. To manage these redirections, visitor contains some additional fields (Figure 8). To avoid reprocessing of already resolved groups during this redirecting, *resolvedGroups* list is maintained by the visitor. The member *enteredGroups* is a stack that keeps track of visitor's redirections and allows detecting mutual inclusions in group declarations. Information about parent class and namespace of currently processed group (maintained in *parentNodes* and *nextNamespace* fields) is needed to identify the methods and groups, which are represented by the items of the group.

When a class containing a group declaration includes also a declaration of a method with the same signature, it means that declaration of this method represents the common declaration shared by all methods listed in the group. In the following text, this declaration is referred as "common declaration". This code reuse is possible only when the modifiers and signatures of the group, its methods and the common declaration match. The part of this check involving inspecting on arguments and return types is done *CheckSignatures* method.

Methods listed in a group are defined as follows.

When an item in the group has no counterpart in the specification, a new method is declared with the name of the item, and the signature (i.e. parameters and return value) and body copied from the common declaration.

When both the declaration of a listed method and the common declaration are provided, contracts are merged together (if present) and the body from the declaration of the specific method is used (i.e. it overwrites the common one).

Merging of the bodies via *group* label and the extension mechanism described in DeSpec language specification is not supported.

When group declarations are resolved and appropriate method declarations are generated, group invocations can be replaced by invocations of individual methods and then groups can be removed from the model.

It remains to explain, how the group invocations are replaced by visitors. Since replacing of group invocations requires generating quite complex segments of code, it cannot be managed in a single pass through the AST. On the other hand, it is not necessary to repeatedly traverse the whole AST. Rather, a single descent from the AST root to the group invocations is performed and only relevant subtrees are processed repeatedly by special visitors. The whole process is driven by the *GroupReplacingVisitor* (Figure 9). There are also three helper visitors that deal with the individual expansions in the invocation subtrees – *GroupInvocationSearchingVisitor*, *GroupInvocationReplacingVisitor* and *GroupVarInvocationReplacingVisitor*.

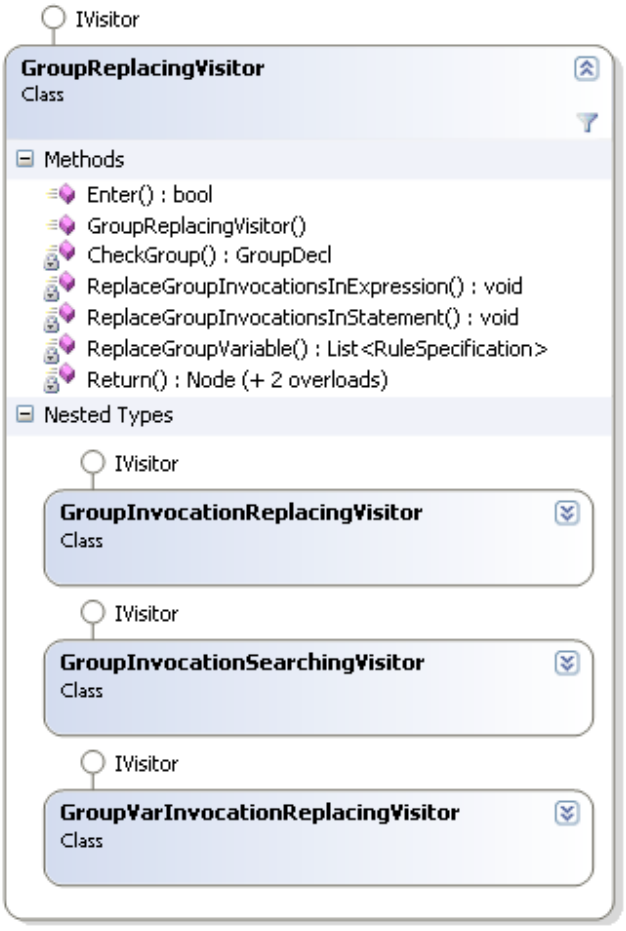

Figure 9: Visitors for elimination of groups

To deal with the invocation of groups in method and property bodies, *GroupReplacingVisitor* runs *GroupInvocationSearchingVisitor* on every statement. This helper visitor looks for group invocation expressions in the statement and when it detects such an expression, it stores information about the used group and about the target of the invocation expression in its fields. In that case, *ReplaceGroupInvocationInStatement* method of the controlling *GroupReplaceVisitor* generates an appropriate *select* statement as a replacement for the original statement containing the group invocation expression. For every method included in the group, a clone of the original statement is processed by *GroupInvocationReplacingVisitor*. This visitor simply replaces the first invocation of the group in the cloned statement with the invocation of one of the grouped methods. These clones with replaced invocations are then used as *wait* statements for the non-deterministic select. This process is repeated on the statement until all group invocations are replaced with the select statements. Thus it is assured that on the places of original group invocations an arbitrary method from that group is invoked.

Groups can be used also as quantification variables in the *forall* clause of a quantified rule. This variable is then used in the rule expressions to denote method events. In this context it means that the rule must be satisfied for every method from the group. When a rule quantified by one or more group variables is found by *GroupReplaceVisitor*, its *ReplaceGroupVariable* method is used to clone the rule specifications and to modify them to involve each of the grouped methods. In every cloned rule specification, the *GroupVarInvocationReplacingVisitor* replaces the method events referring to the group with events referring to one of the grouped methods. When the rule is quantified by more group variables, this process is repeated for every variable over the newly expanded rule. Thus, in the end each rule specification contains one combination of invocations of methods from the respective groups. Whole rule covers all possible combinations.

While a group quantification variable corresponds to a universal quantifier, the usage of group invocation directly as a target of a method event operator in the rule expression represents quantifying over the group methods by an existence quantifier. Every invocation expression using a method event operator is a boolean expression. Thus, it can be replaced with a conjunction of its clones and each clone is modified to refer to one specific method from the group. The required conjunction is created by *ReplaceGroupInvocationsInExpression* of the *GroupReplaceVisitor*. Since the replacement of targets of method event operators is analogous to the replacement of method invocations in cloned *wait* statements from the first case, same visitors are used for this task.

5.7. Type Analysis

The type analysis is the crucial phase of the model extraction. It is necessary to prove the semantic correctness of the specification and its results are used in further phases to generate the correct additional code. During this analysis, the type information is added to every expression present in the specification. With this information, it is possible to implement inheritance, to check the type correctness of assignments, matching of method calls to their signatures, etc.

5.7.1. Classification of Types

DeSpec types can be either *value* or *reference*. That corresponds to Zing *simple* and *complex* types respectively. Built-in integer types, boolean type and specifications of enumerations, ranges and delegates are all value types. Reference types include built-in end user classes. Instances of the built-in templates, *string* and *object* type represent built-in reference types. This classification does not follow DeSpec grammar, though. Only *generic*, *range* and *primitive* types are distinguished syntactically. Moreover, the type analysis requires yet another categorization of the types. Its representation by the AST node types is depicted on Figure 10.

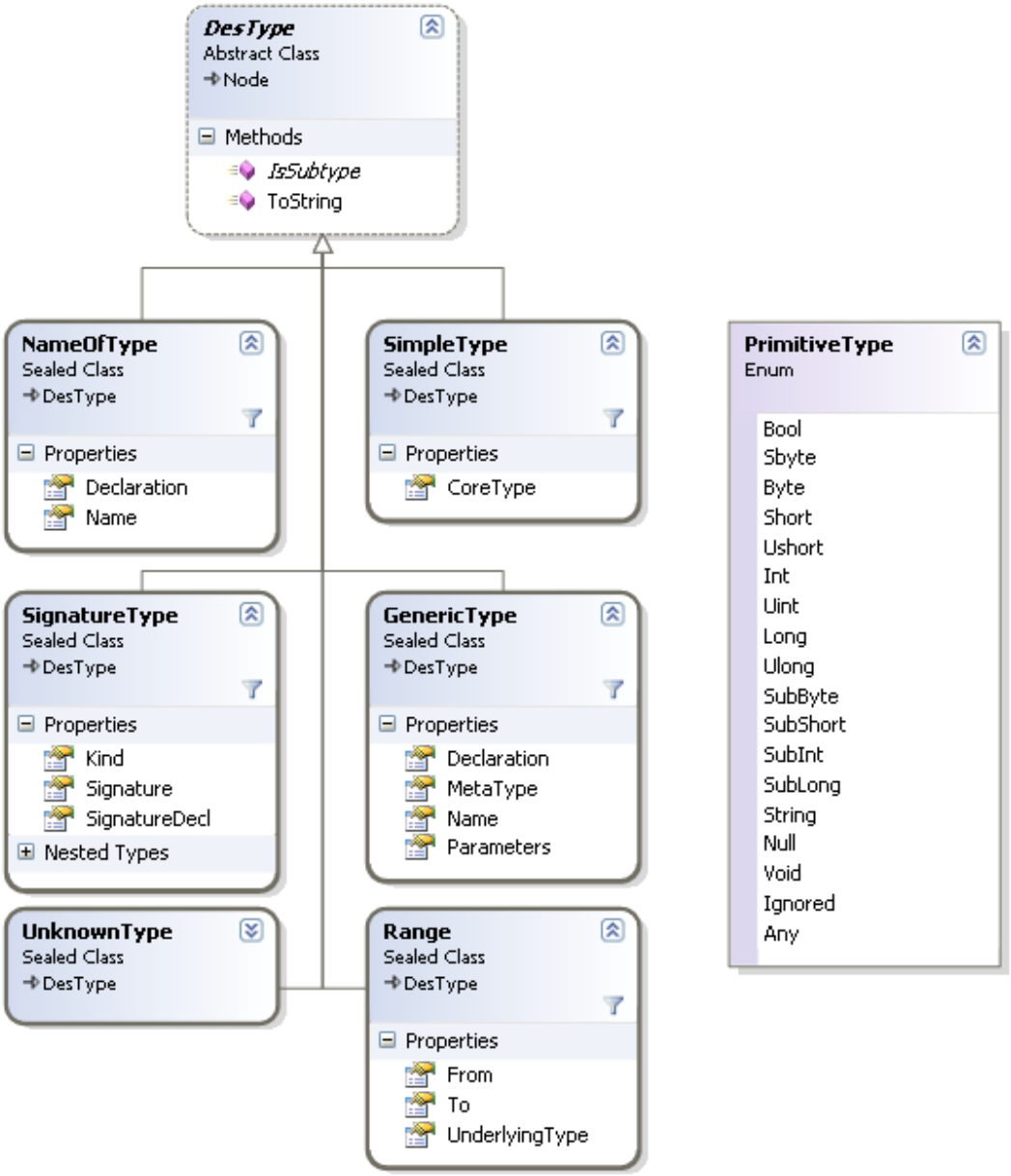

Figure 10: Type classification

The base class for all type categories is *DesType*. The key member provided by this class is the *IsSubtype* method, since this relation must be defined for all types to enable the type analysis. Only *SimpleType* and *GenericType* and *Range* represent the types generated during parsing the specification. However these three categories do not match the value-reference classification. Rather, *SimpleType* nodes correspond to primitive types without any explicit declaration like for example integer types and *Range* and *GenericType* nodes correspond to the type declarations present in the specification, like ranges, classes, enumerations, etc. The other type categories are designed only for the needs of the type analysis.

The *SimpleType* category consists of types, which do not need any additional information about themselves. The identifier fully describes the type. An example of such type can be any built-in integer type. Nevertheless, not only a value type can be represented by the *SimpleType* class. For example *string* is a reference type, but it is self-describing and thus it falls into the *SimpleType* category. This category also covers some specialties, like *null* or *ignored* type. The concrete type represented by the *SimpleType* node is determined by the value of the *CoreType* field, which holds one of the items of *PrimitiveType* enumeration (Figure 10).

In contrary, *GenericType* class represents reference types, which are declared in the specification. It allows specifying parameters of the type in the *Parameters* list, however this property can be used only by the instances of the built-in templates. User-defined classes cannot be parameterized. The reference to the declaration of the type is contained in the *Declaration* member. The value of this member is resolved during the type analysis and represents the complete type information. Often it is sufficient to know, whether the type stands for a class, enumeration, etc., it means that only the information about the kind of the declaration is needed. This information can be obtained by inspecting the type of *Declaration* member, but for convenience, it is also contained in the *MetaType* member. The value of this member is an item of the *TypeDeclarations* enumeration (Figure 11). Both of these members are resolved during the type analysis and they cannot be used till this phase.

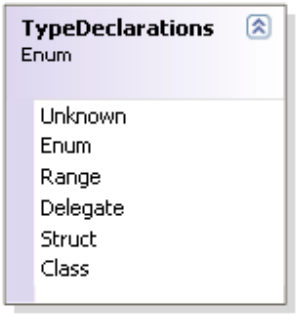

Figure 11: MetaType classification

Similarly to *GenericType*, *Range* nodes require additional fields to hold the complete type information. In contrary to *GenericType*, this information is held directly in the appropriate fields, since no range type is preserved in the form of a declaration, which could be pointed to.

The other type classes are designed only for the type analysis and they have no counterparts among DeSpec types. The *SignatureType* class is used for holding the declaration of the method. If a *MemberAccess* node refers to a method, its type is set to the

SignatureType node. In its parent expression – the invocation of the method or an operation with the delegate it represents – the declaration of the method is used to determine its type. The *NameOfType* class has a similar purpose – it is used as the type of *MemberAccess* nodes, which refer to the declarations of types and it holds these declarations. For example, in the invocation of a static method of some class, the type of the *MemberAccess* node denoting the name of the class is resolved to the *NameOfType* node containing the declaration of this class. This declaration is then used for searching of the declaration of the invoked static method. The *UnknownType* nodes are used for initializing of the type members before the type analysis and then for signaling that the type of an expression cannot be resolved (for example if the appropriate declaration cannot be found).

The type analysis consists of two phases. Firstly, the information about the type itself must be known. As the type information about a *GenericType* node is represented by its declaration, references to these declarations must be provided. As for the *Range* node, their type information is given by the bounds of the range, which are already known in the time of instantiation. However, determining of the underlying type of the range is not a trivial process, so the underlying type must be resolved and stored in the node.

When all type nodes contain the complete type information, the types of the expressions in the specification are resolved and their type correctness is checked.

5.7.2. Declarations of Generic Types

When instances of the *GenericType* class are created during parsing, nothing is known about the corresponding declarations. It can happen that at the time of instantiation, the AST does not contain the declaration of the referenced type, since it is not parsed yet. Thus, *Declaration* and *MetaType* members cannot be correctly set in the constructor and they are initialized with the special values to indicate that the corresponding type is unknown.

When whole specification is parsed and the type table is completed, it is possible to determine the types, which the *GenericType* nodes refer to. This is done by the *FindTypeVisitor*. This visitor has an access to the type table, so it knows about all declarations in the specification.

