UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

SALT^{XT}: An Xtext-based Extendable Temporal Logic Compiler

SALT^{XT}: Ein auf Xtext basierender erweiterbarer Temporallogikcompiler

Bachelorarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Sebastian Hungerecker

ausgegeben und betreut von
Prof. Dr. Martin Leucker

mit Unterstützung von
Normann Decker

Lübeck, den 20. August 2014

Abstract

SALT is a high-level temporal logical specification language that facilitates the writing of specifications that describe the behavior of complex systems. This thesis describes the implementation of SALT^{XT}, a new compiler and Eclipse plug-in for SALT.

SALT^{XT} is a new implementation of SALT that has been designed to be extendable and easily deployable. It also includes an Eclipse plug-in that offers, for the first time, IDE support for creating SALT specifications.

Kurzzusammenfassung

SALT ist eine high-level temporal-logische Spezifikationssprache, die das Schreiben von Spezifikationen erleichtert, die komplexe Systeme spezifizieren. Diese Arbeit beschreibt die Implementierung von SALT^{XT}, einem neuen Compiler und Eclipse-Plug-in für SALT.

SALT^{XT} ist eine neue Implementierung der SALT Sprache und wurde entworfen, um erweiterbar und einfach zu installieren zu sein. Es enthält auch ein Eclipse-Plug-in, das IDE-Unterstützung dafür bietet, um SALT-Spezifikationen zu schreiben – etwas, was es bisher noch nicht gab.

Declaration

I hereby declare that I produced this thesis without external assistance, and that no other than the listed references have been used as sources of information.

Lübeck, August 20, 2014

Acknowledgements

I want to thank Normann Decker and Prof. Leucker. I also want to thank Timo Brinkmann for helping me with the graphics and Lukas Schmidt for proofreading.

Contents

1 Introduction

Linear Temporal Logic

Linear temporal logic (LTL) [Pnueli, 1977] is a logic that extends propositional logic to allow reasoning about time. LTL views time as a series of separate states – that is it views time discretely, not fluently. As in most logics, formulas in LTL make statements over a set of atomic propositions ("variables"). In LTL each atomic proposition can either be true of false during any given state. So, given the atomic propositions a and b LTL formulas can express statements like "a is true in the current state and b will be true in the next state".

In the following let $w, i \models \varphi$ mean that the temporal logical formula φ holds true for the sequence of states w at the position i. We say that w satisfies φ iff φ is true for w at position i.

In addition to atomic propositions (the variables over which we make statements) LTL consists of the constants $\top$ (true) and $\bot$ (false) and the following operators (where φ and ψ stand for arbitrary LTL formulas):

Propositional Operators

The following operators are equivalent to the corresponding operators in propositional logic.

$\neg$ **("not")** $w, i \models \neg\varphi \Leftrightarrow w, i \not\models \varphi$

$\wedge$ **("and")** $w, i \models \varphi \wedge \psi \Leftrightarrow (w, i \models \varphi) \wedge (w, i \models \psi)$

$\vee$ **("or")** $w, i \models \varphi \vee \psi \Leftrightarrow (w, i \models \varphi) \vee (w, i \models \psi)$

$\rightarrow$ **("implies")** $w, i \models \varphi \rightarrow \psi \Leftrightarrow (w, i \not\models \varphi) \vee (w, i \models \psi)$

$\leftrightarrow$ **("if and only if")** $w, i \models \varphi \leftrightarrow \psi \Leftrightarrow (w, i \models \varphi) \wedge (w, i \models \psi) \vee (w, i \not\models \varphi) \wedge (w, i \not\models \psi)$

Future Operators

These operators make statements that are not just about the state i (i.e. the "current" or "present" state), but about states $j > i$ (i.e. "future" states).

X (or $\bigcirc$ – "next") $w, i \models \mathsf{X}\,\varphi \Leftrightarrow w, i+1 \models \varphi$

U ("until") $w, i \models \varphi\,\mathsf{U}\,\psi \Leftrightarrow \exists j \geq i : (w, j \models \psi) \wedge \forall k | i \leq k < j : w, k \models \varphi$

F (or $\Diamond$ – "eventually") $w, i \models \mathsf{F}\,\varphi \Leftrightarrow \exists j \geq i : w, j \models \varphi$

G (or $\Box$ – "globally" or "always") $w, i \models \mathsf{G}\,\varphi \Leftrightarrow \forall j \geq i : w, j \models \varphi$

W ("weak until") $w, i \models \varphi\,\mathsf{W}\,\psi \Leftrightarrow w, i \models \varphi\,\mathsf{U}\,\psi \vee \mathsf{G}\,\varphi$

Past Operators

LTL can also be extended with past operators that are similar to the future operators, but make statements about previous states rather than future states. Past operators do not add expressive power to LTL, but allow writing some formulas more succinctly [Gabbay et al., 1980].

Y ("previous") $w, i \models \mathsf{Y}\,\varphi \Leftrightarrow i > 0 \wedge w, i-1 \models \varphi$

S ("since") $w, i \models \varphi\,\mathsf{S}\,\psi \Leftrightarrow \exists j \leq i : (w, j \models \psi) \wedge \forall k | i \geq k > j : w, k \models \varphi$

O ("once") $w, i \models \mathsf{O}\,\varphi \Leftrightarrow \exists j \leq i : w, j \models \varphi$

H ("historically") $w, i \models \mathsf{H}\,\varphi \Leftrightarrow \forall j \leq i : w, j \models \varphi$

B ("weak since" or "back to") $w, i \models \varphi\,\mathsf{B}\,\psi \Leftrightarrow w, i \models \varphi\,\mathsf{S}\,\psi \vee \mathsf{H}\,\varphi$

Smart Assertion Language for Temporal Logic

Linear temporal logic is a powerful tool to specify the properties of various types of components – be they computer programs that are verified using runtime verification tools or model checkers or hardware components that are checked using model checkers. However linear temporal logic is a rather low-level way of writing specifications. It only offers a small set of core operators and no means of abstraction that can be used to structure large specifications or to avoid repetition. This can make it hard to write, read, debug and maintain large specifications and easy to make mistakes in them.

It is therefore desirable to have a higher-level language that has the same expressive power as linear temporal language and can be used with the same tools, but at the same time offers a greater set of operators, a more easily readable syntax and means of abstraction that make it possible to easily write large specifications that are still readable and maintainable.

Some such languages (e.g. Sugar/PSL [Accellera, 2004] and For-Spec [Armoni et al., 2002]) have been designed for the hardware domain, but are not applicable to other domains because they have been designed with hardware verification in mind and as frontends for proprietary verification tools.

SALT, which is short for "Smart Assertion Language for Temporal Logic", is a general purpose high-level temporal logic that has been inspired by Sugar/PSL, but can be used to write specifications in any domain. It was proposed in [Bauer et al., 2006] and first implemented by Jonathan Streit in [Streit, 2006]. It offers a greatly expanded set of operators – all of which have English, rather than symbolic, names for greater readability – and the ability to define one's own operators to facilitate code reuse, maintainability and readability. All temporal operators in SALT have past operator counter-parts that make statements about the past rather than the future – like past operators for LTL.

It also offers looping constructs to make assertions over a set of expressions, further facilitating code reuse and concise code. To enable SALT specifications to be used with existing model checking and runtime verification tools, SALT can be compiled to the linear temporal logic dialects supported by those tools.

A SALT specification might look like this:

```
-- sometimes_but_not_always(x) holds if x holds at least
-- once at some point, but not always
define sometimes_but_not_always(x) :=
    eventually x and not (always x)

assert x implies eventually y

assert sometimes_but_not_always z

assert
  allof enumerate [1..3] with i in
    motor_$i$_used implies motor_$i$_has_power
```

```
-- The event e may not happen more than 5 times
assert holding [ <= 5 ] e
```

In LTL those assertions would be written like this:

- $x \rightarrow \mathsf{F}\, y$
- $\mathsf{F}\, x \wedge \neg\, \mathsf{G}\, x$
- $(\texttt{motor\_1\_used} \rightarrow \texttt{motor\_1\_has\_power}) \wedge (\texttt{motor\_2\_used} \rightarrow \texttt{motor\_2\_has\_power}) \wedge (\texttt{motor\_3\_used} \rightarrow \texttt{motor\_3\_has\_power})$
- $\neg\, (\mathsf{F}\, (e \wedge (\mathsf{X}\, (\mathsf{F}\, (e \wedge (\mathsf{X}\, (\mathsf{F}\, (e \wedge (\mathsf{X}\, (\mathsf{F}\, (e \wedge (\mathsf{X}\, (\mathsf{F}\, e)))))))))))))$

The expressive power of SALT is equal to that of LTL, so SALT specifications can be compiled to LTL-based formats that are understood by commonly used model checkers and runtime verification tools.