For every *GenericType* node, it finds the path back to the AST root and uses it to generate the fully qualified name of the type. Then it is checked in the type table, whether a type declaration with such name exists. It actually means that the type declaration is looked for in the closest possible scope. If the declaration is not found, the scope is broadened by shortening the prefix of the type name and the declaration is looked for in this new scope. This process is repeated until the declaration is found or the namespace scope is reached. If the declaration is not present in the parent namespace, all namespaces included by *using* clause are searched. If the search is not successful, it means that the original name is already fully qualified and the declaration can be found in the type table keyed exactly by this name. This process ensures that the closest acceptable declaration is chosen.

When the declaration of the type is found, it is stored in the *Declaration* member and the *MetaType* member is set according to its kind. Thus, the complete type information is provided for the node.

5.7.3. Underlying Types of Enumerations and Ranges

For checking the type correctness of expressions that involve a variable of the range type, it is necessary to determine underlying integer types of these ranges. It is also required for the translation into Zing, since the ranges are implemented by the appropriate integer types and additional constraints checking the bounds. The underlying types of enumeration declarations also must be resolved to allow checks of the bit operations over the flag enumerations.

The underlying types are resolved by the *UnderlyingTypeVisitor*. This visitor inspects declarations of enumerations and ranges and sets *UnderlyingType* members to the least built-in integer type, which covers all values from the domain given by the type. For ranges, the underlying type is the integer type, whose domain includes both of the range bounds. For enumerations, the underlying type is the one, whose domain includes both the minimal and the maximal item.

To allow the usage of some binary operators in the expressions involving range or enumeration value, additional integer types must be introduced. This need can be demonstrated for example on the assignment of a value from a specific range (selected e.g. by non-deterministic *choose* statement) to a variable of some integer type. The assignment expression passes the type check only if the type of the right side is a subtype of - or the same type as - the left-side type. Let the bounds of the range on the right side of the assignment be e.g. 0..1024. Then it should be possible to assign a value from this domain to the variable on the left side no matter if its type is *uint* or *int*, because both of the types cover all values from the range. However, regardless of the underlying type, which is set for the range, (both *int* and *uint* are possible), there are combinations of the resolved types, which would not pass the type check. The same problem can occur in an expression involving an integer literal, as its type is resolved in the same way.

To allow expressions like this, it is necessary to introduce the artificial integer types, which represent the intersections of the signed and unsigned versions of each built-in integer type. These intersections are represented by *SubByte*, *SubShort*, *SubInt* and *SubLong* items of *PrimitiveTypes* enumeration (Figure 10) and are subtypes of both built-in versions of the corresponding type. The underlying types of enumerations, ranges and integer literals are always set to the least possible intersection type. As a consequence, these expressions always pass the type check. The intersection integer types are not included in DeSpec grammar and can be used only by the compiler for the needs of the type analysis.

5.7.4. Resolving of Type of Expression

As soon as the *DesType* nodes contain the complete type information, it is possible to resolve the types of the expressions. Together with the operators, the expressions can involve accesses to members or variables and method invocations. The *ResolveTypeVisitor* identifies the types of these subexpressions (represented by the *DesType* nodes) and check the type correctness of whole expression with regard to the used operator and to the types of these subexpressions. This phase is also suitable for evaluation of the constant expressions. However, this feature is not implemented.

There are 5 basic expressions that can be regarded as the building units of the compound expressions involving operators. They are represented in the AST by following nodes types: *MemberAccess*, *SpecialAcces*, *ElementAccess*, *InvocationExpression* and

Literal. The *MemberAccess* refers to any variable or parameter, to any global declaration, to an enumeration item or to any class or structure member – a field, a property, a method or a nested declaration. The *SpecialAccess* refers to the special variables – *thread*, *this*, *value* and *result*. The *ElementAccess* refers to a collection item. The *MethodInvocation* refers to a call of any class method. In the following text, all these basic expressions are referred to as "accesses".

The key task for the *ResolveTypeVisitor* is to identify the types of the accesses. Then the types of the compound expressions can be easily resolved their type correctness can be checked. Resolving of *SpecialAccess* is quite straightforward, as the types can be easily extracted from the parent method declaration or from the parent class declaration. For resolving of the *ElementAccess* and the *MethodInvocation* it is necessary to know the type of their targets, which are represented by the *MemberAccess*. Thus, the main issue is resolving of the *MemberAccess* nodes.

For resolving of the type of the *MemberAccess* it is necessary to identify the declaration of its target. The declaration contains a member represented by the DesType node, which holds the complete type information, since it was resolved in the previous phase. The *ResolveTypeVisitor* contains two methods, which are used for identifying the target declaration. The *SearchUp* method traces the path from the *MemberAccess* node to the AST root and in every relevant node on the path (i.e. node that can contain the target declaration) it invokes the other method – *SearchInNode*. This method, according to the kind of the node that is inspected, explores the relevant members of the node that can contain the searched declaration. When the declaration is found, its type (that was resolved in the previous phase) is retrieved and returned.

There are several complications related to the type resolving of the *MemberAccess* nodes.

In some cases it is necessary to impose additional requirements on the searched declaration, which must be taken into account by the *SearchInNode* method. These requirements depend on the context of the *MemberAccess* node and must be recognized within the *Return* method when the visitor is leaving the access node. For example, when the access appears on the left side of an assignment, it implies that the target must be writable. Thus, when *SearchInNode* inspect members of a class and finds a property declaration with the desired name, it must be checked that the property includes the setter. The *MemberReq* flag enumeration is defined for specifying all possible combinations of additional requirements:

```
[Flags]
enum MemberReq
{
   Static = 1,
   Instance = 2,
   Readable = 4,
   Writable = 8
}
```

Another issue is related to the return value of the searching methods. The visit of the access node should result in determination of nodes' type. Thus, the searching methods should return a *DesType* node that contains the complete type information. However, there

are some cases, when this output is not sufficient. There are two types of the bindings related to the access nodes. In one direction, the access node contains the reference to the type of its target declaration. In the other direction, the property declarations and the field declarations contain the lists of the access nodes, which refer to them. These bindings are needed during the transformation of fields into properties later in the process of the model extraction. These lists of references are filled during the type analysis, when the targets of the accesses are identified. To update the lists of references, the corresponding declarations must be returned by the searching methods as well.

The last problem emerges from the fact that in some cases the type of access is dependent on the context of the access. When the target of an access node is identified as the name of a method and its declaration is found, the semantic meaning of the access is not clear. It can stand not only for the target of the method invocation or for the delegate instance, but also for the boolean expression that says, whether the function pointer mapping is established for this method[4]. The real meaning of the access must be determined from its context. In the first two cases, type of the access is represented by the instance of *SignatureType* node type. This type has no counterpart among DeSpec types. Rather, it is designed only for compiler's needs when it checks type correctness of the expressions. In the second case, the type of the access is set to *bool* represented by the appropriate *SimpleType* node. By default, the type of the method access is always resolved to the appropriate *SignatureType*. To deal with this issue, *ResolveTypeVisitor.Return* override for *MemberAccess* nodes accepts one more parameter, which says, whether to resolve the type as usual or whether the boolean expression is expected. In the context of the parent expression, the real meaning of the method access is determined and if the boolean type is expected, rather than the provided *SignatureType*, the Return method is invoked once more with this access and the additional parameter correctly set. This quite complicated workaround would not be needed, if the determining of the function pointer mapping was supported syntactically by a unary operator. Then it would be possible to resolve the method access always to a *SignatureType* node and the type of whole expression including the operator would be set to the boolean *SimpleType*. This process would be analogous to resolving of the type of the method invocation's target and the type the invocation itself (which is given by the return type of the invoked method).

When the types of the accesses are determined, resolving of the types of their parent expressions is quite simple, especially in case of the binary and unary expressions that involve only the operators applied on the accesses. The main task of the *ResolveTypeVisitor* during visits of the compound expressions and invocation expressions is checking of the type correctness. For the binary expressions it is checked, whether the operator is defined for the operands of the resolved types and whether the is-subtype relation is correct.

In case of the method invocation the more complex check is performed to ensure that the passed arguments match the condensed version of the method signature. The condensed version of the signature contains only those parameters, which are neither placeholders nor *instances*. This signature is used for the method invocation. The input arguments are checked to be subtypes of the appropriate parameters defined in the condensed signature and the output arguments are checked to be supertypes of the corresponding parameters. In

[4] DeSpec makes it possible to determine, whether a driver method from the *IEarlyBoundRoutines* interface is bound to the method from the specification.

case that this check passes but the reference type of an actual argument value does not strictly match the type of the corresponding parameter, the appropriate cast must be added. The polymorphism is emulated by transformation of the argument value into the type expression with *as* operator. The similar operation must be performed on the right operand of an assignment expression, when its type is subtype of the left operand's type.

When ascending the part of the AST representing a compound expression, the *ResolveTypeVisitor* resolves at first the types of the accesses at the leaves of the subtree. The types of literals are already solved, as they known at the time of their creation. With the type information about the leaves, the visitor determines the type of its parent expression and performs the check of the type compatibility for the operator, which defines the parent expression. This process is repeated on the higher and higher levels of the subtree, until the root expression is resolved. After traversing the AST, every expression contains the complete type information and is checked to be type-correct or its type is set to *UnknownType*, when it cannot be resolved and the appropriate error is added into the error list.

5.8. Post-Type Analysis

When the type correctness of the specification is ensured by passing the type analysis, it is necessary to check that the other semantic rules are satisfied. The specification of the DeSpec language defines a large number of rules for the usage of all sorts of DeSpec constructs. It is not possible to classify the imposed rules into some strict categories, as they are related to a wide variety of aspects of the language and they must be checked on the various levels of the AST. However, some basic groups of the rules related to the common issues can be distinguished. There are rules concerning the application of modifiers, non-nullable constraints, contracts and parameter constraints, initialization of the fields and variables, properties of the constructors and inheritance. Beside these groups, still there are many rules that do not fit into any category.

Checks of these rules are covered mostly by the *PostTypeSemanticsVisitor*. Unlike most of the other visitors, this one extensively exploits the possibility to act both when entering an AST node and when leaving it and at the same time it does not restrict its operation only to a few specific node types.

The *PostTypeSemanticsVisitor* does not only check the semantic rules but it also helps to implement some of the DeSpec features. In the declarations of methods, it transforms the constraints imposed on the parameters into the equivalent contracts (i.e. preconditions for the input parameters and postconditions for the output parameters). This move simplifies the implementation of the method constraints later during the model extraction, since it is sufficient just to turn the contracts into the equivalent assertions (see section 5.11.1).

Another important task accomplished by the *PostTypeSemanticsVisitor* is providing the initializers of the local variables and class fields. The generated field initializers are then inserted into the bodies of the constructors. The appropriate default values are generated by visitor's *GetInitExpression* method on the base of the type of the variable or field being initialized.

The implementation of DeSpec features often requires intercepting the access to some field and performing a check or another action. This is achieved by transforming such fields into the properties. The code performing the required operation is then inserted into the

generated getter or setter. For example the declaration of a non-nullable field requires that the *null* value is never assigned to this field. To implement this constraint, the field declaration is replaced with the corresponding property declaration of the same name and with the declaration of the backing field. Then the assertion ensuring that the assigned value is not *null* is inserted into the generated setter. The transformation of the fields into the properties supplemented with required checks is also performed by the *PostTypeSemanticsVisitor*. The *FieldToProperty* method is designed for this purpose.

5.9. Implementation of Inheritance

Since inheritance is not supported by Zing so far, quite a complex workaround is required for its emulation. Not all features supported in common class-based languages are necessary for DeSpec specifications. The design of the Windows kernel environment involves several patterns that simulate classes and the inheritance hierarchy and exploits the possibilities given by the control of memory layout to implement kind of polymorphism. DeSpec should allow to express these concepts, but it cannot use kernel's approach to achieve this aim, since the properties of C and Zing are quite different in this respect. The features that are useful for modeling the kernel are related to polymorphism. Above all, the support for the type conversion and access to the inherited members should be provided.

5.9.1. Phases of Inheritance Implementation

The implementation of inheritance is divided into several steps.

At first, in every class from which others are inherited, all its fields are turned into the properties. This is necessary for delegating the access to the inherited fields. This transformation must be completed before the type analysis, because that process sets up the bindings between the member declarations and their accesses in the specification. If the transformation of the fields into the properties took place after the establishment of these bindings, they would be lost. Since this modification must be performed globally and traversing of the AST is not needed, it is implemented by the *EliminateInheritedFields* method, which is declared in the *Specification* class.

Although during the type analysis the support for type conversion is not yet implemented, it is suitable to prepare the type expressions, which are necessary for emulation of polymorphism, in this phase. Since the conversion of arguments in the method invocations and right values in assignments to their supertypes is expected to be implicit in DeSpec, these values must be explicitly converted to the required supertypes by the compiler.

When the semantic correctness of the specification is checked, further steps in implementing inheritance take place. The hierarchy of classes involved in inheritance is determined and corresponding inheritance trees are built during the analysis of inheritance relationships. Thus, inheritance bindings between individual classes are recognized and the appropriate typecasting mechanism can be implemented.

For implementation of polymorphism and support for the type operators, the type conversion routines are generated for every class involved in inheritance. These routines require references to the instances of the parent and child classes. The fields for these references are also declared in this phase. Declarations of these fields allow building of

whole chain of instances of the inherited classes (inheritance chain). Since the chains represent the complete paths[5] in individual inheritance trees, they allow typecasting both to the subtypes and to the supertypes and they contain all target instances for delegating accesses to the inherited members.

In the last step, the code for setting up parent-child bindings, building of inheritance chains and especially dispatching of accesses to the inherited members is generated.

5.9.2. Analysis of Inheritance Relationships

Minor preparations for the inheritance implementation take place in early phases of the model extractions and during the type analysis, namely transformation of the inherited fields into the properties and the explicit typecasts of the values to the supertypes on places, where the implicit conversion is expected. The key steps are made after the completion of the semantic analysis.

First of all, hierarchy of classes involved in inheritance must be recognized. This information is gained during traversing the AST by the *InheritanceAnalysisVisitor* (Figure 12).

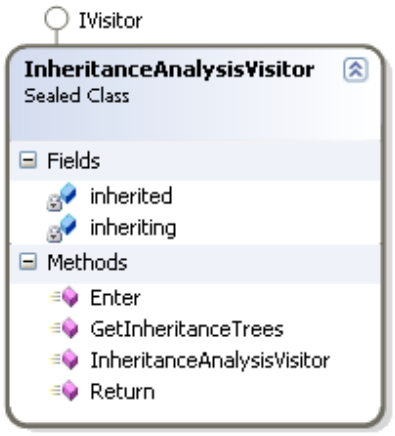

Figure 12: Visitor for inheritance analysis

This visitor descends just to the level of class declarations and stores the information about inheritance relationships in two collections. The *inheriting* list contains all non-base classes, which are involved in inheritance, i.e. they inherit from other classes. The *inherited* collection contains for every class the list of all its descendants, i.e. the classes, which are inherited from it directly or indirectly. Thus, *inherited* includes the base classes and all their derivatives except the terminating ones. These members are designed for gathering the information about the inheritance bindings when ascending the AST. A data structure that reflects the class hierarchy more conveniently is needed for further processing.