Contribution

The first implementation of SALT, implemented by Jonathan Streit in [Streit, 2006], is a Java application that, given a SALT specification, generates Haskell code that generates a specification in the specified LTL-based output format, then compiles that Haskell code, runs the resulting executable and produces the executable's output as its result.

As a result of this approach the compiler needs a Haskell compiler and runtime environment to be installed in addition to a Java runtime environment. It also requires some initial configuration to make the SALT compiler aware of where it can find the Haskell compiler. Another complication is that the generated Haskell code requires old versions of Haskell packages that are not compatible with the versions that ship with current Haskell compilers. This means that a potential user of SALT would either have to find and install an old Haskell compiler or expand some effort into installing old versions of packages and making them work with a current compiler. This makes installing the SALT compiler a rather complicated endeavour.

The SALT compiler has a web interface that requires no installation, so this is not as much of an issue as it might otherwise be, but SALT users might still reasonably want to install the compiler on their own PCs for offline access or to be able to use the SALT compiler as part of a tool chain. For example a user might want to have a shell script that invokes the SALT compiler to compile a SALT specification to an LTL specification in SMV syntax and then feeds that specification into SMV. This cannot be done using the web interface unless the user writes a web scraper to access the web interface programmatically. Further the installation complications still apply when deploying the compiler on the server whenever the web interface is moved to a new server.

There is also currently no tool support for SALT. That is no IDEs or text editors with any kind of support for SALT; nor are there any other kinds of tools that help with writing, navigating, debugging or refactoring SALT specifications. The only tools are the compiler itself and the web interface. The web interface only offers a plain text box into which the specification can be written without any SALT-specific features like syntax highlighting. Since features like syntax highlighting, automatic completion of operator and variable names and as-you-type error detection greatly improve programmer productivity, the lack of any such features for SALT is a problem.

We therefore implemented the SALT^{XT} compiler, which is a new compiler for the SALT language. It is extendable through a flexible plug-in system. New output formats, optimizations or new translation strategies for existing output formats can all be implemented through translation phase plug-ins.

Translation phase plug-ins can also have arbitrary requirements. Translation phase plug-ins with requirements can only be used when compiling a specification that meets those requirements. For example some plug-ins may require that a specification only uses a specific subset of features or that all used variables meet a given naming scheme. Any decidable property of a specification can be used as such a requirement.

Additionally SALT^{XT} also supports domain specification plug-ins that allow plug-in authors to provide special support for writing specifications in a specific domain. Like translation phase plug-ins, domain specification plug-ins also allow to express arbitrary requirements on specifications, but unlike translation phase plug-ins, the plug-in will report error messages (defined by the plug-in) when a requirement is not met, rather than becoming unavailable. Thus domain specification plug-ins can use domain knowledge to verify properties of a specification other than the specification just being a syntactically valid SALT specification. They can also be used to generate warnings, rather than errors.

Domain specification plug-ins can also perform arbitrary transformations on a specification before it is translated by the translation phase plug-ins. This allows plug-in writers to create domain specific notations that simplify writing specifications for the given domain, but that would not make sense for specifications in any other domain.

The SALT^{XT} compiler is written in Java and Xtend, a Java-like JVM language that comes with the Xtext framework. It can be distributed as a single JAR file that has no external dependencies other than a Java runtime environment.

SALT^{XT} comes with an Eclipse plug-in, that provides many common IDE features for writing SALT specifications. These features include syntax highlighting, automatic completion of operator and macro names, on the fly error detection, renaming macros, jumping to macro definitions and integration of the compiler into the IDE.

Outline

This thesis will first describe the fundamentals of compiler design in chapter 2 to provide the information needed to understand the workings of the SALT^{XT} compiler. It will then give an overview of existing integrated development environments and their features in chapter 3 to motivate the choice to achieve tool support for SALT through an Eclipse plug-in and to provide a context for evaluating the features of the SALT^{XT} Eclipse plug-in. Chaoter 4 will describe the Xtext framework that has been used to implement SALT^{XT}. Next, chapter 5 will summarize the syntax and semantics of the SALT language. Chapter 6 will describe the design and implementation of the SALT^{XT} compiler and its plug-in system. It will also describe how to extend the compiler's functionality through the plug-in system. It will also describe some ways in which the SALT language implemented by the SALT^{XT} compiler differs from the SALT language as described in [Streit, 2006] and [Bauer et al., 2006] and explain why those changes were made. Chapter 7 will describe the SALT^{XT} Eclipse plug-in and the features it provides for writing SALT specifications. Last, chapter 8 will summarize the contents of this thesis and suggest future work that could be done to build on this thesis and further enhance the SALT^{XT} compiler and Eclipse plug-in. Additionally the appendices will give a continuous EBNF grammar of the SALT language and a complete listing of the the SALT operators.

2 Compilers

Compilers are tools that translate code written in high-level languages into lower-level representations with the same semantics. Compilers are generally used to translate programming languages into machine code or bytecode that can be executed by a virtual machine. However they can also be used to translate other kinds of languages into formats that can be used by certain programs. For example the LaTeX compiler translates code in the LaTeX language into binary formats understood by document viewers. Likewise a compiler can translate a logical program specification written in a high-level logic language like SALT into a lower-level format – like SPIN or SMV – that can be understood by a model checker or runtime verification tool, which is, in fact, what the SALT^{XT} compiler does.

This chapter will explain the benefits and drawbacks of compilation compared to other ways of implementing languages, describe how compilers work and explain techniques that are often used in compilers. Those techniques are explained in detail in [Aho et al., 1986] and used by real-world compilers such as [clang, 2013] or [gcc, 2013].

2.1 Alternatives to Compilation

When implementing a language, implementers have two choices: they can either write a tool that understands the language directly or write a compiler to a lower-level language for which tools already exist. In the case of programming language this means deciding between an interpreter that executes the language directly and a compiler that translates the language to machine code or another programming language for which implementations already exist. In case of logical specification languages the equivalent choice would be between a model checker or runtime verifier that understands the language and a compiler that translates into a format that is understood by existing model checkers or runtime verifiers.

For programming languages the choice of writing an interpreter, rather than a compiler, for a high-level language can be a viable option in some cases. However the situation is more clear-cut when implementing a high-level logical specification language like SALT:

For such a language the equivalent of writing an interpreter, i.e. writing a tool that directly "executes" the source code, rather than just translating it to something else, would

be to write a model checker or runtime verifier that directly accepts this format. Since those are very complex tools, writing a compiler is a lot less effort than implementing a full model checker or runtime verifier would be. It also has the benefit that SALT can be used anywhere where LTL-based formats can be used, thus making SALT broadly applicable. The compiler could also be extended to support non-LTL based formats, broadening its applicability even more.

2.2 The Anatomy of a Compiler

2.2.1 Compilation Stages

The job of a compiler is to take a file written in a high-level language and translate it into a lower-level language. This job is usually accomplished through multiple stages:

Tokenization Tokenization or lexical analysis is the process of taking a sequence of characters and transforming it into a sequence of tokens. A token is an atomic sequence of characters, that is a sequence of characters that, for the purposes of compilation, can be considered as a single unit that cannot be taken apart any further. So for example an identifier like `name` would be a single token because we wouldn't need to ever examine any substring of it individually. However a variable declaration like `int x` would not be a single token because we need to be able to look at the type and the variable name individually. Every token is associated with a token type. Usually there is a token type for identifiers, one token type for each type of literal in the language (e.g. one token type for string literals, another for integer literals and so on) and one token type for each symbol and keyword in the language. Comments and whitespaces are generally removed during tokenization. For example a code fragment like `string s = "hello";` could be tokenized into the token sequence: `IDENTIFIER("string")`, `IDENTIFIER("s")`, `EQUALOP`, `STRING_LITERAL("hello")`, `SEMICOLON`.