When the analysis is completed, the visitor's *GetInheritanceTrees* method creates such structure. For every group of classes, which are derived from a specific base class, an inheritance tree is generated. This tree is composed of *InheritanceNode* nodes. Every node contains the declaration of the represented class and the list of nodes, which represent the

[5] Here the complete path stands for the path from the real class of the object to the root of the inheritance tree, which represents the base class. The object can be represented by any instance from the inheritance chain, depending on its actual type context.

immediately derived classes. Since DeSpec does not allow multiple inheritance, there are no cycles in the hierarchy and every inherited class is included in exactly one tree.

Although Zing *object* type can be regarded as the base class of every other class declared in the model, it has quite specific meaning due to it typecasting possibilities and it is not included in the inheritance hierarchy.

5.9.3. Support for Type Conversion

The support for the type conversion is necessary for emulation of polymorphism. It is required for both explicit casting via *as* operator and implicit casting to a supertype in expressions which require that. The *as* type expressions for implicit casting of the arguments and the right values are prepared during the type analysis.

For every class involved in inheritance, two type converting routines are generated and added to its declaration – *__upcast* and *__downcast*. The *__upcast* method attempts to find an instance of the target type in the upper part of the inheritance chain, i.e. among the nodes on the path from the actual instance to the instance of the base class. The *__downcast* make this attempt in the lower part of the inheritance chain, i.e. on the path from the actual instance to the instance of the real type of the object. The *is* and *as* type operators are implemented using these methods.

The information about the target type is passed to these methods by the argument of the *__Classes* enumeration type. The declaration of such enumeration is possible due to the fact that number of classes involved in inheritance is fixed and known to the compiler from the results of the inheritance analysis. This enumeration is filled during the final phase of inheritance implementation on the specification level. Every item represents one class involved in inheritance. Thus, the type operands of *is* and *as* operators can be represented by these items and passed as arguments to the type converting routines.

Absence of Zing *object* type in inheritance hierarchy is not a problem. Every valid *is* expression with the *object* operand is implicitly true and thus no calls of the type converting routines are needed to replace it. Zing allows assignment of a strongly typed value to a variable of *object* type as well as assignment of a value held in a variable of *object* type to the variable of the same real type. In both of the cases, the necessary type cast is provided by Zing. Thus, no explicit *as* expression with *object* operand is needed in the specification.

When the *is* operator is used in a type expression, the type converting methods are called and attempt to convert the actual type of the expression value into the target type is made. Since it is not known, whether the target type a supertype or subtype of the value's actual type, both of the parts of the inheritance chain must be inspected. The expression is true, iff one of the attempts is successful, i.e. the instance of the appropriate type is present in the inheritance chain. Since this condition is quite simple, all *is* expressions in this form

<*variable-name*> `is` <*type-name*>

are replaced with this equivalent conditional expression:

```
<variable-name>.__downcast(__Classes.<type-name>) != null
||
<variable-name>.__upcast(__Classes.<type-name>) != null
```

In case of the *as* expressions, the replacement is more complicated, since the return value must be preserved. To solve this issue, *__as* method is generated for every class involved in inheritance and added to its declaration. This method calls the appropriate *__downcast* and *__upcast* methods as necessary and returns the result of the cast. For returning the result, the implicit Zing typecasting from and to *object* type is used. The instance of the target type, which is returned either from *__downcast* or *__upcast* has already the *object* type and it is returned from the *__as* method unchanged. Out of the method, the result is implicitly converted back to the target type. When the cast to the target type is not possible, because the real type of the inspected object is a supertype of the target, *null* value is returned. The declaration of the *__as* method is following:

```
object __as(__Classes.<type-name>)
{
   result = __downcast(__Classes.<type-name>);
   if(result == null)
         result = __upcast(__Classes.<type-name>);
}
```

This implementation exploits the implicit declaration of the *result* variable and the implicit return statement provided in by the *ZingFinalizingVisitor* during generating of method's models (see section 5.11).

The support for the type conversion is provided by the *CastImplementingVisitor*, when it enters the *ClassDecl* nodes representing the classes from the inheritance hierarchy. It generates declarations of *child* and *parent* fields for child-parent bindings in the inheritance chain and also the type converting routines – *__downcast*, *__upcast* and *__as*. The information about the inheritance bindings is gained from the inheritance trees generated during the inheritance analysis.

5.9.4. Access to Inherited Members

To make use of the class hierarchy, access to the inherited members must be provided. This is achieved by the delegating the access to the instance of the class, which contains the declaration of the accessed member.

Let class B be inherited from A. The B class contains an additional field of type A (*parent*) and the A class contains an additional field for every class, which is inherited from it (*child*). Every *child* field has a type of the corresponding inherited class. When a new instance of the class B is created, the class A is also instantiated and bindings between the two objects are established. The object of type A is set as *parent* in the object of type B and reference to this B object is stored in the appropriate *child* field of the A object. When a member inherited from A is accessed in the B instance, the access is delegated via the *parent* field to the instance of the parent class.

In case of a class hierarchy with more levels of inheritance, the members inherited from more distant ancestors are accessed recursively with a higher level of indirection. When creating an instance of a class involved in inheritance, all its ancestors must be instantiated at the same time. They are linked by the parent-child bindings and the inheritance chain is

formed. This chain can correctly represent the instance in any valid type context. As the instances of all ancestors must be included in the chain, the abstract classes, which can stand on the top of inheritance hierarchy, are turned into ordinary classes to enable their instantiation. The *abstract* modifier is relevant only for the semantic analysis, which checks that no instances are created explicitly in the specification.

DeSpec does not allow overriding of the inherited class members and the semantic analysis checks that the names of the members are unique when merged into one set. The only exceptions are auto-generated methods *Initalize* and *CopyTo*. These methods should be declared in every class in the specification and their purpose is to enable zero-initializing and copying off all class' fields. These methods cannot be modeled in the specification, because the complete set of the declared fields is not known till the end of the extraction process. The methods also allow specifying how many bytes of the fields should be initialized with zeros or copied. For implementation of this functionality it is necessary to determine the offsets of the fields in the corresponding C declaration. Since the analyzer of C source code is not yet available, *Initalize* and *CopyTo* methods are not currently supported.

The approach via the inheritance chains is quite straightforward but requires generating of the dispatching code. For methods and properties, this redirection to the parent instance is inserted into their bodies. The fields must be turned into properties and the redirecting code is inserted into their getters and setters.

Example 3 shows access to the inherited members in simplified fragments of two classes involved in inheritance. *KEVENT* class is inherited from *DispatcherObject* class and no other classes are included in the corresponding inheritance tree. *DispatcherObject* contains the original declarations of one instance method and one static method. The additional *__child_Default_KEVENT* field that contains an instance of the derived *KEVENT* class is generated by the *CastImplementingVisitor* (described in detail in section 5.9.3). Most of the code shown in the fragment of KEVENT class is generated by the compiler. The reference to the instance of the parent class (*__parent*) is set to the *DispatcherObject* object instantiated by the extended *KEVENT* constructor. The declarations of the inherited methods are copied and their bodies are replaced with expressions, which invoke the methods on the appropriate targets – the instance methods are invoked on the *__parent* field and static methods are invoked on the *DispatcherObject* class.

```
class DispatcherObject
{
   synthetic KEVENT __child_Default_KEVENT;

   synthetic bool IsIrqlCorrect(LARGE_INTEGER timelimit)
   {...}

   static synthetic bool IsSignaling(
                                ArrayList<DispatcherObject>! objects,
                                WaitType type)
   {...}

   ...
}
```

```
class KEVENT
{
   synthetic DispatcherObject __parent;

   synthetic KEVENT()
   {
         __parent = new DispatcherObject();
         __parent.__child_Default_KEVENT = this;
         ...
   }

   synthetic bool IsIrqlCorrect(LARGE_INTEGER timelimit)
   { __parent.IsIrwlCorrect(timelimit); }

   static   synthetic   bool   IsSignaling(ArrayList<DispatcherObject>!
objects,
                                           WaitType type)
   { DispatcherObject.IsSIgnaling(objects, type); }
   ...
}
```

Example 3: Access to inherited members

The supplementary code for providing access to the inherited members is generated at several different places during the model extraction.

The declarations of the additional fields for the parent-child bindings and the code instantiating the parent class and setting the bindings, is created in the earlier steps. The dispatching methods and properties are added by *ResolveInheritanceInClass* method declared in the *Specification* class .In the final phase of inheritance implementation, the specification recursively calls this method for all classes from the inheritance trees

5.10. Rules

Rules are the major feature of DeSpec language and also one of the most complicated. Their implementation requires generating of a complex checking mechanism and extensive changes in the extracted model.

At first, a brief introduction into the semantics of rules is appropriate. Detailed description of all features and properties can be found in [2].

A rule can express requirements related to so-called *source code events*. The source code events can be addressed by two groups of *source code event operators*. The first group contains operators that address entering a method and returning from a method. The second group consists of operators that address accesses to fields and properties.

The operators can be combined together in common boolean expressions. These expressions are then used in specific *temporal patterns*. The patterns express an order of precedence or a chronology of specified events that must hold during the execution of a model.

Moreover, a temporal rule pattern can be quantified by variables listed in *forall* clause. Variables listed in the clause can be used in rule expressions as parameters or targets of operators. A quantified rule must hold when applied on every value (or combination of values) of quantification variable (or variables) present in the model during its execution.

An example of a rule using some of these features follows:

```
rule
   forall(DEVICE_OBJECT device)
   { CreateDevice(_,out device)::succeeded }
   leads to
   { device.IoDeleteDevice()::returned }
   globally;
```

Example 4: Quantified "leads-globally" rule

This rule verifies that all instances of *DEVICE_OBJECT* created by *CreateDevice* method whenever during model execution are eventually deleted. Every instance created in the model must be checked for fulfilling this requirement.

Rule patterns supported by DeSpec are easy to use shortcuts for temporal patterns in LTL. Actually the LTL-X subset is sufficient for definition of the patterns because *Next* operator can be avoided. It is possible to verify LTL-X formulas by a corresponding Büchi automaton on finite traces. Reasons why it suffices to constrain only to finite traces are discussed in detail in [2]. For every LTL-X formula it is possible to construct a Büchi automaton for its run-time verification.

In the context of driver environment, verification properties of necessary Büchi automata can be further concretized. As required Büchi automata are defined by their states, input alphabet and state-transition function, a representation by Zing integer variables, arrays, enumerations and a transition method come into consideration.

Rules can be *instance* or *static*, depending on targets of used operators. This property of a rule determines a place, where the state of the corresponding automaton is stored. Thus, mixing of static and instance targets in one rule can bring forth significant problems in the implementation, even if it is allowed by DeSpec.

Rules are declared within the scope of a DeSpec class but they are bound to the parent class by no means. However, an implementation of the corresponding automaton must reside in a particular class and its location determines its accessibility from other parts of the model. This is an issue in case of instance rules, as some mechanism must be provided for keeping track of instantiated automata and calling their transition methods.

During the development, the implementation of all features of DeSpec rules turned to be very time-consuming task. Priority was given to an implementation of basic features, which are necessary to express at least some of requirements imposed by Windows kernel on drivers. Successful completion of this task would prove that concept of rule temporal patterns is feasible and its role in model verification is well-designed.

Current state of rule patterns implementation follows.

Basically, it is possible to declare instance rules qualified by a single variable of a reference type, if other variables are listed, they must represent a group. Group variables have different semantics and causes expansion of rule to cover all methods listed in referenced groups. Only default severity *error* is supported.

Only selected temporal patterns are supported. This limitation results from the fact that equivalent automata templates are built-in in the compiler rather than generated by an LTL-converting algorithm. Implementation of an algorithm for converting LTL formulae to Büchi automata would be a more universal solution. Nevertheless, this task is not quite trivial and generated automata should be further processed to make them deterministic and

normalized. Considering the closed set of DeSpec temporal patterns and low complexity of corresponding formulae (at most 4 variables involved) implementation of rule patterns by built-in automata templates is acceptable. Every pattern consists of two parts – *property* and *scope*. All properties except *corresponds-to* are implemented. As *corresponds-to* property is the conjunction of *leads-to* and *precedes* properties, it can be easily replaced with them, if needed. As for the scopes, *after-until* and *between-and* are not implemented so far.

As for pattern expressions, only && and ||combinations of method event operators are allowed. Access to variables is forbidden with exception of the one declared in *forall* clause and *this*. These two variables can be used only as invocation targets or arguments of methods supplied with an event operator. Use of static method is allowed only if a variable with instance of parent class of the rule is passed as an argument. Regarding limitations imposed on variables and logical operators in expressions, there is no use for !== method event operator. Using literals as arguments is not supported.

DeSpec introduces syntactic sugar that allows to express rules containing factories in a short form without quantification. This is not supported since the expansion of such rule into appropriate long form by the compiler is quite complicated.

Pure static rules, i.e. rules with only static targets of event operators, are not supported, since they are not used in the specification of Windows driver environment. *Static* modifier is not applicable. In spite of this, a procedure for generating automaton for a static rule should be similar to the one for instance rules.

Even with all limitations described above, the grammar for rule patterns is still expressive enough to specify many properties, which are required by Windows kernel.

Generating of automata for specified rules is driven by *RuleImplementingVisitor*, which triggers construction of an automaton for every declared rule. This visitor also maintains segments of code triggering source code events. This code is generated during automata construction but it is inserted into appropriate places later, when the compiler is finalizing bodies of methods.

5.10.1. Rule as Automaton

Once a rule is transformed into an automaton, it is possible to verify it by Zing *assert* statements. Breach of a rule can be recognized by two ways. If there is no possible transition for a triggered source code event from the current state of the automaton, an assert in its transition method is broken. When the model checker discovers this state, verification fails and violation of the rule is reported together with the corresponding trace. But even if for every received source code event a transition in the automaton exists, it does not necessarily mean that the represented rule fulfilled. If at the end of model execution the automaton is found in a non-accepting state, it means that the rule is violated as well.

The solution for the first case is quite straightforward, as the violation of the rule is realized within its transition method. When implementing this method, it is sufficient to list only transitions that do not break the rule and append an assertion that always fails.

The second case requires checking of the state of the automaton at the end of model execution. Firstly, it implies that information about state of the automaton must be stored independently on the automaton itself because the scope of its existence is the same as that of its parent class. Secondly, states of all automata must be accessible from one place at the

end of model execution. Both persistence and accessibility issues are solved by using static arrays added to the declaration of a main class of the model. These arrays hold information about an actual state and non-accepting states of all automata, which were instantiated during model execution.