It should be noted that some languages, e.g. [Python, 2013] and [Haskell, 2013], have significant indentation, that is the amount of whitespace at the beginning of a line can affect the meaning of a program in those languages. In those languages whitespace – or at least whitespace at the beginning of a line, i.e. indentation – will not be removed by the tokenizer as it contains relevant information. Instead there will be a special token type that represents indentation.

Token types can often be described through regular expressions and there are tools – for example lex and its free clone flex – that can generate code to perform lexical analysis from a list of regular expressions. The code generated by those tools will tokenize a string by checking which of the given regular expressions matches the longest prefix of the string, generating a token for that prefix using the token type to which the matched regular expression belongs and then tokenizing the remaining string in the same way. If

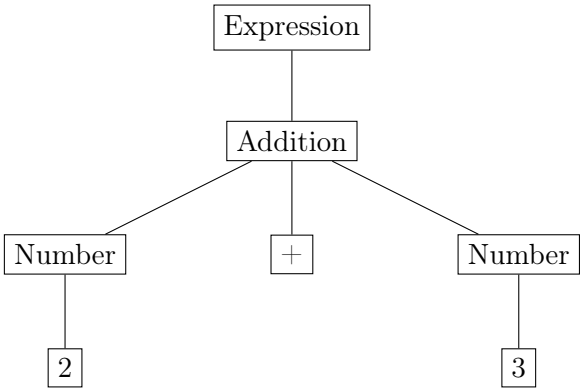
at any point none of the regular expressions match any prefix of the given string, an error message will be produced.

It is not strictly necessary for compilers to have a tokenization stage – it is entirely possible to perform the parsing stage on the input string directly. However separating the stages tends to be simpler and more efficient [Aho et al., 1986]. Tokenization can often be performed more efficiently than parsing: commonly used tokenization algorithms run in linear time with very little constant overhead when used with the kind of regular expressions that appear in practice (in the context of tokenization) – commonly used parsing algorithms also run in linear time, but with significantly larger constant overhead. There also exist algorithms that can perform tokenization in linear time for all regular expressions [Reps, 1998]. Therefore it is often more efficient to first tokenize the input and then perform the parsing stage on the tokenized input (which will contain less tokens than the original string contained characters because whitespace and comments are stripped and because each token generally corresponds to more than one character, thus decreasing the input size for the parsing stage) and that is indeed what most compilers do.

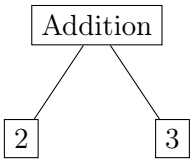
Parsing Parsing is the process of taking a sequence of tokens, verifying that these tokens make up a valid program (or specification, document etc.) in the given language, and processing the code depending on the syntactic structure of the code. Often the parsing code will generate a tree structure that represents the syntactic structure of the code. This tree can then be further processed by the subsequent compiler stages. It is however also possible for it to generate an intermediate representation of the code that is not a tree – for example 3-address code or any other form of bytecode – or to directly produce output in the target language without any intermediate stages (in which case the parsing stage would be the last stage of the compiler).

Like tokenization parsing code is often generated by tools – so called parser generators. Examples of such tools are YACC and ANTLR. These tools generate parsing code from a grammar file that describes the syntax of the language using some variation of context free grammars. Each production of the grammar will be annotated with information that tells the parser which code to execute when that production is used. This can be achieved by directly writing the code to be executed into the grammar file or, when the parser is used to build a tree structure, by simply annotating the productions with the type of node that should be generated for them (or whether a node should be generated for that production at all). Some tools even accept grammars without annotations and generate a tree that has one node per used production where each node's type is the name of the production. The tree generated by such a tool is called a syntax tree. Since a syntax tree generally contains a lot of redundant information, it is more useful to generate a so-called abstract syntax tree that only contains as many types of node as necessary.

For example a grammar might have a production like `expression ::= addition | number`. Using this production the parse tree of the expression `2 + 3` would be:



The abstract syntax tree for this expression would be:



Since abstract syntax trees are more compact and easier to use than parse trees, parse trees will often be converted to abstract syntax trees right away when using a parser generator that generates parse trees.

Desugaring Languages often contain syntactic constructs that could also be expressed in terms of existing constructs. For example many programming languages allow programmers to write `x += y` instead of `x = x + y`. Such syntactic shortcuts are referred to as "syntactic sugar". They are useful to programmers as they allow them to write more concise code. However they can complicate the job of the compiler:

Most stages of the compiler work by walking the tree structure representing the program and then executing different actions depending on which type of node is currently being visited. Adding new types of nodes will thus increase the number of cases that have to be handled in each stage. Since most types of syntactic sugar are only useful to the programmer and being able to distinguish between the shortcut and the expanded form is not useful for the compiler, it would be best if introducing new types of syntactic sugar would not add new types of nodes that have to be handled in each stage. In simple cases this can be achieved by making the parser generate a tree in which the syntactic shortcuts have already been replaced by their expanded forms. However in more complex cases it can be useful to perform such replacements in a separate stage to preserve separation of concerns. That is the parser would generate different nodes for syntactic shortcuts and an extra stage that runs directly after the parser would replace all shortcut nodes with nodes representing the expanded form. Subsequent stages would then no longer need to handle the shortcut nodes. Since the sole purpose of such a stage is the removal of syntactic sugar, such a stage is referred to as the "desugaring" stage.

Type Checking A compiler for a statically typed language will have a type checking stage. In this stage the compiler will verify that all expressions are valid according to the language's typing rules and will produce an error message when that is not the case. In many cases the type checker will also annotate each expression's node with the expression's type, so that later stages can simply read that information to find out an expression's type without performing any type checking themselves. This is useful because in statically typed languages, typing information is often necessary in later stages of compilation. For example the size of a variable can depend on its type in many languages and the code generation stage needs access to that information (some optimization stages might make use of that information as well).

Even languages that are not statically typed as such can have statically verifiable correctness properties. For example even if it is not possible to statically determine which type a given value has in a programming language, it might still be possible to determine whether a function or variable with a given name exists in the current scope and how many arguments a function accepts. So name errors (i.e. referring to a variable or function name that doesn't exist) and arity errors (i.e. calling a function with the wrong number of arguments) could still be detected statically in such a language. A compiler for such a language could thus have a stage akin to a type checking stage that detected such errors and rejected programs that contain them.

In the SALT language the macro system has a simple type system that can be statically checked. That is the compiler will reject specifications that call non-existent macros, call macros with the wrong number of arguments or call macros with logical expressions as arguments when the macro takes another macro as its argument – or vice versa.

High-Level Optimization Most compilers perform some optimizations on the programs that they compile. An optimization is a transformation that takes a representation of a program and modifies it in such a way that it still has the same semantics, but better time or space behavior. Optimizations can roughly be divided into high-level and low-level optimizations. A high-level optimization is one that can be performed on a program's tree representation without access to or knowledge of any details of any low-level formats that the program will be converted to in later cases.

The version of the SALT^{XT} compiler that is described in this document does not perform any optimizations – high-level or otherwise. However some possible optimizations (both high- and low-level) are described in appendix F of [Streit, 2006] and were implemented in the previous SALT compiler. These optimizations will likely be added to the SALT^{XT} compiler in future versions – so will additional optimizations beyond those.

Conversion to Intermediate Representations Instead of taking an abstract syntax tree (or an abstract syntax tree annotated with types) and directly producing text in the target language from that, it is often advisable to perform the conversion in multiple steps. In each step one representation of a program (or specification or document) will

be converted to another representation that is a bit closer to the final output format. This can mean transforming one type of tree into another type of tree whose node types are closer to the operations that exist in the target language (whereas the node types in the previous representations would have been closer to those in the source language) or it could mean transforming a tree into a flat representation of the program, like a sequence of instructions in some bytecode format.