The solution of this problem relies on some properties of the specification, which are characteristic for a model of Windows driver environment. In context of this environment, the end of model execution always corresponds to the end of *Main* method. This static method is a required part of the specifications, since it manages entire life of the driver. It serves as the entry point for Zing model checker. It must be declared in *Model* static class, which represents model of I/O manager's behavior the with respect to the driver. These requirements are stated by DeSpec language specification rather then by the compiler. As the presence of *Model.Main* method can be taken for granted, code that checks violation of rules is appended at its end. The static arrays with current and non-accepting states of automata are declared in the *Model* class.

In the following text only instance automata are considered. Potential implementation of static automata would be similar and in some cases even more simple.

From the discussion, it is obvious that an automaton cannot be represented by a single Zing class. Rather, its representation is scattered throughout the model. At first, automaton's parent class must be determined. If the corresponding rule is quantified, the class represented by the quantification variable is the parent one. Otherwise, the class declaring the rule is the one. A parent class contains declarations of automaton's transition method and fields necessary for determining current letter of the input alphabet. Automaton's states must be stored on common place in *Model* class. The violation checking routine is placed in *Model.Main* method. Moreover, enumeration of all methods, which can trigger an event used in the rule, must be declared. It is used also for determining current letter of the input alphabet. Since it is an enumeration, it must be declared in global scope, according to Zing grammar.

Since generating of the transition method will be described in detail, an example of its representation in Zing follows. It moves an automaton generated for this rule:

```
rule
   forall(IRP irp)
   {
         irp === IoAllocateIrp(_) ||
         irp === IoBuildAsynchronousFsdRequest(_,_,_,_,_,_)
   }
   leads to
   { irp.IoFreeIrp()::returned }
   globally;
```

Example 5: DeSpec rule

This rule uses *v0 leads to v1 globally* temporal pattern, which is expressed by □(v0 ⇒ ◇v1) LTL formula. An equivalent Büchi automaton is depicted in Figure 13:

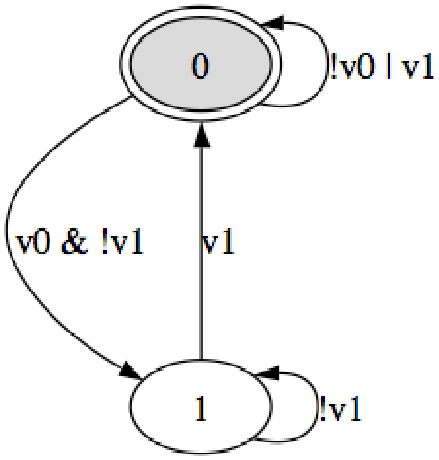

Figure 13: "v0 leads to v1 globally" equivalent automaton

The simplified and commented transition method for this automaton follows. :

```
//Parameters determine the event which caused the transition.
//'action' tells about the method triggering the event and '_event'
denotes //the event operator (entered/returned/failed/succeeded).
//Since the rule is anonymous, 'lambda3' suffix was autogenerated.
void __Step_lambda3(IRP___Actions_lambda3 action,__Events _event)
{
atomic
{
   //1. Find out which event was triggered
   //and update corresponding sub-expression.
   if((action == IRP___Actions_lambda3.IoAllocateIrp))
         (__action_0_lambda3 = (_event == __Events.Succeeded));
   else if((action ==
               IRP___Actions_lambda3.IoBuildAsynchronousFsdRequest))
         (__action_1_lambda3 = (_event == __Events.Succeeded));
   else if((action == IRP___Actions_lambda3.irp_IoFreeIrp))
         (__action_2_lambda3 = (_event == __Events.Returned));

   //2. Evaluate root expressions.
   bool  new__ruleExpressionValue_0_lambda3  =  (__action_0_lambda3  ||
__action_1_lambda3);
   bool new__ruleExpressionValue_1_lambda3 = __action_2_lambda3;

   //3. Check whether some of the root expressions were changed by
event
   bool change =
   ((__ruleExpressionValue_1_lambda3 !=
                                    new__ruleExpressionValue_1_lambda3)
   ||
   ((__ruleExpressionValue_0_lambda3 !=
                                    new__ruleExpressionValue_0_lambda3)
   ;
```

```
    //4. If so, update the stored root expressions values
    if(change)
    {
    (__ruleExpressionValue_0_lambda3 =
                                  new__ruleExpressionValue_0_lambda3);
    (__ruleExpressionValue_1_lambda3 =
                                  new__ruleExpressionValue_1_lambda3);
    }
    else
          return ;

    //5. Copy values to variables used in the transition routine common
    //for this rule pattern
    bool v0 = __ruleExpressionValue_0_lambda3;
    bool v1 = __ruleExpressionValue_1_lambda3;

    //6. Load current state of the automaton from the global arraylist
    int state;
    (state = Model.__automataStates.thisGet(__automatonIndex_lambda3));

    //7. Transition routine - [states] X [alphabet] -> [states] step
    select first
    {
          wait(((state == 0) && (!(v0) || v1))) -> ;
          wait(((state == 0) && (v0 && !(v1)))) -> (state = 1);
          wait(((state == 1) && !(v1))) -> ;
          wait(((state == 1) && v1)) -> (state = 0);
          wait(true) -> assert(false, "rule broken");
    }

    //8. Save the new state of the automaton back to the global
arraylist
    (Model.__automataStates.thisSet(__automatonIndex_lambda3,state));
}
}
```

Example 6: Transition method of automaton

For every supported rule pattern, a common part of transition method and a set of non-accepting states are generated. They are available as a part of *RuleImplementingVisitor*. The generated transition routine (Example 6, section 7) is common for all rules following the particular rule pattern. Thus, it must not contain variables with values of rule-specific sub-expressions. It includes only variables standing for root expressions, which are known from the rule pattern (Example 6, section 5). To adopt the common transition routine into the transition method of the specific rule, it is necessary to analyze these expressions and evaluate the variables in the routine accordingly.

5.10.2. Analysis of Rule Expressions

The analysis of rule expressions is performed by a *RuleExploringVisitor*. One instance of the visitor explores all expressions in the rule. The goal of this visitor is generating supporting code for the transition method of an automaton. It prepares declaration of *rule expression variables* that reflect values of the rule expressions. Their values will be then used in the transition routine of the automaton. As the result of transition routine is given by the current state of the automaton and by the values of these variables, the boolean combinations of these variables actually forms the input alphabet of the automaton.

The visitor is run on clones of the rule expressions and recognizes method events. It replaces these operators with boolean helper variables, effectively generating initializers for the rule expression variables (Example 6, section 2). It also prepares declarations of these helper variables and bindings between these variables and corresponding method events. Thus, after traversing the rule the visitor contains all bindings for the automaton.

RuleExploringVisitor also prepares items for an enumeration representing all methods in rule expressions. This enumeration is then used as type of an argument of the transition method to determine the event (Example 6, section 1). For the sample rule (Example 5), following enumeration is generated:

```
enum IRP___Actions_lambda3
{
   IoAllocateIrp,
   IoBuildAsynchronousFsdRequest,
   irp_IoFreeIrp
};
```

Another enumeration with items representing event operators is declared. This enumeration is used globally by transition methods of all automata.

```
enum __Events
{
   Returned,
   Succeeded,
   Failed,
   Entered
};
```

This enumeration does not include an item representing `===` operator, as it is translated to *succeeded* operator. This transformation is possible thanks to limitations imposed on rule variables. Since only reference quantification variables and *this* can be used as the left value for `===` operator and only invocation expression can stand on its right side, an expression including this operator is true iff the invoked method successfully returns the instance from the left side. Thus, the transition method of an automaton checking the rule for the instance is triggered by *succeeded* event iff the instance is returned by the invoked method.

With the outputs of the rule expression analysis and enumeration of the event operators it is possible to provide arguments for transition method calls. When a method event occurs, i.e. the method is entered or returning, transition methods of all interested automata are called. Arguments passed to these calls specify which method triggered the event (by an item from the first enumeration) and which kind of event it is (by an item from the second enumeration). In the transition methods, this information is used for evaluation of rule expression variables, which were declared during the rule expression analysis.

5.10.3. Evaluation of Rule Expressions

During the rule expressions analysis, method event operators were replaced with variables. When transition method of the automaton is invoked, these variables must be

correctly evaluated and their values must be used to specify, which letter from the input alphabet will determine the transition.

The input alphabet Σ is defined as a set of all combinations of root expressions. For the sample rule (Example 5), which follows *v0 leads to v1 globally* pattern involving two variables, the alphabet is $\{v0 \wedge v1, v0 \wedge \neg v1, \neg v0 \wedge v1, \neg v0 \wedge \neg v1\}$. Generated transition routines understand only variables from rule patterns, which correspond to rule expressions in specific rules. The letter determining the transition is given by the combination of these variables.

To get the values of root expressions it is necessary to evaluate rule expression variables declared corresponding to used method event operators. At first, event triggering the transition method must be determined from the arguments (Example 6, section 1). Then, root expressions are evaluated (Example 6, section 2). Triggered event itself does not necessarily cause the transition of particular automaton. It just informs interested automata and they alone decide whether to move or not. This decision is based on changes in values of the root expressions. This is the reason, why the values of root expressions must be persistent and appropriate fields storing their values must be declared in parent classes. After the evaluation of root expressions, the decision about performing a transition is made and updated values of root expressions are saved (Example 6, section 3 and 4).

The last necessary step before performing the transition is mapping of rule expressions' values to variables, which are understood by the transition routine (Example 6, section 5).

5.10.4. Transition of Rule Automaton

The core of a transition method is its transition routine – *select* statement, which implements the transition function of an automaton (Example 6, section 7). For an automaton with an input alphabet Σ and a set of states S, this function is defined as $T : S \times \Sigma \rightarrow S$. The codomain of the function is given by determinism of used automata. In *select* statement, they can be assured by using *first* qualifier and by correct order of *wait* branches. *First* qualifier ensures selection of the first valid *wait* statement in the list. Another option is to use strictly only the letters defined in the input alphabet and include all possible transitions given by the alphabet. Thus, always exactly one wait statement in the select would be valid and determinism would be assured.

It remains to explain, what exactly is meant by "using strictly only letters from the alphabet". A transition from one particular state *s* for two different letters φ and ψ can end up in another state *t* common for both of them, i.e. $T(s, \varphi) = t$ and $T(s, \psi) = t$. In this case, the two transitions can be merged together and they can be conditioned by the disjunction of the two letters - $T(s, \varphi \vee \psi) = t$. As the letters represent logic formulae, they contain inner semantics, which is hidden to the alphabet. However, this semantics allows to express the compound formulae by letters, which are not included the alphabet, but which are logically equivalent with the included ones. E.g. in an automaton on Figure 13, there is actually only one transition (from state 0 to state 1) defined by a valid letter from its input alphabet[6]. The others make use of merging and logical meaning of the letters. The transition from state 0 to state 0 through "not-defined" letter $\neg v0 \vee v1$ covers three transitions driven

[6] Addressing the alphabet defined in section 5.13.3

by valid letters v0 ∧ v1, ¬v0 ∧ v1 and ¬v0 ∧ ¬v1. The transition from state 1 to state 0 trough letter v1 covers transitions for letters v0 ∧ v1 and ¬v0 ∧ v1. The transition from state 1 to state 1 covers transitions for the rest of the alphabet. It is obvious that every transition function that exploits merging possibilities and the extended alphabet is equivalent to T.

Computation of the values necessary for determining the letter parameter of transition function was described in the previous section. The other parameter – current state of the automaton – is simply retrieved from the globally accessible static array, which is declared in *Model* class (Example 6, section 6). The index into this array is set during instantiation of the automata and it is fixed for whole its lifetime.

Bodies of *wait* statements are actually just simple assignment statements that set new state of the automaton. When no transition is needed, i.e. the new state is the same as the old state, only void statement is used. The only exception is the last *wait* statement with "catch" functionality. This branch is selected only if no listed transition is possible, results in violation of assert statement and causes failure of the verification. This is one of two mechanisms for reporting the breach of a rule. The other mechanism is based on checking the current states of automata at the end of model execution and is described in next section.

It would be possible to extend the set of states with one special non-accepting state *f* and extend the transition function with transitions violating the rule, ending up in state *f*. Thus, no catching *wait* statement would be necessary and no breach of a rule would be recognized until the global check at the end of model execution. However, the first approach is more suitable, because it can lower the time necessary for Zing model checker to detect a mistake in the verified model. It assures that violation of the rule is reported as soon as it is certain that the automaton cannot get to an accepting state any more. Detection of such violating state requires exploring of (often dramatically) lesser state space than checking of automata at the end of model execution.

After the transition, it remains to update actual state of automaton. This is done by rewriting the old state with the new one in the array containing current states of all automata (Example 6, section 8). It can happen that an automaton moves to a non-accepting state and later it is deleted from the heap together with the instance of its parent class. This means that the rule represented by the automaton is broken, however it is not reported at the end of automaton's lifetime. An instance of the parent class can be deleted e.g. when the scope of its declaration is exited. Quite a complex mechanism would be required to recognize this moment, because Zing does not support destructors. The solution of this issue is to retain the last state of the automaton in the array even after its deletion. The fact that this state is non-accepting is recognized and reported at the end of model execution.

If generated automata were non-deterministic, i.e. transition function was defined by T' : $S \times \Sigma \rightarrow P(S)$,[7] the transition routine would have to be slightly modified. Bodies of *wait* statements would be extended with *choose* statement, which would non-deterministically select a new state from the appropriate set. In this case, use of the extended alphabet and merging of transitions is not recommended, since it becomes quite confusing.

[7] P(S) denotes power set of S.

5.10.5. States of Rule Automaton

The implementation of automata and mechanism for keeping track of its state requires declaration of several fields and data structures in various places of the model.

Firstly, there are some fields needed to determine the letter parameter of transition routine within the transition method (computation of this parameter from root expressions was described in section 5.10.3). The persistent storage for values of root expressions is provided by fields in the parent class of the automaton. For evaluating the root expressions, variables holding values of their subexpressions are needed. Instead of declaring local variables in every call of the transition method, additional fields are generated in the parent class, even if persistence is not required in this case.

Secondly, current state of an automaton must be maintained for providing the state parameter of its transition function. This state must be accessible from the transition routine itself as well as from the place of final check of automata's states in *Model.Main* method. Moreover, the last state of an automaton must be available even if the automaton is deleted before the final check. These requirements implies that a data structure containing information necessary for the final check and for keeping track of automata's states must be declared in static *Model* class.

This data structure must contain current (or last) states of all automata instantiated during model execution for the needs of both transition routine and final check. Final check also requires information about non-accepting states for every instantiated automaton. Expressed in C#, the data structure could have following form:

```
Dictionary<AutomatonID, Pair<int, List<int>>>
```

Example 7: C# collection for automata states

AutomatonID denotes any type of key used for access from transition method. Retrieved dictionary value contains two items, the first being current state of the automaton and the second being a list of all non-accepting states of the automaton. The transition method uses only the first item. Key of the dictionary is not used in the final check, because it is necessary to enumerate all the items in the dictionary and only their values are needed.