Using multiple steps like that makes it easier to implement different target languages. When implementing a new target language it is not necessary to rewrite a completely new translation from an abstract syntax tree of the source language to the target language. Instead some of the steps used for the existing target language can be reused for the new one and only the steps that would need to be different for the new target language would have to be implemented. Depending on how similar the new target language is to the old one, it might only be necessary to implement very few new steps.

Low-Level Optimization Low-level optimizations are optimizations that only apply to one specific intermediate representation and cannot be applied at an earlier stage. Low-level optimization stages are usually interspersed with stages that convert to a lower-level representation. That is a program will be converted to a lower-level representation and then all low-level optimizations that apply to that representation will execute before it is converted to the next representation.

Code Generation In the code generation stage the lowest-level representation of a program will be converted to a program in the final target language.

2.2.2 Benefits and Drawbacks of Multiple Stages

As mentioned in the previous section many of the described stages are optional and often multiple different stages can be combined into a single stage. It is even possible to write a compiler that only consists of a parsing stage or a lexical analysis stage followed by the parsing stage. This design has been somewhat common in the past, but has become increasingly uncommon in modern times. In this section we will discuss the drawbacks and benefits of a compiler design with many stages compared to one with few stages or only a single stage.

One language-specific factor that needs to be considered is that some languages can only be compiled in multiple stages because certain decisions cannot be made without information that can only be known if later parts of the program have already been analyzed. For example many modern programming languages allow function calls to syntactically precede the definition of the called function without requiring any forward declarations. In those languages no part of the program can be type checked until all of the program has been parsed and the names and types of the functions defined in the

program have been collected. In those languages the choice is not whether or not to use multiple phases, but whether to minimize the number of stages or use as many stages as is convenient.

The major drawback of using multiple stages is that it can lead to longer compilation times as creating various intermediate representations and then processing them (multiple times in some cases) will generally involve more computational overhead than doing everything in one go. However in modern days computers have become fast enough that the overhead of multiple passes will not be a problem. Further modern compilers often perform intensive semantic analyses and optimizations that go far beyond what was possible in the past and whose costs far outweigh the cost of using multiple stages – making the latter cost insignificant in comparison. Note that this does not apply to tokenization and parsing – that is having a separate tokenization stage before the parsing stage will lead to improved performance as described in the previous section. Therefore even compilers that are designed to achieve minimal compilation times separate the tokenization and parsing stages.

The major benefit of using many stages is that it increases modularity. Using a multi-stage design each stage can perform a single function making it more readable and maintainable. It also becomes possible to modify one piece of functionality without affecting or having to touch any code that is not directly related to that functionality (and since all the code responsible for a given piece of functionality will be located in the same place, it will also be reasonably easy to do so). This also makes it easy to add new stages (like additional optimizations) or even multiple alternative for a given stage with only minimal changes to existing code. The most common example of this is that many compilers can produce different output formats (like machine code for different processors) depending on the platform or user choice. This is something that would require much more substantial changes in compilers with a less modular design and could quickly lead to unmaintainable code. Further it makes it possible to add a plug-in system through which users of the compiler can add additional stages like new optimizations or output formats, without having to touch any other code at all. This would be impossible to accomplish in a single-stage design.

In addition to some languages not being implementable using a single stage, some optimizations and optional semantic analyses also require information about the whole program from previous stages. Thus having a multi-stage design enables optimizations and analyses that are not otherwise possible. An example of an optional semantic analysis would be an analysis that collects semantic information about a piece of code that is not actually needed to compile the program, but can be useful to generate warnings (like “This line of code can never be reached”) or enable additional optimizations. For example many optimizations (like common subexpression elimination) in programming languages can only be performed on functions that don’t have any side-effects, so having a semantic analysis stage that checks which functions have side-effects, would enable performing such optimizations in cases where they are allowed.

The reasons listed above thus suggest that a multi-stage design is generally preferable to a single-stage design or a design that minimizes the number of stages.

3 Integrated Development Environments

Integrated development environments (IDEs) are computer programs that integrate the functionality of various development tools into one consistent environment. They can either do so by replicating that functionality themselves or by simply integrating existing tools into their user interface.

3.1 Features Commonly Found in IDEs

The functionality of an IDE generally includes the following:

Project Management The ability to create and manage projects and control which files are part of which project. This basic information can be used by other features of the IDE to make those features work better.

Building the Code Virtually all IDEs offer the ability to compile and/or execute one's project. By using the information that the IDE has about which files are contained in one's project and information that can be gained by performing code analysis on those files, the IDE can determine dependencies between the files in one's project automatically, making it unnecessary for the user of the IDE to set up make files (or similar build systems) manually.

Version Control Most IDEs offer the ability to integrate with version control systems. This allows the user of the IDE to view version control information (like which local files are in sync with the repository) in the IDE's project view, perform version control operations (like committing, updating, merging code) through the IDE's user interface and automatically inform the version control system of file operations performed through the IDE's project management features (like adding, removing and renaming files in the project).

Editing Code The most fundamental ability an IDE needs to support is editing code. In addition to basic editing capabilities this includes features commonly found in advanced code editors like:

- Syntax highlighting
- Automatic indentation
- Automatic completion of names and keywords

Code Navigation Most IDEs will offer navigation features such as listing all functions, variables and classes defined in a given file, the ability to jump to the definition of a given symbol from its use-site (taking into account properties like scope), even across file boundaries.

On the Fly Error Detection Most IDEs will detect code that contains compilation errors as it is typed and will mark it as such. Some also offer common fixes for some errors – for example Eclipse might offer to add an import statement to the code in Java if it detects that a class is being used whose name is not currently in scope, but that exists in the standard library.

Refactoring It is common for IDEs to offer certain refactoring tools like the ability to rename a class, function or variable, updating all references to it.

Debugging IDEs usually also integrate a debugger, allowing one to set break points, run the program in debug mode, step through the code and examine the contents of the stack from within the user interface of the IDE.

3.2 Well-Known IDEs

Some well-known IDEs are [Eclipse, 2013], [Netbeans, 2013] and [Visual Studio, 2013]. There are also certain advanced text editors, like [Emacs, 2013], that are sometimes considered IDEs as they offer most or all of the features common in IDEs either directly or through plug-ins.

One thing that sets Eclipse apart from other IDEs is the Xtext framework, which has been written for Eclipse. Xtext allows language implementers to create an Eclipse plug-in for their language that offers most of the functionality listed in the previous section without writing much (or, in some cases, any) code in addition to the compiler. The Xtext framework is described in more detail in chapter 4. The existence of this framework combined with the popularity of Eclipse as a Java IDE is what convinced us to use Eclipse as the basis of IDE support for SALT.

4 Xtext

Xtext is a compiler framework that allows language implementers to write a compiler for a language and an Eclipse plug-in for that language at the same time. Xtext generates code for an Eclipse plug-in that reuses code written for the compiler to implement IDE functionality. So, by writing their compiler using Xtext, language implementers can create an Eclipse plug-in for their language that offers most of the functionality listed in chapter 3 without writing much (or, in some cases, any) code beyond what is necessary to create the compiler anyway. This chapter will describe how Xtext works, what its features are, how it compares to other tools for compiler construction and, based on that, why Xtext was chosen to implement the SALT^{XT} compiler and Eclipse plug-in.

4.1 Structure of an Xtext Project

The heart of an Xtext project is its grammar file. The grammar file contains the following information:

- The types of tokens that the language consists of are specified by regular expressions for each type of token. Xtext will generate code to tokenize the language using this information. This works the same way as the common lexer generator tools described in section 2.2.1.
- The syntax of the language is described through a grammar. Information about what the produced abstract syntax tree should look like is provided by annotations that, for each production rule of the grammar, specify whether the abstract syntax tree should contain a node for that production and the name of the class of which the node should be an instance. Xtext will generate the code to parse the language and generate the abstract syntax tree from this information. It can also automatically generate the classes that make up the tree if instructed to do so.
- The grammar is annotated with information about when names are introduced and where they are used. Code for name resolution and auto-completion (in the Eclipse plug-in) are generated from this information.

In addition to the classes that will be generated from the grammar file an Xtext project will of course also contain non-generated classes. These classes are separated into two categories: classes that fulfill functions needed by the compiler and classes that only enhance or customize the Eclipse plug-in. The latter classes are all part of a separate

sub-project. None of the classes in the main project will contain code that is specific to IDE functionality.

4.2 Relation to ANTLR

Xtext uses the ANTLR parser generator to generate the parsing and lexing code. The syntax of Xtext's grammar file is the same as that of ANTLR except that, where ANTLR contains embedded Java code to be executed when a given production is used, Xtext contains annotation that describe which types of nodes should be generated as well as additional information (as described in the previous section).

Xtext works by generating an ANTLR grammar from the Xtext grammar (by replacing Xtext's annotations with embedded Java code) and then invoking ANTLR to generate a parser and a lexer from that grammar. Since Xtext does not allow Java code to be embedded into the grammar, there is no way in Xtext to make parsing decisions based on the results of executing Java code – something that can be done in ANTLR. Therefore Xtext grammars are exactly as powerful as ANTLR grammars that do not use Java code to make parsing decisions and strictly less powerful than ANTLR without that restriction.

4.3 Comparison to Other Tools

The most common tools that exist to facilitate the development of compilers are lexer generators and parser generators. A lexer generator is a tool that generates tokenization code from a list of regular expressions as described in section 2.2.1. A parser generator is a tool that generates parsing code from some form of annotated context-free grammar as described in the same section.

As described in section 4.2 Xtext, like ANTLR, offers the functionality of both of these types of tools. Unlike most other parser generators – including ANTLR – it does not allow arbitrary code to be executed during parsing; it is only possible to generate abstract syntax trees from the grammar. However any compiler that uses the multi-stage design described in section 2.2.1 will use the parser to generate an abstract syntax tree anyway, so this restriction of functionality does not affect such a compiler.

Another side-effect of the inability to embed executable code into an Xtext grammar is that it's not possible to make parsing decisions that are not context-free and it is thus not possible to ideally parse languages that are not context-free – i.e. it is possible for the generated parser to generate an "ambiguous" syntax tree, that is an abstract syntax tree where one type of node could represent one of multiple different syntactic constructs. A separate post-processing stage could then walk that tree and replace each ambiguous node with a node that can only represents one specific syntactic construct, but it is not

possible for an Xtext-generated parser to create an unambiguous abstract syntax tree directly. However this is not relevant for the SALT^{XT} compiler as SALT's syntax is entirely context-free. Therefore Xtext offers all the lexing and parsing functionality that is required for the SALT^{XT} compiler. Since Xtext not only automatically generates the code to build the abstract syntax tree, but also the class that make up the nodes of the tree, it is especially convenient to use.

In addition to parsing and lexing Xtext also offers features that help with parts of a compiler for which other tools do not offer any assistance. One of those features is that the grammar from which the parser is generated can also be annotated with information about references – that is a syntactic construct that introduces a new name can be annotated with that information and a syntactic construct that refers to a particular type of name can be annotated with that information as well. So for example the syntax for variable definitions could have an annotation to indicate that it introduces a new variable name and the syntax for using variables could be annotated to indicate that it refers to a variable name. That would look like this:

```
// A variable declaration consists of the keyword "declare" followed by
// an ID. The ID will be the name of that variable declaration.
VariableDeclaration : 'declare' name = ID ;

// A variable usage consists of an ID, but that ID should be the name of
// a variable declaration.
VariableUsage : [VariableDeclaration] ;
```

From these annotations Xtext will generate code to perform name resolution. This code can be extended by using Xtext's API to affect the scoping rules where the auto-generated code's assumptions about scope differ from the rules of your language and to enable importing and exporting of names across different files.

Xtext also provides a validation API that you can use to find and report errors in the source code. The main, not IDE related, benefit this has over writing validation code without such an API is that the mechanics of walking the tree are covered by the API – that is you don't have to write an implementation of the visitor pattern yourself.

However one of the most valuable features of Xtext is that it generates an Eclipse plug-in that makes use of much of the compiler functionality that is implemented using Xtext. For example, the generated plug-in uses the parser to highlight the syntax of the code. This functionality can be further customized through Xtext's API, but is already fully functional without writing any additional code. Similarly the plug-in performs auto-completion of keywords and names by using the grammar of the language as well as the name resolution functionality. No additional code – beyond what is needed for the compiler anyway – is needed to implement auto-completion. Likewise all errors and warnings that are produced by the compiler through the validation API, will also be detected on-the-fly by the plug-in and marked in the code and listed in Eclipse's problem view.

Additionally the plug-in provides an outline view that is generated using the grammar. However the outline that is provided by default will list a lot of unnecessary information (as it generates an entry for every node in the source code's abstract syntax tree) and is thus less useful. So unlike the other IDE features provided by Xtext, the outline is not very useful without writing additional code using the Xtext API to modify the view. For the SALT^{XT} plug-in this was not done as an outline view was not considered to be an important feature for SALT. Therefore the SALT^{XT} Eclipse plug-in does not provide an improved outline view so far.

There are also libraries like LLVM, which help in the code generation phase by allowing you to generate platform independent LLVM byte code, which can then be compiled into various machine code formats through the LLVM API. So you only have to write code generation code for one output format (LLVM byte code) and get support for many different machine code formats without having to write any additional code for any of them. Xtext does not offer any comparable functionality, but could be used in combination with such tools if needed. However, since SALT is a logical specification language that is compiled to logical formulas, not machine code, this is not relevant to the SALT^{XT} compiler.

5 Smart Assertion Language for Temporal Logic

Linear temporal logic is a powerful tool to specify the behavior of various types of components – be they computer programs that are verified using runtime verification tools or hardware components that are checked using model checkers. However linear temporal logic is a rather low-level way of writing specifications. It only offers a small set of core operators and offers no means of abstraction that can be used to structure large specifications or to avoid repetition. This can make it hard to write, read, debug and maintain large specifications and easy to make mistakes in them.

It is therefore desirable to have a higher-level language that has the same expressive power as linear temporal language and can be used with the same tools, but at the same time offers a greater set of operators, a more easily readable syntax and means of abstraction that make it possible to easily write large specifications that are still readable and maintainable.

SALT, which is short for "Smart Assertion Language for Temporal Logic", is such a language. It was proposed in [Bauer et al., 2006] and first implemented by Jonathan Streit in [Streit, 2006]. It offers a greatly expanded set of operators – all of which have English, rather than symbolic, names for greater readability – and the ability to define one's own operators to facilitate code reuse, maintainability and readability.

It also offers looping constructs to make assertions over a set of expressions, further facilitating code reuse and concise code. To enable SALT specifications to be used with existing model checking and runtime verfication tools, SALT can be compiled to the linear temporal logic dialects supported by those tools.

This chapter will incrementally describe the syntax and semantics of the SALT language. A complete, continuous definition of the SALT syntax will be given in appendix A. A complete list of operators and their semantics can be found in appendix B. A more comprehensive look at the SALT language can be found in [Streit, 2006].

The SALT language as defined in [Streit, 2006] also contains timed operators, which make it possible to write specifications that correspond to formulas in Timed LTL [Raskin, 1999]. It also includes a restricted form of regular expressions. The SALT^{XT} compiler described in this thesis does not currently support those constructs. Therefore this chapter will not describe the syntax and semantics of timed operators and confine itself to the subset of SALT without timed operators and regular expressions.

5.1 Operators and Atomic Propositions

In its simplest form a SALT specification consists of a list of assertions. The basic syntax of an assertion is as follows:

```
<assertion>     ::= 'assert' <expr>

<expr>          ::= <atomic_proposition>
                  | <operation>
                  | '(' <expr> ')'

<operator_expr> ::= <prefix_operator> <expr>
                  | <expr> <infix_operator> <expr> ( ',' <expr> )*
                  | <operand> <operator> '(' <operand> ( ',' <operand> )* ')'

<operand>       ::= <modifier>? <expression>
```

An atomic proposition is either one of the constants `true` or `false`, an alphanumeric identifier or a string in double quotes. Prefix operators are operators that have exactly one operand. Infix operators are operators that have two or more operands. Both infix and prefix operators can be used with the `operator(operands)` syntax. As usual parentheses can be used to affect the order of operations. Comments in SALT start with two dashes and extend to the end of the line.