As Zing does not support generic collections, an implementation using only Zing arrays must be generated in the model:

```
class Model
{
   static ArrayList_int_ __automataStates;
   static ArrayList_int_ __automataNonTerminalsCounts;
   static ArrayList_int_ __automataNonTerminalsStarts;
   static ArrayList_int_ __automataNonTerminals;

   ... //other members
}
```

```
...
array InnerArrayint[] int;
```

Example 8: Zing arrays for automata states

*ArrayList_int_* type denotes a class that represents integer instance of DeSpec *ArrayList* built-in template, being basically an extended wrapper of Zing *InnerArrayint* array (declared out of *Model* class).

The first array *__automataStates* contains current states of automata. The array *__automataNonTerminals* contains lists of non-accepting states of all automata. The lists are sequentially serialized in the same order as the states of corresponding automata in *__automataStates* array. To recognize where the lists for individual automata begin and end, two supplementary arrays are needed. The array *__automataNonTerminalsStarts* contains indices of beginnings of these lists in *__automataNonTerminals* and the array *__automataNonTerminalsCounts* contains lengths of these lists. With information retrieved from the two supplementary arrays, it is possible to effectively iterate through the lists of non-accepting states, which are stored in *__automataNonTerminals* array.

As the described arrays are used for keeping track of instantiated automata, they are filled gradually during the model execution. When a new automaton is instantiated, it must be registered in these fields. Its initial state and information about its list of non-accepting states is added to the corresponding arrays and an index pointing to these values is assigned to the automaton. This index simulates function of *AutomatonID* key for C# dictionary from Example 7 and is stored in another field of automaton's parent class.

5.11. Method Models

The key goal of DeSpec language is to allow specification and verification of requirements imposed on Windows drivers in form of rules and constraints. Most of mechanisms that support these DeSpec features are implemented in model's methods. They also require declaration of some supporting classes and enumerations, but most of the work related to verification is done by transition methods of automata and by compiler-generated code inserted into methods specified in the model.

5.11.1. Method Pattern

The original patterns for extending DeSpec methods are described in [2]. Based on these patterns and using the same notation, the one reflecting currently implemented features is stated below. A pattern for extending synthetic methods[8] is quite simple, as it involves only insertion of preconditions and postconditions. A pattern for extending driver methods relies on analysis and extraction of driver source code, which is to be provided by another tool. Until it is available, it is necessary to do its work manually. Since the rest of the pattern is similar to the one for kernel methods[9], it suffices to describe just the model of extended kernel methods:

[8] Synthetic methods have no counterpart in kernel or driver code and they cannot be involved in rules.
[9] The only difference is that method body is placed out of the atomic block.

```
<return-type> <name>(<arguments>)
{
   <return-type> result;
   assert(<conjunction of preconditions>);
   atomic
   {
      //method event triggering for interested automata
      <trigger-enter-event>;
      ...
      //extracted method body with modified returns
      <method-body>

   }
   //label for redirecting returns from method-body
   __returning:
   atomic
   {
      //method event triggering for interested automata
      if (IsSuccessful(result))
         <trigger-successful-event>;
         ...
      else
         <trigger-failed-event>;
         ...

      <trigger-returned-event>;
      ...

      assert(<conjunction of postconditions>);

      return result;
   }
}
```

Example 9: Pattern for kernel method model

Original methods of the model are adapted to this pattern by the *ZingFinalizingVisitor*. Most of the code was already prepared by during processing of the model and this visitor just retrieves prepared segments and the original method body, supplies code common for all methods and assembles all parts to match the pattern. The same process is applied to properties, because they will be transformed to methods as well.

Firstly, *result* variable must be declared. The declaration must be at the beginning of the method, because using of *result keyword* in specification is backed by this auto-generated variable.

After that, it is necessary to check conformance with constraints expressed in preconditions. This is done by asserting the conjunction of all preconditions. Constraints imposed on arguments via non-nullity checks, ranges, etc. were added to preconditions and postconditions by *PostTypeSemanticsVisitor*.

In contrary with the original pattern, snapshots of variables from *old* operators and blocking pre- and postconditions are not supported. However, since *old* operator is only syntactic sugar, it can be easily avoided in specifications.

As the final body of the method is appended with an atomic *postblock* with postconditions and method events, it is necessary to assure that this block is always executed. It means that every *return* statement in original method body must be redirected

to the atomic postblock. Thus, every *return* statement is replaced by *ZingFinalizingVisitor* with storing the return value into *result* variable (if applicable) and with jump to *__returning* label leading to the postblock. The return statement at the end of the postblock is the only one in the final method. During finalizing methods, the *__returning* labels are inserted before every postblock. It is not mandatory to finish all branches of an original DeSpec method with return statement, because a catching return is always added during the extraction of the model.

Segments of code triggering method events are prepared during implementation of rules and they are stored in *RuleImplementingVisitor*. A data structure holding these segments has a form of a dictionary keyed by methods, which must include the triggering code. The value retrieved from the dictionary is a pair of lists, each of them containing calls of transition methods of interested automata. The first list contains calls for *entered* event operator and the other one contains calls for the rest of the method event operators. Calls corresponding to *succeeded* and *failed* operators are already correctly conditioned.

5.11.2. Zing Limitations

There are other tasks to be done by *ZingFinalizingVisitor* even if they are not directly related to modeling methods. Nevertheless, this phase of model extraction is the most suitable place for them. These tasks involve transformation of some expression, which Zing does not understand, propagation of enumerations out of classes and modification of factories.

DeSpec boolean expressions can use $\Rightarrow$ *(implies)* operator, which is not supported by Zing. As $A \Rightarrow B$ expression is equivalent to $\neg A \vee B$, all expressions using $\Rightarrow$ operator are transformed appropriately.

Expression with *is* operator are not understood by Zing, because inheritance is not supported. Even if the value of a strongly-typed variable can be assigned to an *object* variable and vice versa, there is no built-in mechanism for retrieving the type of the value stored in the *object* variable at runtime. When an invalid typecast via assignment of a strongly-typed value is attempted, a runtime error is reported by Zing. The mechanism for determining the type is generated during implementation of inheritance and is described in section 5.9. It includes declaration of two typecasting methods in every class involved in inheritance – *upcast* and *downcast*. If one of them returns a non-null value, the queried typecast is possible, otherwise it is invalid.

Thus, every boolean expression with *is* operator matching following pattern:

```
<variable> is <typename>
```

can be replaced with

```
<variable>.downcast(<typename>) != null &&
<variable>.upcast(<typename>) != null
```

expression. More precisely, <*typename*> argument must be turned into an appropriate item from *inheriting classes* enumeration[10] to match the signatures of *downcast* and *upcast* method.

Using of *this* keyword in DeSpec factory methods has specific semantics. In contrary to Zing and common object-oriented languages, the value accessed via *this* keyword is not read-only. In context of factories, *this* represents a newly created object, which is returned by a method either as its return value or as its output parameter. Thus, keyword *this* refers to a variable access according to type of the particular factory. In case of a factory returning its product as the return value, *this* is equivalent to access to *result* variable. When a factory returns its product via the output parameter[11], it is equivalent to access to *instance* argument. *ZingFinalizingVisitor* replaces *this* access with the access to the corresponding variable. Factory methods are declared as instance method, because use of *this* in static methods would be misleading. Nevertheless, semantics of factories implies that these methods are actually static and must be marked as such during the visit of *ZingFinalizingVisitor* to enable their invocation during model execution.

Another limitation of Zing is related to declarations of enumerations. Enumerations can be declared only at global scope. Contrary of Zing, DeSpec grammar allows to including enumerations in class declarations. These enumerations must be moved outside the classes and their names must be mangled to show, where they belong. The same mangled names are used for updating references to original enumeration declarations. Everything stated above holds for range declarations as well.

5.11.3. Initialization in Entry Point

Model.Main method has a specific role in the model and requires special extension. This method is the only one marked with *activate* Zing modifier and thus it represents a single entry point of the extracted model. This means that initialization of whole model, which is not caught by the specification, must be done at the beginning of *Main* method. Segment of initialization code is generated during the visit of *ZingFinalizingVisitor* by its *InsertPrologue* method.

Firstly, a new instance of *Thread* class is created. This object represents a parent thread of whole model. All other threads are created only if specified in the model via *async* statement. Thread static data included in these objects are available in method bodies through DeSpec *thread* keyword. This keyword is actually transformed to a reference to hidden *thread* parameter, which is passed to every invoked method.

Secondly, static constructors fall some classes are called. It would be possible to determine the classes, whose static members are accessed in the model and invoke only

[10] See section 5.9.3

[11] According to DeSpec languge specification, such a factory must have exactly one output parameter named *instance*.

their static constructors. However, this analysis is not implemented and static initialization is preformed for all classes that contain non-empty static constructors. Thus, an access to all static members of all classes in the model is assured.

5.11.4. Checking Rules before Termination

The other important extension of Main method takes place at its end. If no violation of a rule is detected during model execution, this is the place where the verification finishes. Not all breaches can be recognized during the lifetime of an automaton. Thus it is necessary to check, whether its last state was accepting or not. This check could be performed at the time of automaton deletion, but it is quite difficult to recognize this moment. Since the last states of all automata remain stored till the end of model execution, it suffices to perform the check at that time.

A segment of code for checking the last states of automata is generated in InsertEpilogue method of ZingFinalizingVisitor. It checks for every registered automaton, whether its last state is included on the set of its non-accepting states and if so, an assertion is violated and the verification fails. Zing implementation of the algorithm is following:

```
int __i = 0;
int __count = __automataStates.CountGet();
while( __i < __count)
{
   int __from;
   __from = __automataNonTerminalsStarts[__i];
   int __to;
   __to = __from + __automataNonTerminalsCounts[__i];
   int __j = __from;
   while(__j < __to)
   {
         assert(__automataStates[__i] != __automataNonTerminals[__j],
               "rule broken");
         __j = __j + 1;
   }
   __i = __i + 1
}
```

Example 10: Final automata check

This check is possible thanks to arrays declared during implementation of rules. Their structure was described in detail in section 5.10.5.

5.11.5. Transformation of Expressions into Statements

The last issue that is related to modeling methods and translating them to Zing is its restriction set on expressions. Especially use of *assignment expressions* and *invocation expressions* is limited. Some fragments of Zing grammar that cause these limitations are listed below. The rules are taken from [5] and some of them are expanded to a specific form, which points to the restriction. In such cases, expansion is marked by ellipsis:

statement:
    labeled-statement
    declaration-statement
    embedded-statement

declaration-statement:
    ...
    type identifier = expression;

embedded-statement:
    ...
    invocation-expression;

expression:
    conditional-or-expression
    assignment

conditional-or-expression:
    ...
    primary-expression

boolean-expression:
    expression

assignment:
    unary-expression = expression
    unary-expression = invocation-expression

Example 11: Zing rules for expressions

One of the significant limitations is the fact, that *invocation-expression* can be transcribed neither to *primary-expression* nor to *expression*. In contrary, DeSpec allows *invocation-expressions* both in *boolean-expressions* and in initializer of *declaration-statements*. From Example 11, it is apparent that this is not possible in Zing. As a result, *invocation-expression* can appear only on the right side of *assignment* or in an *expression-statement* alone. Since properties will be eventually transformed into methods, this restriction holds for them too.

Moreover, even if *element-access* can be transcribed to *primary-expression* according to Zing grammatical rules, it is treated as an invocation by Zing compiler.

Similarly, despite of the fact that choose-expression can be syntactically transcribed to *primary-expression*, Zing limits its use only to right operand of an assignment.

Since constructors are not supported by Zing at all, DeSpec *new-expressions* must be implemented by Zing *object-creation-expression* and accompanied with an invocation of DeSpec constructor as an initialization routine. Thus, the replacement must be moved out of the place of original *new-expression*.

These issues must be solved by replacing the critical expressions with Zing-acceptable equivalents.

ForbiddenExpressionVisitor visits nodes representing an invocation, property access, element access, new-expression and choose-expression. It recognizes the context of the expression and if it is not valid for Zing, it replaces the expression with an auto-generated variable of the same type. When returning from the parent statement of the expression that is being replaced, the visitor inserts additional statements before the parent one. The first one is a declaration of the local variable, which is used as the replacement of the invalid expression. The second one is an assignment to this variable, its right side being the expression invalid in its original context. This solution is possible thanks to the fact that all problematic expressions can stand on the right side of an assignment.

When replacing *new-expression*, the original expression cannot be simply assigned to the variable. Constructors are not supported and objects are instantiated via *object-creation-expression* without parameters. To assure that the body of appropriate constructor is executed, constructors are transformed into void returning methods. They are actually degraded to initializing routines. By executing Zing instantiation and the initialization consecutively, the intended functionality of DeSpec constructor is provided.

It remains to discuss the equality of original DeSpec code and generated Zing replacement. The first problem emerges from possible side-effects. For example, if the critical expression is included as a parameter (not the first one) of an invocation statement, its evaluation can count on a side-effect caused by the computation of a previous parameter. As the evaluation of the critical expression is moved before the side-effect, its result can be different. Another, less significant problem is interleaving of threads. When the critical expression is enclosed in an atomic block, it is obvious that neither the original DeSpec statement containing the expression nor its replacement with the list of described statements can be interleaved with execution of another segment of the model. On the other hand, if the critical expression appears in the body of a driver method or a method marked with *non-atomic* attribute, interleaving can take place. An interleaved thread cannot access the newly created variable holding the value of the critical expression, as this variable is not referenced outside of the method. However, the interleaved thread can change some accessible data that are used for evaluation of the expression. This is not a serious problem, because a correct model cannot rely on a specific intersection of threads.

Paradoxically, the same limitations of Zing, which require this problematic workaround, eliminate the problem with side-effects. Pre- and postincrements and pre- and postdecrements are supported neither by Zing nor in DeSpec. The original DeSpec grammar includes these expressions, but they are not implemented yet. If they were implemented, they would belong to the other critical expression and they would be treated uniformly. All other expressions, which can cause side-effects, are already included in the critical ones. Thus, all possible sources of side-effects are moved before the original statement and their order is preserved. Correct order of the evaluation of the critical expressions is guaranteed by the replacing algorithm.

When *ForbiddenExpressionVisitor* traverses the AST and recognizes a critical expression, it sets a replacing flag and until finishing the replacement of this expression, no other replacement can be started. As the replacement is finished when the visitor is leaving the parent statement of the critical expression, only one replacement per statement can be made during one traversing of the AST. If some statement contains more critical expressions, *ForbiddenExpressionVisitor* must be run on the AST several times, till there are expressions to replace. Thus it is assured, that critical expressions are propagated before

the statement in the same order as they would be evaluated in the statement, if Zing supported them.