Semantically the constants `true` and `false` are, rather unsurprisingly, propositions that always true of false respectively. Identifiers and strings represent variables or states that exist in the system which is being specified. Their semantics depend on that system. It also depends on the system which identifiers and strings have a meaning and which are meaningless or invalid – as far as the SALT language is concerned there are no restrictions on strings and identifiers. There is no semantic difference between a string and an identifier with the same contents (i.e. the identifier `x` will mean the same thing as the string `x`) – the only difference is syntactic: strings may contain characters that are not allowed in identifiers (e.g. spaces).

This is an example of a valid SALT specification:

```
assert x implies y
assert "hello world"
assert false implies (true and eventually z)
```

Some operators can also be used with scope modifiers. In that case one or more modifiers are inserted between one or multiple of the operands depending on the operator. The possible scope modifiers are `optional`, `weak`, `required`, `inclusive` and `exclusive`. Only one of `inclusive` and `exclusive` and one of `optional`, `weak` and `required` can be used per operand. Which modifiers are allowed or required before which operand depends on the operator.

```
<modifier> ::= <wro> <ie>
             | <ie> <wro>
```

```
              | <ie>
              | <wro>

<wro>       ::= 'weak'
              | 'required'
              | 'optional'

<ie>        ::= 'inclusive'
              | 'exclusive'
```

An example of a valid specification with scope modifiers is:

```
assert always x upto excl opt y
assert (next a) until weak b
assert b until required c
```

The semantics of an operator expression depend on the operator. The operators available in SALT are textual versions of the ones that are available in LTL as well as additional operators and generalized and extended versions of the operators known from LTL. Some of the available operators are:

Basic Logical Operators SALT has all the basic logical operators like `and`, `or`, `not` and `implies`. They have the obvious semantics.

Basic Temporal Operators SALT also has the basic temporal operators that exist in LTL, like `globally` (which can also be written as `always`), `eventually`, `releases` and `next`. Those operators have the same semantics as in LTL.

until The `until` operator in SALT is an extended version of the U and W operators in LTL. Its second operator can optionally be modified using the modifiers `weak`, `required` or `optional` and/or `exclusive` or `inclusive`. If none of the modifiers `weak`, `required` or `optional` are used, it acts as if the modifier `required` had been used. If neither `inclusive` nor `exclusive` are used, it acts as if `exclusive` had been specified.

When used with the modifiers `required` and `exclusive`, `until` is equivalent to the U operator in LTL and will hold iff the right operand holds eventually and the left operand holds during every step before then. When used with the modifiers `weak` and `exclusive`, it is equivalent to the W operator and holds iff the right operand holds eventually and the left operand holds during every step before then or the right operand never holds and the left operand always holds. When `inclusive` is used instead of `exclusive`, the left operand must still be true during the step in which the right operand first becomes true (whereas it usually would only need to be true during every step before then). When `optional` is used instead of `weak` or `required`, it will behave the same as `weak` except that it will always be true if the right operand never becomes true (even if the left operand is false during any or all of the steps).

upto The `upto` operator holds iff its left operand holds when only considering the steps up to the step where the right operand first becomes true. The right operand has to be modified using either `inclusive` or `exclusive` and either `weak`, `optional` or `required`. When `exclusive` is used, only the steps before the right operand first becomes true are considered. When `inclusive` is used, the step at which the right operand first becomes true is considered as well. When `required` is used, the expression does not hold if the right operand never holds. When `weak` is used and the right operand never holds, the expression holds iff the left operand holds. When `optional` is used and the right operand never holds, the expression holds regardless of whether the left operand holds. When `exclusive` is used on the right operand, either `weak` or `required` can (and, in some cases, must) be used on the left operand. When `inclusive` is used, the left operand must not be modified. If `exclusive` is used and the right operand holds in the current step, the rules determining whether the expression holds depend on the form of the left operand and the modifier used on the left operand. These rules are explained in [Streit, 2006] and will not be repeated here for conciseness. The right operand needs to be a purely Boolean (i.e. not temporal) proposition. The `upto` operator can also be written as `before`.

accepton The `accepton` operator holds iff the left operand holds when only considering the time before the step during which the right operand first holds or the right operand holds at a step before anything has happened that would mean that the left operand does not hold. For example `a until b accepton c` holds iff `a until b` holds when only considering the time before the step at which `c` first holds or `c` holds before any step at which `a` does not hold. If the right operand never holds, the expression holds iff the left operand holds. The right operand of `accepton` must be a purely Boolean proposition.

In addition to prefix and infix operators there are also counted operators:

```
<operator_expr> ::= <counted_operator> '[' <count> ']' <expr>

<count>         ::= <int>
                  | <rel_op> <int>
                  | <int> .. <int>

<rel_op>        ::= '<' | '=' | '>' | '<=' | '>='
```

A counted operator is called like a prefix operator except that there's a count in square brackets between the operator and the operand. The count can either be a single number, a range of two numbers separated by two dots or a number prefixed by a relational operator. Unlike prefix and infix operators, counted operators cannot be called using the `operator(operands)` syntax. An example of a valid specification with counted operators is:

```
assert nextn [3..5] x
assert occurring [42] y
```

```
assert holding [> 23] z
```

The semantics of the counted operators are as follows:

nextn `nextn [n..m] f` is true iff, for any i between n and m (inclusive), the formula `f` holds at step $c+i$, where c is the current step. `nextn [>= n] f` is true iff, for any $i \geq n$, the formula `f` holds at step $c+i$.

holding `holding [n..m] f` is true iff, for any i between n and m (inclusive), the formula `f` holds exactly during i steps. `nextn [>= n] f` is true iff f holds during at least n steps.

occurring `occurring [n..m] f` is true iff, for any i between n and m (inclusive), the formula `f` occurs exactly i times. `nextn [>= n] f` is true iff f occurs at least n times.

The difference between `holding` and `occurring` is that, if a formula holds for n consecutive steps, that counts as n steps during which it holds, but only as one single occurrence.

For all of these operators `[n]` and `[=n]` are equivalent to `n..n`, `[<= n]` is equivalent to `[0.. n]`, `[< n]` to `[0 .. (n-1)]` and `\il`[$>$ n]! to `[>= (n+1)]` for all integers `n`.

There is also the if-then-else operator:

```
<expr> ::= 'if' <expr> 'then' <expr> 'else' <expr>
```

The expression `if c then t else e` is equivalent to `(c implies t) and (not(c) implies e)`.

All temporal operators in SALT, except `accepton` (and its counterpart `rejecton`, which is explained in appendix B), have a corresponding past operator that has the same name with `inpast` appended to it. Some of them also have alternative names that are more intuitive. For example the past equivalent of `until` is `untilinpast`, which can also be written as `since`.

5.2 Loops

SALT also has looping constructs that can be used as expressions:

```
<expr>       ::= <quantifier> <list> 'as' <id> 'in' <expr>

<quantifier> ::= 'allof'
               | 'noneof'
               | 'someof'
               | 'exactlyoneof'

<list>       ::= 'list' '[' <expr> ( ',' <expr> )* ']'
               | 'enumerate' '[' <int> '..' <int> ']'
               | <list> 'with' <expr>
               | <list> 'without' <expr>
```

The semantics of the expression `quantifier list as var in f` are as follows: Let F be the set that contains the result of substituting the expression e for each free occurrence of the identifier `var` for each expression e in the list `list`. The semantics now depend on the used quantifier:

- The expression `allof list as var in f` will hold if all of the expressions in F hold.
- The expression `noneof list as var in f` will hold if none of the expressions in F hold.
- The expression `someof list as var in f` will hold if at least one of the expressions in F holds.
- The expression `exactlyoneof list as var in f` will hold if exactly one of the expressions in F holds.

5.3 Macros

In addition to assertions SALT specifications can also contain macro definitions. All macro definitions will be written at the beginning of a SALT specification, before the first assertion. The syntax for macro definitions is as follows:

```
<macrodef>    ::= 'define' <id> '(' <parameters> ')' ':=' <expr>
                | 'define' <id> ':=' <expr>
<parameters>  ::= <id> ( ',' <id> )*
```

A macro that has been defined with parameters can be used like an operator. If its parameter list contains only one parameter, it can be used like a prefix operator. If its parameter list contains two or more parameters, it can be used like an infix operator. In either case it can be used with the `operator(operands)` syntax.