One minor issue is related to the replacing algorithm. The statements, which are generated by the algorithm and are to be inserted before the parent statement of the critical expression, are actually enclosed together with the parent one in a newly created block statement. This block statement is returned when the visitor is leaving the parent statement. This original statement is then replaced by the new block. Enclosing in a block does not change the meaning of the replacement, it only makes the modification of the parent method easier. However, if the parent statement is a local declaration, it cannot be nested in this block, because it would become hidden within its original scope. When returning from a block determining the scope of some local variables, the statements from the generated blocks are moved to the original one.

The conclusion is that the replacement of a critical expression with a block of statements is functionally equal to the intended effect of the original DeSpec expression. Nevertheless, because of thread interleaving, which can occur if the replacement is not enclosed in an atomic block, the state space of resulting model can be larger then expected.

A context, which makes an expression critical, is described in detail for the individual expression types

An *invocation expression* does not become critical, if it is used in an expression statement, just to denote an invocation of the method. It can also be used as a right side of an assignment statement. If the parent assignment is nested in another expression or in local declaration statement, the invocation expression becomes critical, as well as in any other case. These limitations are apparent from Example 11, where the only two allowed occurrences of the invocation expression are listed.

The only place, where *element access* does not become critical, is an assignment statement. Zing compiler treats element access similarly as invocation expression, however it has no sense as a standalone statement and on the other hand, it can appear on both sides of assignment statements.

A *choose-expression* becomes critical anywhere except the right side of an assignment statement. This restriction is not expressed by Zing grammar, but it is explicitly stated in Zing language specification.

A *member access* denoting getting a property value is transformed into an invocation expression later during model extraction. Thus, all limitations stated for invocations hold for these expressions too. For a *member access* denoting setting a property value, the situation is different. DeSpec semantics requires this access to appear only on left side of assignment expressions. When eliminating properties in the model, these assignment expressions are turned into method invocations, with their right sides turned into arguments. Thus, whole parent assignment expression must be treated as a future invocation expression. Appropriate limitations must be applied on this parent expression, not on the member access itself. This expression becomes critical in one more special case. In DeSpec it is possible to assign a property to a property (*setter-access = getter-access*). After property elimination, this would result into following invocation expression – *propertyNameSet(propertyNameGet())*, which is critical. As property elimination takes place after the processing by *ForbiddenEpressionVisitor*, this critical expression would not be replaced. Thus it is necessary to recognize this pattern even before the elimination and treat it as critical too.

Since a *new expression* must be always replaced with separate instantiation and initialization, it is critical in any context.

DeSpec ternary conditional operator *?:* nor is not supported, but its implementation could use a similar mechanism. Expressions using this operator would be treated as critical and they would be transformed into Zing supported statements. The value of the variable, which would replace such expression, would have to be determined in *if-statement*.

5.11.6. Emitting Zing Code

After traversing the AST by *ForbiddenEpressionVisitor*, the extraction of the model is almost done. Most of the model is described by constructs common for both DeSpec and Zing and it is prepared for representation in Zing. Remnants of DeSpec-specific code will be translated to Zing "on-the-fly", in specific dumping routines.

Emitting of Zing code is performed by dumping methods, which are declared in every AST node, since they are inherited from *Node* base class. For nodes representing DeSpec-only constructs, these methods are empty.

For most of the nodes, *Dump* method simply takes an instance of *TextWriter* passed as the argument and appends it with serialized Zing representation of the node. At the end, the instance of *TextWriter* contains a string with Zing code of the model and writes it into a selected file. However, some of the nodes use their *Dump* methods for non-trivial transformation of their content to Zing. Most important of these transformations is elimination of properties and generating built-in collection classes.

Since Zing does not support properties, they must be replaced by methods. When dumping method of a parent class calls for dump of a property declaration, the appropriate *Dump* actually emits a declaration of a special method for corresponding getter and setter, if provided.

Signatures of these methods have following forms:

<*property-type*> <*property-name*>`Get( Thread thread )`

for the getter and

`void` <*property-name*>`Set( Thread thread,` <*property-type*> `value )`

for the setter.

For getters and setters of indexed properties, signatures are

```
<property-type> <property-name>Get( Thread thread,
                                    <index-type> <index-name> )
```

and

```
void <property-name>Set( Thread thread, <index-type> <index-name>,
                         <property-type> value )
```

respectively.

Access to original properties is transformed into invocation expressions in three types of nodes – *member access, element access* and *assignment expression*. In member access node, references to getters of original common properties are simply transformed into <*property-name*>`Get(thread)` calls. In element access node, references to getters of original indexed properties are transformed into <*property-name*>`Get(thread, index)` calls. Finally, in assignment expression node, assignments to setters of original properties are replaced with <*property-name*>`Set(thread,` <*right-expression*>`)` calls for common properties and <*property-name*>`Set(thread, index,` <*right-expression*>`)` calls for indexed properties.

Whereas the process of dumping was chosen for transformation of properties just because of the effectiveness and convenience, for implementing the built-in collections it is the only possibility.

Templates for built-in collections are included in every specification; however they cannot specify the access to underlying arrays on necessary level of detail. To abstract higher-level collection templates like *ArrayList* or *Queue* from necessary implementation details, *Array* template is included in DeSpec specifications. During the model extraction, it serves as a proxy to a Zing array, which will eventually replace the instances of Array[12] providing desired functionality in the resulting Zing model. Thus it is possible to represent collections by DeSpec constructs during the model extraction. In the end, when generating Zing representation of the AST, DeSpec *Arrays* are replaced with Zing arrays. This transformation cannot be done earlier in the extracting process, because it is not possible to represent Zing-specific code in AST nodes.

However, such exploiting of DeSpec *Array* template prevents it from being used in specifications as common built-in collection template. A proxy to Zing array serves entirely compiler's needs and cannot appear in Zing representation of the model. However this limitation can be easily overcome by using DeSpec *ArrayList* template on places, where *Array* would be used. This template provides the same interface as *Array*, extended with methods for adding and removing items. When these additional methods are not used, it represents an equivalent replacement of *Array* template from the point of DeSpec specifications. Still, if *Array* template were required for purposes of specification, it would be possible to use it, provided that another proxy to Zing array would be created.

[12] Instances of a built-in template denote classes derived from this template with specified type argument. For example *ArrayList*<*int*> class is an instance of *ArrayList* template.

When dumping of an instance of *ArrayList* template, Zing declaration of underlying array type is prepared and initializer of *items* field[13] is modified. The original instantiation of DeSpec instance of *Array* template is turned into an instantiation of the prepared Zing array type. The original *size* argument is used. References to *items* are preserved and element accesses into this member remain valid. The only expression including *items*, which must be changed, is accessing *items.Count* property, because this expression is invalid in Zing. Occurrences of *items.Count* expression are replaced with Zing operator *sizeof* with *items* passed as the argument. The prepared declaration of Zing array is dumped out of the generated class as required by Zing.

[13] This field represents underlying instance of *Array* template during the model extraction.

6. Open Problems and Further Work

The compiler in the current state of development does not support all features of DeSpec language. The reason is partly absence of the tools necessary for complete model extraction and partly complicated implementation of some of DeSpec constructs. The unimplemented features are described below.

The limitations are given above all by the absence of tools for C source code analysis and for slicing of the model. The analysis of kernel header files is necessary for extraction of symbols used in the specification. For example values of modeled constants and enumerations must be retrieved. The analysis of driver source code is more complicated, since it is required for automatic merging of DeSpec specification with the bodies of modeled driver functions. Not only mapping of DeSpec method declarations to driver functions must be set, but also bodies of the functions must be analyzed. Once the parser and analyzer of C source code are implemented, all sources for completion of the model will be available.

For determining relevant code of both DeSpec specification and supplied driver and kernel, a tool performing a slicing algorithm is necessary. With this tool it will be possible to select one of defined namespaces and thus determine the level of detail of DeSpec specification. Specific constraints and rules to verify will be selected. Based in these inputs, reduction of the model will be performed, effectively lowering the state space of resulting model as well as the time necessary for model extraction. If the slicing algorithm is applied on the model once more at the end of the extraction, its state space can be further reduced.

Implementation of these tools is beyond the scope of this thesis and it is the key task for further development. Other unsupported features depend on their outputs.

One of these features is DeSpec *delegate* concept. This construct is designed for modeling callbacks from *ILateBoundDriverRoutines* interface passed to the kernel by the driver. Since Zing has no notion of function pointers, a mechanism for calling referenced functions must be provided. The implementation is proposed in [2] and involves declaration of a specific class for every delegate in the specification. This class keeps track of where appropriate function pointers refer to and provides dispatch to the target methods. To enable such functionality, it is necessary among others to determine all possible targets of the particular delegate. This requires a pointer-to analysis of driver C source code.

Another issue related to function pointers is mapping of driver functions from *IEarlyBoundDriverRoutines* interface. Since these function bindings do not change during driver's execution, *delegates* are not required for their modeling. Rather, less complicated mapping exploiting DeSpec attributes is used. Attributes *EarlyBound* and *EarlyBoundOpt* applied on a field, which represents the pointer to a driver function, accept an argument with name of the function model. Since pointers to functions from *IEarlyBoundDriverRoutines* interface are passed to kernel in *DRIVER_OBJECT* and *DRIVER_EXTENSION* structures, fields in DeSpec models of these structures are marked with *EarlyBound*/*EarlyBoundOpt* attributes with names of appropriate function models.

For reflecting the bindings stored in the kernel structures in specification, outputs of driver source code analysis are needed. Without C source code analyzer, *EarlyBound* and

EarlyBoundOpt attributes are not supported. *Anonymous* attribute, which says that marked structure or union has no name in the C declaration, is also relevant only for this analyzer.

Other attributes require the slicing tool for their implementation. Both *Conditional* and *CheckConstraints* attributes are relevant only for determining, which parts of the model are to be extracted.

Selection of the namespace to verify is not supported so far. It means that only the model specified in default namespace is checked, as only its *Main* method is marked as the entry point of the model. Models specified in other namespaces are correctly processed and translated, but they are not included in the resulting model.

Several features of rules, which were proposed in DeSpec language specification, are not implemented. The most important of them are *access event operators*, which are applicable on properties and fields, full-fledged use of parameters in method events, arbitrary quantification variables and use of synthetic members in rule expressions. There is also no support for *ThreadBoundEvents* attribute, which assures that events are watched separately in the context of each relevant thread. Unsupported rule patterns and other minor limitations were described in section 5.10.

One of the main issues related to implementation of rules is quite loose syntax and even semantics of rule expressions. DeSpec grammar actually allows using any valid expression in the rule. This obviously allows creating many syntactically correct rules that have no sense. Due to lack of syntactic restrictions, semantic control of rule expressions is quite complicated. It could be beneficial to reconsider, whether it would be possible to create more accurate grammar rules for the rule expressions, or eventually, whether some more suitable means for specifying them could be found.

With current grammar rules, it is necessary to recognize the context of expression and treat it differently, if it is included in a rule. DeSpec event operators are designed exclusively for using in rule expressions and thus actually a special language is defined just for them. It is possible that if this language would not be just extension of DeSpec constructs for expressions, but also some limitations would be defined and expressed by a special grammar, a better control over rule expressions could be gained on syntax level.

The verification of unions is not supported. It means that unions can be used in the specification, but their correct behavior is not checked. Correct behavior of unions is that only the last written field is read from. Verification of this rule requires a discriminator field, which maintains information about the last written field of the union.

Original DeSpec *Set* built-in collection template is not supported. As this template relies on Zing *set* type, an approach to its implementation should be analogous to the one used for *Array* template and its higher-level derivatives.

Other unsupported features are actually syntactic sugar and they do not lower the expressive power of implemented subset of DeSpec in comparison with original version. However, these constructs should be implemented to make writing specifications more convenient and straightforward. The most significant limitation is missing support for *extension*. Extension is a mean of code reuse and is not reflected in the resulting model. It cannot be eliminated simply during preprocessing, because it requires information about

types. Main issues of its implementation are related to incomplete type information at the time of elimination and to loose rules for its combination with inheritance. Other minor restrictions on the usage of syntactic sugar are apparent from the modified grammar.

The usage of some DeSpec constructs, which were mentioned above, is limited or even forbidden usually due to the missing implementation in the back-end if the compiler. However, in most of the cases, the classes for corresponding AST node types are prepared for their full-fledged usage and usually also the semantic analysis takes them into account.

7. Related Work

The compiler from DeSpec to Zing is intended to be a part of a framework for formal verification of Windows driver environment. In a broader context, any work addressing model extraction for further model checking is related to the thesis. Some tools that cover model extraction and model checking are Bogor framework [12], Spin [13], Java PathFinder [14] and also Zing compiler and model checker [4], [5].

As for the compiler itself, there are no other tools for translating or analyzing DeSpec. DeSpec language is inspired in particular by Zing modeling language and some of its constructs for expressing requirements on models are inspired by the Spec# language[15] and Spec temporal patterns [3]. Gauss project [16] involves translation of a model to Zing language, however the input is an MPI program written in C.

Main goal of the verification framework based on DeSpec is the verification of Windows drivers. With respect to this fact, Static Driver Verifier (SDV) [17] is the closest work, as it has a similar goal. However, SDV extracts the model directly from the C source code and requirements and constraints are stated in SLIC language [18], which is much less expressive than DeSpec. The mechanism of verification is different from the one used by Zing model checker (i.e. exploring of model's state space). SDV uses a predicate discoverer and a theorem prover for generating potential error traces and analyses, whether these traces can occur during the execution of the driver. In most of the cases, modeling of kernel interaction with the driver is degraded to yielding non-deterministically chosen return values and output parameters. This can cause acceptation of a trace as correct even if the driver incorrectly relies on a value that was generated non-deterministically. A more detailed comparison of SDV and DeSpec can be found in [2].

As there is no other tool capable of extraction of the model from a DeSpec specification, contribution of the compiler to modeling Windows driver environment by DeSpec is apparent.

8. Conclusion

The thesis has introduced a tool for extracting Zing models from DeSpec specifications of Windows driver environment. Thus, it allows formal verification of these models by Zing model checker.

For the model extraction from a complete and full-fledged specification, it is necessary to implement other tools, namely an extractor of C source code of kernel and driver and a tool for slicing and reduction of the models. Development of these tools is beyond the scope of the thesis.

Under these conditions, main focus was given to the implementation of DeSpec analyzer, as the semantic analysis is crucial for further steps of the extraction. The analyzer, which is represented by the front-end of the introduced compiler, supports most of the features of DeSpec language. Unsupported constructs represent syntactic sugar and their absence does not reduce expressivity of DeSpec.

The back-end of the compiler represents an attempt to prove that extraction of Zing models from DeSpec is possible, rather than a full-featured implementation. Since all key features of DeSpec, like rules, constraints, groups or inheritance, are at least partially implemented, this attempt can be claimed successful.