When a macro is used this way, the expression is replaced with the macro's body (i.e. the expression right of the `:=` in the macro definition) and each free occurrence of any of the parameters in the body is replaced with the corresponding operand.

A macro that has been defined without a parameter list can be used like a variable and each use of it will be replaced by its body.

5.4 Composite Identifiers and Strings

When identifiers and strings are used inside a macro definition or loop, they may include, surrounded by dollar signs, any of the macro's parameters or the loop's variables. In that case they are composite identifiers or composite strings respectively and when the macro parameters or loop variables are substituted, the dollar-surrounded parts of identifiers and strings are replaced by the expression (presumably another identifier or string) being substituted for that identifier.

Here's an example of using composite identifiers in a loop:

```
assert
  allof enumerate [1..3] with i in
    motor_$i$_used implies motor_$i$_has_power
```

This will be equivalent to the following assertion without a loop:

```
assert
  (motor_1_used implies motor_1_has_power) and
  (motor_2_used implies motor_2_has_power) and
  (motor_3_used implies motor_3_has_power)
```

5.5 Variable Declarations

In addition to assertions and macro definitions, a SALT specification can also contain zero or more variable declarations, that must come before all macro definitions. So the complete production rule for a SALT specification is:

```
<spec> ::= <declaration>*
           <macrodef>*
           <assertion>+
```

The syntax for a variable declaration is:

```
<declaration> ::= 'declare' <id> (',' <id> )*
```

If the specification contains at least one declaration, then any time an identifier is used as an atomic proposition, the identifier (or, in case of a composite identifier, its expansion) must currently be bound according to the following rules:

- `declare` statements and macro definitions without parameters bind the given identifiers for the entire specification.

- Macro definitions with parameters bind the parameters inside the expression to the right of the `:=` and any of its sub-expressions.
- Loops bind the identifier following the `as` keyword inside the expression to the right of the `in` keyword and any of its sub-expressions.

If the identifier is used as an atomic proposition and the atomic proposition is used directly as an operand to a macro, the identifier may also be the name of an operator or previously defined macro. Note that this has been changed in SALT^{XT} as explained in section 6.6.2.

If the specification does not contain any declarations, any identifier can be used as an atomic proposition without restrictions. There are no restrictions on strings, even if the specification contains declarations.

6 The SALT^{XT} Compiler

6.1 Structure of the SALT^{XT} Compiler

The SALT compiler's structure consists of various phases, most of which are configurable and extensible through plug-ins. Those phases are lexing and parsing, validation and code generation.

The lexing and parsing phase in the SALT^{XT}compiler are largely performed by code that is auto-generated from a grammar that is written in Xtext's grammar language with some manually written code that runs between the lexer and the parser and preprocesses the token stream to implement composite variables. The parser generates an abstract syntax tree using classes that have also been auto-generated from the grammar. Those classes use the ECore mechanism of the Eclipse Framework. If syntax errors are found during this stage, the compilation aborts. When using Eclipse syntax errors are also marked in the code editor. Due to Xtext's support for cross-references references to non-existing macros or variables are also caught by the parser and handled in the same way. If the parser finishes parsing without any errors, the compilation process will continue with the validation phase.

In the validation phase, the validator goes over the AST to catch errors that haven't been caught by the parser. These errors are calling macros with the wrong arity and using modifiers (like `weak`) with operators that don't support them or leaving them out with operators that require them. In addition to these general validations, domain specific validations can also be performed using domain specification plug-ins, which will be explained in section 6.5. Like in the parsing phase, if errors are found, the compilation will be aborted and, when using Eclipse, the errors will be marked in Eclipse's editor. Otherwise compilation continues with the code generation.

The code generation phase takes the validated abstract syntax tree and converts it into one of the supported output formats. This process is divided into multiple translation phases. First if a domain specification plug-in is used, the plug-in can return a new AST on which some transformations have been done. Then the preprocessor runs on the AST and produces a new AST in which all macros, looping constructs and composite variables have been expanded. After this the rest of the code generation process is controlled by translation phase plug-ins. A translation phase can translate the code from one intermediate representation to another or into a final output format. It could also transform a program in an intermediate form into an optimized program in the

same intermediate form. Section 6.3 will describe how these plug-ins work and which translation phases currently ship with the compiler.

Some translation phases are only available when the specification meets certain requirements. For example generation of Spin output is only available when no past operators are used. To express this, predicate plug-ins, which are explained in section 6.4, are used.

6.2 Lexing and Parsing

In the SALT^{XT} compiler, lexing is performed in two stages: The first is performed by the automatically generated lexer that uses the token rules defined in the grammar file. The token stream produced by this lexer differs from the final token stream in how composite variables are treated:

There are four token types defined in the grammar to represent composite variables: `COMPOSITE_ID`, `COMP_ID_START`, `COMP_ID_MIDDLE` and `COMP_ID_END`. Of those the automatically generated lexer only generates the first type of token (the others are set to dummy values in the grammar that cannot possibly be matched). In the non-terminal rules, however, only the latter three types of tokens are used. So after the automatically generated lexer runs and before the parser all the `COMPOSITE_ID`s must be replaced by the token types that the parser expects. For this purpose there is a custom lexer class, that takes the token stream produced by the generated lexer and replaces each `COMPOSITE_ID` in it with a sequence of `COMP_ID_START`, `COMP_ID_MIDDLE`, `COMP_ID_END` and `IDENTIFIER` tokens.

The reason that the automatically generated lexer cannot produce the three different types of `COMP_ID_` tokens directly is that it's not possible for the generated lexer to tell that, for example, `Alice$likes$Bob` should be tokenized into `COMP_ID_START(Alice$)`, `IDENTIFIER(likes)` and `COMP_ID_END($Bob)` as opposed to `COMP_ID_START(Alice$)`, `COMP_ID_START(likes$)` and `IDENTIFIER(Bob)`. The reason for that is that the decision depends not just on which characters are currently being read, but also which token has been read previously. The generated lexer cannot keep track of such information, but our custom lexer can.

By splitting `COMPOSITE_ID`s into multiple tokens like this, we can explicitly state in the grammar the between the various `COMP_ID_*` tokens must be references to existing variables. This enables auto-completing of variables inside composite variables and produces errors when the given identifier is not the name of an existing variable currently in scope.

After the custom lexer processed the token stream, the parser parses it into an abstract syntax tree. The parser is again wholly generated from the grammar, which contains annotations to specify which parsing rule should create which type of node and what its

member variables should be set to. The node classes of the abstract syntax tree are also wholly generated from the grammar.

6.3 Translation Plug-ins

The SALT^{XT} compiler is designed to be extensible, so that it's possible to support as many output formats as possible. Some of the possible output formats may be LTL-based and some may be based on other logics. Some may support all SALT operations and some may only support a subset of them. In addition it should be possible to easily implement new translation strategies and optimizations and compare them to existing ones.

For this purpose the SALT^{XT} compiler has a plug-in system for translation phases, so that new translation phases can quickly be implemented, plugged into the existing infrastructure and run alongside the existing translation phases. Such phases might be new optimizations, a new implementation of an existing phase using a different translation scheme or a new output format.

This section describes how this plug-in system works.

6.3.1 The Translation Phase Interface

A new translation phase can be implemented by writing a translation plug-in. A translation plug-in is a class that implements the `TranslationPhase<From, To>` interface. Such a class must implement the methods `Specification<To> translate(Specification <From>)` and `List<Predicate> requirements()`.

The `translate` method takes some representation of a SALT specification and returns a transformed representation. The returned representation might either be of the same type – this might be the case in an optimization pass that performs some replacements on the AST, but does not change the types of nodes that can appear in the tree – or a new one, as is commonly the case when translating from SALT to an output format by way of different intermediate representations that are represented by different AST types. A translation phase that produces a final representation of the specification that is not meant to be processed further – a representation in what we call an output format – will use `String` as its target type (`To`). The class `Specification<T>` contains a list of the representations of each of the specification's assertions in the given type as well as some metadata like the specification's name – that is the name of the file that the specification is in without the file extension – as well as whether the specification uses a domain specification plug-in and if so, which one.