Since the tools for analysis of C source code and slicing the model are not yet available, extraction of Zing model from DeSpec specification is limited. A specification can contain only constructs from the supported subset of the DeSpec language. Extraction of C source code is done manually and necessary symbols and driver function bodies are incorporated in the specification. If reduction of state space is desired, slicing must be performed manually by removing irrelevant parts. When a specification matching these requirements is passed as an input, the compiler produces an equivalent Zing model, which can be transformed into the executable form by Zing compiler. The resulting model then can be verified by Zing model checker. Thus, the main goal of the thesis is accomplished.

The usage of the compiler is not bound only to the specifications of the Windows driver environment. The compiler can create Zing model from the specification of any environment that defines some interface and interacts with plugins, providing that it can be described with the supported subset of DeSpec.

The successful though not full-fledged implementation of the compiler proves that the DeSpec language is well designed and that it is suitable for creating the specifications of real-world environments, which can be further analyzed and formally verified.

9. References

[1] Microsoft: Windows Driver Kit, WHDC, http://www.microsoft.com/whdc/DevTools/WDK/default.mspx

[2] Matoušek, T.: Model of the Windows Driver Environment, 2005

[3] Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in Property Specifications for Finite-state Verification, Proceedings of the 21st International Conference on Software Engineering 1999

[4] Andrews, T., Qadeer, S., Rajamani, S. K., Rehof, J., Xie, Y., Zing: A Model Checker for Concurrent Software, Microsoft Research Technical Report, 2004

[5] Microsoft: Zing Language Specification, Microsoft Research, 2005

[6] Hatcliff, J., Dwyer, M.B., Zheng, H.: Slicing Software for Model Construction, Journal of Higher-order and Symbolic Computation 13(4), 1999

[7] Aho, A.V, Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986

[8] Jones, J.: Abstract Syntax Tree Implementation Idioms, The 10th Conference on Pattern Languages of Programs, 2003

[9] Gough, J., Kelly, W.: The GPPG Parser Generator, PLAS, 2007

[10] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1998

[11] Adámek, J., Kofroň, J., Plášil, F.: Lectures on Behavior Models and Verification, Lecture 6, DSRG MFF CUNI, 2006

[12] Robby, Dwyer, M.B., Hatcliff, J.: Bogor Software Model Checking Framework, http://bogor.projects.cis.ksu.edu

[13] Bell Labs: Spin, http://spinroot.com

[14] Robust Software Engineering Group, NASA ARC: Java PathFinder, http://javapathfinder.sourceforge.net

[15] Barnett, M., Rustan, K., Leino, M., Schulte, W.: The Spec# programming system: An overview, Springer, 2004

[16] Palmer, R., Barrus, S., Yang, Y., Gopalakrishnan, G., Kirby, R.M.: Gauss: A Framework for Verifying Scientific Computing Software,' Workshop on Software Model Checking, Edinburgh, 2005

List of Appendices

A. DeSpec Grammar

The grammar stated below is based on the original DeSpec grammar published in [2] and reflects the modifications made during the development of the compiler.

A.1. Tokens

Although not all keywords introduced in the original DeSpec grammar are used in the current version, they remain reserved. This is necessary to avoid conflicts in existing specifications, when the currently missing features are implemented. The definition of `Identifier` is modified and forbids the usage of double underscore (__) prefix. This prefix is reserved for needs of the compiler. The definition of the other literals is not changed.

This is the unchanged list of keywords:

```
absent          entered         max             short
abstract        enum            min             static
abstracts       error           namespace       string
after           executes        new             struct
and             exists          notice          succeeded
any             extends         null            synthetic
anytimes        extracted       object          this
as              failed          old             thread
assert          false           otherwise       timeout
assume          first           out             to
async           flags           precedes        true
atomic          forall          raise           try
base            foreach         range           uint
before          forever         read            ulong
between         get             readonly        union
bool            globally        ref             universal
break           goto            refines         until
byte            group           requires        ushort
class           choose          responds        using
const           if              result          value
correspond      in              return          void
s               instance        returned        wait
delegate        int             rule            warning
else            is              sbyte           while
end             leads           select          with
ensures         long            set             written
```

A.2. Production Rules

The production rules are stated in Backus-Naur Form. Non-terminals are marked by acute brackets and terminals are named or marked by single quotes. The empty right sides of the rules are marked by `!empty` comment. The starting symbol is `<Specification>`.

A.2.1. Global Declarations

```
<Specification> ::= <GlobalDecls>

<GlobalDecls> ::= <GlobalDecls> <GlobalDecl>
                | !empty
```

```
<GlobalDecl> ::= <SpecDecl>
               | <Namespace>

<SpecDecl> ::= <ClassDecl>
             | <EnumDecl>
             | <RangeDecl>
             | <Using>

<SpecDecls> ::= <SpecDecls> <SpecDecl>
              | !empty

<Namespace> ::= <AttributeList> 'namespace' <Refines>
                '{' <SpecDecls> '}'
              | <AttributeList> 'namespace' Identifier <Refines>
                '{' <SpecDecls> '}'

<Refines> ::= 'refines' Identifier
            | !empty

<ClassDecl> ::= <AttributesAndModifiers> 'class' Identifier <Inherits>
                '{' <MemberDecls> '}'

<Abstracts> ::= 'abstracts' <Type>
              | !empty

<Inherits> ::= ':' Identifier
             | !empty

<Using> ::= 'using' <QualifiedName> ';'
```

A.2.2. Types

```
<Types> ::= <Types> ',' <Type>
          | <Type>

<Type> ::= <PrimitiveOrRangeType>
         | <QualifiedName> <GenericParameters>

<PrimitiveOrRangeType> ::= <PrimitiveType>
                         | <RangeType>

<RangeType> ::= 'range' '(' <Range> ')'

<GenericParameters> ::= '<' <Types> '>'
                      | !empty

<QualifiedName> ::= Identifier
                  | 'any'
                  | <QualifiedName> '.' Identifier
                  | <QualifiedName> '.' 'any'

<QualifiedNamesEx> ::= <QualifiedNamesEx> ',' <QualifiedName>
                     | <QualifiedNamesEx> ',' '!' <QualifiedName>
                     | <QualifiedName>

<IntegerPrimitiveType> ::= 'sbyte' | 'byte'
```

```
                        | 'short' | 'ushort'
                        | 'int'   | 'uint'
                        | 'long'  | 'ulong'

<ReferencePrimitiveType> ::= 'string'

<PrimitiveType> ::= <IntegerPrimitiveType>
                  | <ReferencePrimitiveType>
                  | 'bool'

<Literal> ::= 'null' | 'true' | 'false' | StringLiteral | IntLiteral |
HexLiteral
```

A.2.3. Modifiers and Attributes

```
<Modifier> ::= 'static'
             | 'synthetic'
             | 'abstract'
             | 'base'
             | 'readonly'

<Modifiers> ::= <Modifiers> <Modifier>
              | <Modifier>

<AttributesAndModifiers> ::= <Attributes> <Modifiers>
                           | <Attributes>
                           | <Modifiers>
                           | !empty

<AttributeList> ::= <Attributes>
                  | !empty

<Attributes> ::= <Attributes> <Attribute>
               | <Attribute>

<Attribute> ::= '[' <Expression> ']'
```

A.2.4. Members and Inner Declarations

```
<MemberDecls> ::= <MemberDecls> <MemberDecl>
                | !empty

<MemberDecl> ::= <FieldDecl>
               | <MethodDecl>
               | <StructDecl>
               | <EnumDecl>
               | <RangeDecl>
               | <RuleDecl>
               | <PropertyDecl>
               | <GroupDecl>

<FieldDecl> ::= <AttributesAndModifiers> <Type> <NullitySpec>
                <FieldVariableList> ';'
              | <AttributesAndModifiers> 'const' <Type>
                <FieldVariableList> ';'
```

```
<FieldVariableList> ::= <FieldVariableList> ',' <FieldVariable>
                      | <FieldVariable>

<FieldVariable> ::= Identifier
                  | Identifier '=' <Expression>

<PropertyDecl>   ::=   <AttributesAndModifiers>   <Type>   <NullitySpec>
Identifier
                    '{' <PropertyAccessors> '}'
                  |   <AttributesAndModifiers>   <Type>   <NullitySpec>
Identifier
                    '[' <Type> Identifier ']' '{' <PropertyAccessors>
'}'
                  |   <AttributesAndModifiers>   <Type>   <NullitySpec>
'this'
                    '[' <Type> Identifier ']' '{' <PropertyAccessors>
'}'
                  | <AttributesAndModifiers> '_' Identifier
                    '{' <PropertyAccessors> '}'
                  | <AttributesAndModifiers> '_' Identifier
                    '[' <Type> Identifier ']' '{' <PropertyAccessors>
'}'

<PropertyAccessors> ::= <PropertyGetter> <PropertySetter>
                      | <PropertySetter> <PropertyGetter>
                      | <PropertyGetter>
                      | <PropertySetter>

<PropertyGetter>          ::=          <AttributeList>          'get'
<ContractDeclListOrSemicolon>
<PropertySetter>          ::=          <AttributeList>          'set'
<ContractDeclListOrSemicolon>

<MethodDecl> ::= <AttributesAndModifiers> <MethodSignature>
                 <ContractDeclListOrSemicolon>
               | <AttributesAndModifiers> <CtorSignature>
                 <ContractDeclListOrSemicolon>

<MethodSignature> ::= <Type> <NullitySpec> Identifier '
                      (' <ParameterList> ')'
                    | 'void' Identifier '(' <ParameterList> ')'
                    | '_' Identifier '(' <ParameterList> ')'
                    | 'instance' Identifier '(' <ParameterList> ')'

<CtorSignature> ::= Identifier '(' <ParameterList> ')'

<NullitySpec> ::= '!'
                | !empty

<ContractDeclListOrSemicolon> ::= <ContractDeclList> <MethodBody>
                                | ';'

<ContractDeclList> ::= <ContractDeclList> <ContractDecl>
                     | !empty

<ContractDecl> ::= 'requires' <Expression> ';'
                 | 'ensures' <Expression>  ';'

<MethodBody> ::= <Block>
               | !empty
```

```
<GroupDecl> ::= <AttributesAndModifiers> 'group' <MethodSignature> '='
                '{' <QualifiedNamesEx> '}' ';'

<StructDecl> ::= <AttributesAndModifiers> <StructOrUnion> Identifier
                 '{' <FieldOrStructDecls> '}'

<FieldOrStructDecls> ::= <FieldOrStructDecls> <FieldDecl>
                       | <FieldOrStructDecls> <StructDecl>
                       | <FieldOrStructDecls> <PropertyDecl>
                       | !empty

<StructOrUnion> ::= 'struct'
                  | 'union'

<EnumDecl> ::= <AttributesAndModifiers> <EnumOrflags> Identifier
               <Abstracts> '{' <EnumFieldDeclList> '}'

<EnumOrflags> ::= 'enum'
                | 'flags'

<EnumFieldDeclList> ::= <EnumFieldDecls>
                      | !empty

<EnumFieldDecls> ::= <EnumFieldDecls> ',' <EnumVariable>
                   | <EnumVariable>

<EnumVariable> ::= <FieldVariable>
                 | Identifier '=' <Range>
                 | Identifier '=' '{' <RangeItemList> '}'

<Range> ::= <IntLiteral> '..' <IntLiteral>
          | <IntLiteral> '..' <HexLiteral>
          | <HexLiteral> '..' <IntLiteral>
          | <HexLiteral> '..' <HexLiteral>

<RangeItemList> ::= <RangeItemList> ',' <RangeItem>
                  | <RangeItem>

<RangeItem> ::= <Range>
              | <Expression>

<RangeDecl>    ::=    <AttributesAndModifiers>    'range'    Identifier
<Abstracts> '='
                <Range> ';'
```

A.2.5. Rules

```
<RuleDecl> ::= <AttributesAndModifiers> 'rule' <RuleSpecification> ';'
             |     <AttributesAndModifiers>     'rule'     Identifier
<RuleSpecificationList> ';'

<RuleSpecificationList>      ::=      <RuleSpecificationList>      ','
<RuleSpecification>
                           | <RuleSpecification>

<RuleSpecification> ::= <Quantification> <RulePattern>
```

```
<Quantification> ::= 'forall' '(' <QuantifiedVariableList> ')'
                   | !empty

<QuantifiedVariableList>     ::=     <QuantifiedVariableList>     ','
<QuantifiedVariable>
                           | <QuantifiedVariable>

<QuantifiedVariable> ::= <Type> Identifier
```

A.2.6. Temporal Patterns

```
<RulePattern> ::= <RuleExpression> 'is' 'universal' <RuleScope>
                | <RuleExpression> 'is' 'absent' <RuleScope>
                | <RuleExpression> 'exists' <RuleScope>
                |    <RuleExpression>    'precedes'    <RuleExpression>
<RuleScope>
                |   <RuleExpression>   'leads'   'to'   <RuleExpression>
<RuleScope>
                |  <RuleExpression>  'responds'  'to'  <RuleExpression>
<RuleScope>

<RuleScope> ::= 'globally'
              | 'before' <RuleExpression>
              | 'after' <RuleExpression>

<RuleExpression> ::= '{' <Expression> '}'
```

A.2.7. Parameters and Arguments

```
<ParameterList> ::= <Parameters>
                  | !empty

<Parameters> ::= <Parameters> ',' <Parameter>
               | <Parameter>

<ArgumentList> ::= <Arguments>
                 | !empty

<Arguments> ::= <Arguments> ',' <Argument>
              | <Argument>

<Parameter>  ::=  <AttributeList>  <ParamModifier>  <Type>  <NullitySpec>
Identifier
              | <AttributeList> 'out' 'instance'
              | <AttributeList> 'instance'
              | '_'
              | '...'

<Argument> ::= <ParamModifier> <Expression>
             | '_'
             | '...'

<ParamModifier> ::= 'out' | !empty
```

A.2.8. Expressions

```
<PrimaryExpression> ::= <Literal>
                      | <SpecialVariableAccess>
                      | <ParenthesizedExpression>
                      | <InvocationExpression>
                      | <MemberAccessExpression>
                      | <ElementAccessExpression>
                      | <PostIncExpression>
                      | <PostDecExpression>
                      | <NewExpression>
                      | <ChooseExpression>

<ParenthesizedExpression> ::= '(' <Expression> ')'

<MemberAccessExpression> ::= Identifier
                           | 'any'
                           | <PrimaryExpression> '.' Identifier
                           | <PrimaryExpression> '.' 'any'

<ElementAccessExpression> ::= <PrimaryExpression> '[' <Expression> ']'

<InvocationExpression> ::= <MemberAccessExpression> '(' <ArgumentList>
')'
                         <InvocationEvent>

<SpecialVariableAccess> ::= 'thread' | 'this' | 'result' | 'value'

<NewExpression> ::= 'new' <Type> '(' <ArgumentList> ')'

<ChooseExpression> ::= <ChooseConstruct> '(' 'bool' ')'
                     | <ChooseConstruct> '(' <Expression> ')'

<ChooseConstruct> ::= 'choose'

<Expressions> ::= <Expressions> ',' <Expression>
                | <Expression>


<InvocationEvent> ::= '::' 'succeeded'
                    | '::' 'failed'
                    | '::' 'entered'
                    | '::' 'returned'
                    | !empty

<UnaryExpression> ::= <PrimaryExpression>
                    | '+' <UnaryExpression>
                    | '-' <UnaryExpression>
                    | '!' <UnaryExpression>
                    | '~' <UnaryExpression>

<MulExpression> ::= <MulExpression> '*' <UnaryExpression>
                  | <MulExpression> '/' <UnaryExpression>
                  | <MulExpression> '%' <UnaryExpression>
                  | <UnaryExpression>

<AddExpression> ::= <AddExpression> '+' <MulExpression>
                  | <AddExpression> '-' <MulExpression>
                  | <MulExpression>
```

```
<ShiftExpression> ::= <ShiftExpression> '<<' <AddExpression>
                    | <ShiftExpression> '>>' <AddExpression>
                    | <AddExpression>

<RelExpression> ::= <RelExpression> '<' <ShiftExpression>
                  | <RelExpression> '>' <ShiftExpression>
                  | <RelExpression> '<=' <ShiftExpression>
                  | <RelExpression> '>=' <ShiftExpression>
                  | <RelExpression> 'is' <QualifiedName>
                  | <RelExpression> 'as' <QualifiedName>
                  | <ShiftExpression>

<EquExpression> ::= <EquExpression> '==' <RelExpression>
                  | <EquExpression> '!=' <RelExpression>
                  | <EquExpression> '===' <RelExpression>
                  | <RelExpression>

<BitAndExpression> ::= <BitAndExpression> '&' <EquExpression>
                     | <EquExpression>

<BitXorExpression> ::= <BitXorExpression> '^' <BitAndExpression>
                     | <BitAndExpression>

<BitOrExpression> ::= <BitOrExpression> '|' <BitXorExpression>
                    | <BitXorExpression>

<AndExpression> ::= <AndExpression> '&&' <BitOrExpression>
                  | <BitOrExpression>

<OrExpression> ::= <OrExpression> '||' <AndExpression>
                 | <AndExpression>

<ImpliesExpression> ::= <ImpliesExpression> '==>' <OrExpression>
                      | <OrExpression>

<ConditionalExpression> ::= <ImpliesExpression>

<AssignmentExpression>  ::=  <PrimaryExpression>  <AssignmentOperator>
<Expression>

<AssignmentOperator> ::= '=' | '+=' | '-=' | '*=' | '/=' | '%='
                       | '&=' | '|=' | '^=' | '>>=' | '<<='

<Expression> ::= <AssignmentExpression>
               | <ConditionalExpression>

<StatementExpression> ::= <AssignmentExpression>
                        | <InvocationExpression>
```

A.2.9. Statements

```
<Block> ::= '{' <StatementList> '}'

<StatementList> ::= <StatementList> <Statement>
                  | !empty
```

```
<Statement> ::= <EmbeddedStatement>
              | <LocalDeclStatement>
              | <LabelStatement>

<EmbeddedStatement> ::= ';'
                      | <ExpressionStatement>
                      | <ReturnStatement>
                      | <GotoStatement>
                      | <IfStatement>
                      | <LoopStatement>
                      | <SelectStatement>
                      | <AtomicStatement>
                      | <AssertStatement>
                      | <AssumeStatement>
                      | <AsyncStatement>
                      | <TryWithStatement>
                      | <RaiseStatement>
                      | <Block>

<ExpressionStatement> ::= <StatementExpression> ';'

<ReturnStatement> ::= 'return' <Expression> ';'
                    | 'return' ';'

<GotoStatement> ::= 'goto' Identifier ';'

<IfStatement> ::= 'if' '(' <Expression> ')' <EmbeddedStatement>
                | 'if' '(' <Expression> ')' <EmbeddedStatement> 'else'
                  <EmbeddedStatement>

<LoopStatement> ::= 'while' '(' <Expression> ')'
                      <EmbeddedStatement>
                  | 'foreach' '(' <Type> Identifier 'in' <Expression>
')'
                    <EmbeddedStatement>

<SelectStatement> ::= 'select' <SelectQualifiers> '{' <WaitStatements>
'}'

<SelectQualifiers> ::= <SelectQualifiers> <SelectQualifier>
                     | !empty

<SelectQualifier> ::= 'end'
                    | 'first'

<WaitStatements> ::= <WaitStatements> <WaitStatement>
                   | <WaitStatement>

<WaitStatement>    ::=    'wait'    '('    <Expression>    ')'    '->'
<EmbeddedStatement>
                  | 'timeout' '->' <EmbeddedStatement>

<AtomicStatement> ::= 'atomic' <Block>

<AssertStatement> ::= 'assert' '(' <Expression> ')' ';'
                    | 'assert' '(' <Expression> ',' StringLiteral ')'
';'
```

```
<AssumeStatement> ::= 'assume' '(' <Expression> ')' ';'

<LabelStatement> ::= Identifier ':'

<AsyncStatement>    ::=    'async'    <InvocationExpression>    'with'
<Expression> ';'

<LocalDeclStatement> ::= <PrimitiveOrRangeType> Identifier ';'
                       | <PrimitiveOrRangeType> Identifier '='
                         <Expression> ';'
                       |                          <MemberAccessExpression>
<MemberAccessExpression>
                         Identifier ';'
                       |                          <MemberAccessExpression>
<MemberAccessExpression>
                         Identifier '=' <Expression> ';'

<TryWithStatement> ::= 'try' <Block> 'with' '{' <WithClauses> '}'

<WithClauses> ::= <WithClauses> <WithClause>
                | <WithClause>

<WithClause> ::= Identifier '->' <EmbeddedStatement>
               | 'any' '->' <EmbeddedStatement>

<RaiseStatement> ::= 'raise' Identifier ';'
```

B. Sample Specification

This appendix contains a simplified yet representative specification of a class from the sample model of the driver environment and the corresponding Zing model generated by the compiler. The sample class contains a rule and several constraints. Since the class is involved in inheritance, the simplified specification of its parent class is also included.

To make the sample shorter and better readable, only selected class members are included in the specification. As for the Zing model, only some of its interesting parts are included and the compiler's output is formatted. The complete sample specification, which is derived from the specification published in [2], as well as complete translated model can be found on the accompanying CD.

B.1. DeSpec Class Declaration

```
// Event dispatcher object
class KEVENT : DispatcherObject
{

   // Whether event is auto-reset when a wait function succeeds on it
   synthetic bool AutoReset;

   void KeInitializeEvent(instance,EVENT_TYPE type,bool signals)
   {...}


   // Sets event to a signaled state
   int KeSetEvent(instance,_,bool doWait)
         requires thread.Irql <= KIRQL.DISPATCH_LEVEL;
         requires doWait ==> (thread.Irql == KIRQL.PASSIVE_LEVEL);
   {...}

   // Sets event to a non-signaled state
   int KeResetEvent(instance)
         requires thread.Irql <= KIRQL.DISPATCH_LEVEL;
   {...}

   // Sets event to a signaled state
   void KeClearEvent(instance)
         requires thread.Irql <= KIRQL.DISPATCH_LEVEL;
   {...}

   // Gets the current state of the event
   int KeReadStateEvent(instance)
         requires thread.Irql <= KIRQL.DIRQL;
         requires thread.Irql <= KIRQL.DISPATCH_LEVEL;
   {...}



   // Checks whether the event is initialized before used
   rule
         { KeInitializeEvent(_,_)::returned }
         precedes
         {     KeSetEvent(_)::returned ||
```

```
                KeResetEvent()::returned ||
                KeClearEvent()::returned ||
                KeReadStateEvent()::returned
        }
        globally;
    }
```

B.2. Zing Model

```
class KEVENT
{
    bool AutoReset;
    // binding to the instance of the parent class
    DispatcherObject __parent;

    int __automatonIndex_lambda1;
    bool __ruleExpressionValue_0_lambda1;
    bool __ruleExpressionValue_1_lambda1;
    bool __action_0_lambda1;
    bool __action_1_lambda1;
    bool __action_2_lambda1;
    bool __action_3_lambda1;
    bool __action_4_lambda1;

    // delegation to parent property
    bool SignalsGet(Thread thread){
        bool result;
        atomic{{{
            (result = __parent.SignalsGet(thread));
            goto __returning;
        }}}
        __returning:
        atomic {return result;}
    }

    //delegation to parent property
    void SignalsSet(Thread thread, bool value){
        atomic{{
            (__parent.SignalsSet(thread, value));
        }}
        __returning:
        atomic {return ;}
    }

    void KeInitializeEvent( Thread thread,
                            KEVENT_EVENT_TYPE type, bool signals){
        atomic{{
            (AutoReset =
                (type ==
            KEVENT_EVENT_TYPE.SynchronizationEvent));
            (SignalsSet(thread, signals));
            Initialized(thread);
        }}
        __returning:
        atomic{
            // triggering ::returned event
            __Step_lambda1(thread,
                            KEVENT___Actions_lambda1.KeInitializeEvent,
                            __Events.Returned);
            return ;
        }
    }
```

```
    int KeSetEvent(Thread thread, bool doWait){
          int result;
          // preconditions
          assert(((true && (
                thread.Irql <= KIRQL.DISPATCH_LEVEL)) &&
                (!(doWait) || (thread.Irql == KIRQL.PASSIVE_LEVEL))));
          atomic
          {{(SignalsSet(thread, true));}}
          __returning:
          atomic{
// triggering ::returned event
                __Step_lambda1(thread,
                            KEVENT___Actions_lambda1.KeSetEvent,
                            __Events.Returned);
                return result;
          }
    }

    // type converting routine
    object __upcast(Thread thread, __Classes typeName){
          object result;
          atomic{{
                if((typeName == __Classes.Default_KEVENT)){
                      (result = this);
                      goto __returning;
                }
                else{
                      (result = __parent.__upcast(thread,
                                                  __Classes.Default_KEVENT))
                                        ;
                      goto __returning;
                }
          }}
          __returning:
          atomic {return result;}
    }

// type converting routine
    object __downcast(Thread thread, __Classes typeName)
    {
          object result;
          atomic{{
                object cast;
                if((typeName == __Classes.Default_KEVENT)){
                      (result = this);
                      goto __returning;
                }
                {
                      (result = null);
                      goto __returning;
                }
          }}
          __returning:
          atomic {return result;}
    }

    // as operator replacement
    object __as(Thread thread, __Classes typeName)
    {
          object result;
          atomic{{
                (result = this.__upcast(thread,typeName));
                if((result == null))
                      (result = this.__downcast(thread,typeName));
```

```
            }}
            __returning:
            atomic {return result;}
      }

      void KEVENTCtor(Thread thread){
            atomic{{
                  {
                        // linking into the inheritance chain
                        DispatcherObject __tmp16;
                        (__tmp16 = new DispatcherObject);
                        __tmp16.DispatcherObjectCtor(thread);
                        (__parent = __tmp16);
                  }

                  (__parent.__child_Default_KEVENT = this);
                  (this.AutoReset = false);

                  // registration of the automaton
                  Model.__automataStates.Add(thread,0);
                  (__automatonIndex_lambda1 =
                              Model.__automataStates.CountGet(thread));
                  (__automatonIndex_lambda1 = (__automatonIndex_lambda1 -
1));
                  int __start_lambda1;
                  (__start_lambda1 =
                              Model.__automataNonTerminals.CountGet(thread
                              ));
                  Model.__automataNonTerminalsStarts.Add(thread,
                                                      __start_lambda1);
                  Model.__automataNonTerminalsCounts.Add(thread,0);
                  Model.__automataStates.Add(thread,0);
                  (__automatonIndex_lambda6 =
                                    Model.__automataStates.CountGet(thread
                              ));
                  (__automatonIndex_lambda6 = (__automatonIndex_lambda6 -
1));
                  int __start_lambda6;
                  (__start_lambda6 =
                              Model.__automataNonTerminals.CountGet(thread
                        ));
                  Model.__automataNonTerminalsStarts.Add(thread,
                                                      __start_lambda6);
                  Model.__automataNonTerminalsCounts.Add(thread,0);
            }}
            __returning:
            atomic {return ;}
      }

   // transition method of the rule automaton
      void __Step_lambda1(Thread thread,
                        KEVENT___Actions_lambda1 action,
                        __Events _event){
                  // determine the event
                  if(false)
                        {;}
                  else if((action ==
                              KEVENT___Actions_lambda1.KeInitializeEvent))
                        {(__action_0_lambda1 =
                              (_event == __Events.Returned));}
                  else if((action ==
KEVENT___Actions_lambda1.KeSetEvent))
                        {(__action_1_lambda1 =
                              (_event == __Events.Returned));}
```

```
                  else if((action ==
KEVENT___Actions_lambda1.KeResetEvent))
                        {(__action_2_lambda1 =
                              (_event == __Events.Returned));}
                  else if((action ==
KEVENT___Actions_lambda1.KeClearEvent))
                        {(__action_3_lambda1 =
                              (_event == __Events.Returned));}
                  else if((action ==
                              KEVENT___Actions_lambda1.KeReadStateEvent))
                        {(__action_4_lambda1 =
                              (_event == __Events.Returned));}

                  // check if value of a rule expression changed
                  bool new__ruleExpressionValue_0_lambda1 =
                                                      __action_0_lambda1;
                  bool new__ruleExpressionValue_1_lambda1 =
                              (((__action_1_lambda1 || __action_2_lambda1)
                              || __action_3_lambda1) ||
                              __action_4_lambda1);
                  bool change =
                              ((__ruleExpressionValue_1_lambda1 !=
                              new__ruleExpressionValue_1_lambda1)
                              ||
                              ((__ruleExpressionValue_0_lambda1 !=
                              new__ruleExpressionValue_0_lambda1)
                              || false));
                  if(change)
                  {
                        (__ruleExpressionValue_0_lambda1 =
                                          new__ruleExpressionValue_0_lambd
                                    a1);
                        (__ruleExpressionValue_1_lambda1 =
                                          new__ruleExpressionValue_1_lambd
                                    a1);
                  }
                  else
                        return ;

                  // perform the transition
                  bool v0 = __ruleExpressionValue_0_lambda1;
                  bool v1 = __ruleExpressionValue_1_lambda1;
                  int state;
                  (state = Model.__automataStates.thisGet(thread,
                                                __automatonIndex_lambda1))
                                                ;
                  select first{
                        wait(((state == 0) && (!(v0) && !(v1)))) -> ;
                        wait(((state == 0) && v0)) -> (state = 1);
                        wait((state == 1)) -> ;
                        wait(true) -> assert(false, "rule broken");
                  }
                  (Model.__automataStates.thisSet(thread,
                                          __automatonIndex_lambda1,state))
                                          ;
            }
      }
};
```