Some translation phases are only applicable if the specification meets certain conditions. For example Spin output can currently only be produced when the specification uses no

past operators and domain specific output formats may only be available if a specific domain specification plug-in is used (however no domain specific output formats are currently implemented). To express such requirements predicate plug-ins can be written (see section 6.4). The `requirements` method then returns a list of the predicate objects that represent the translation phase's requirement. For translation phases that have no requirements, the method will simply return an empty list.

Each class that implements the `TranslationPhase` interface must be registered with the `TranslationPhaseRegistry` class. That class contains two static lists: one for translation phases whose output are intermediate representations (we call such phases AST transformations) and one for phases whose output is in a final output format. Classes are registered by adding an instance of the class to the appropriate list.

6.3.2 Putting It All Together

In the code generation phase of the compiler, translation plug-ins are used the following way: After performing domain specific transformations (see section 6.5), preprocessing the SALT specification and then translating it to SALT core, the compiler calculates all combinations of translation phases that can be applied sequentially to translate the core specification into a specification output format. This is achieved by performing a depth first search in the graph where each `TranslationPhase<From, To>` is interpreted as an edge from the node `From` to the node `To`. The search starts with the node `CoreExpression`, the class that represents expressions in the SALT core language, and targets the node `String`.

Figure 6.1 shows a visualization of such a graph. Solid edges represent translation phases that exist in the version of the compiler described in this thesis (see section 6.3.3) while dashed edges represent translation phases that could be added as additional plug-ins. Similarly, nodes with solid borders represent intermediate representations or output formats that exist in this version of the compiler while nodes with dashed borders represent ones that could be added as additional plug-ins. The dotted line that divides the graphic horizontally separates the translation phases that always execute from those that are controlled by the plug-in system. Therefore the graph search described here starts directly below that line, i.e. at the node representing Core SALT. The nodes representing are labelled with the name of the output format rather than "String" for clarity's sake.

Before traversing an edge, the compiler checks that the given specification matches all of the translation phase's requirements by applying each of the predicates returned by the phase's `predicates` method. If that is not the case, the edge is ignored. After finding all possible paths this way, the possible paths are presented to the user, who can then select which path to follow. In effect this allows the user to choose which output format to generate and which translation scheme to use to get there.

The dialog for this can be be seen in figure 6.2. It would be advisable if future work on this project included creating a more usable user interface for this selection. After the user

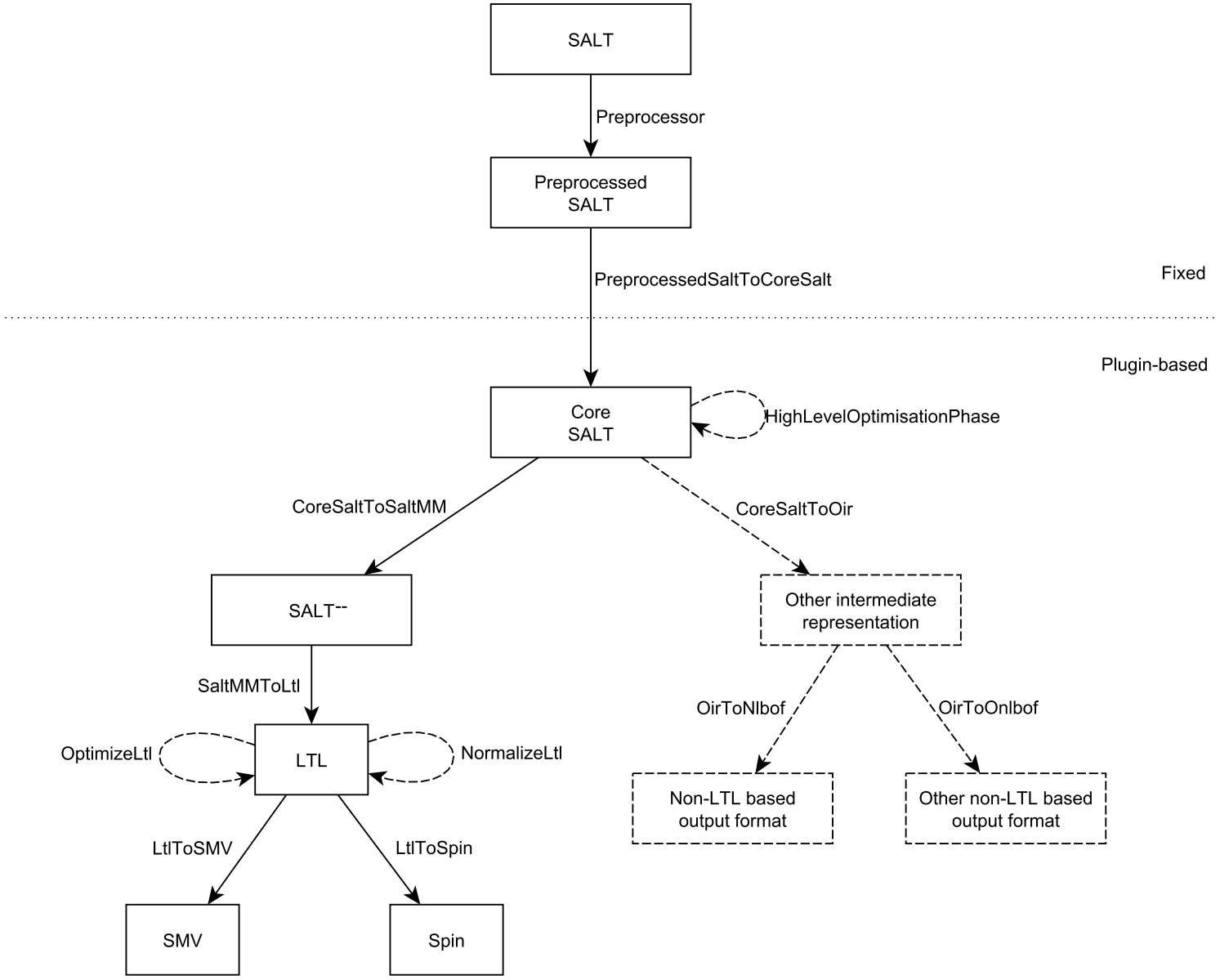


Figure 6.1: Graph of possible compilation paths

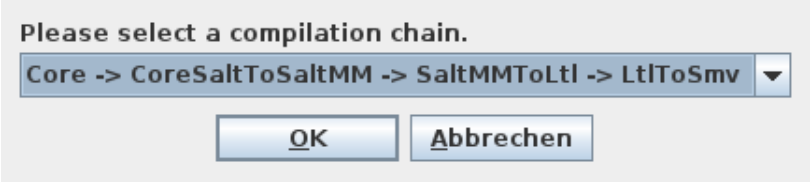


Figure 6.2: Compilation path selection dialog

selected a path, each translation phase from that path will be applied to the specification sequentially. The string returned by the last phase will then be written to the output file. The reason that the preprocessing and core translation phase always happen is that predicates work with `CoreExpression`, so the specification must be translated to core SALT for predicates to work. The rationale for this is explained in section 6.4.

6.3.3 Implemented Translation Phases

Currently the following translation phases are implemented in the SALT^{XT} compiler:

Preprocessor The preprocessor takes the AST generated by the compiler and replaces all macros, looping constructs and composite variables. It takes one AST object that represents the entire specification and returns a `Specification` object containing a list of each assertion's AST. The type of each expression is the Expression type generated by Xtext.

Core SALT The `PreprocessedSaltToCoreSalt` translation phase takes a specification object containing preprocessed SALT expressions and returns a `Specification <CoreExpression>`. `CoreExpression` is the type that represents expressions in the SALT core language. The SALT core language is SALT with the following syntactic restrictions:

- In regular expressions the only allowed quantifier is `*` without a count. All other quantifiers(`+`, `?`, `*{op n}`) are expressed in terms of `*`, `|` and `;`.
- The if-then-else construct is replaced by implication.
- The `between` operator is expressed in terms of `upto` and `from`.
- The `releases` operator is expressed as `until inclusive weak`.
- The `never` operator is expressed in terms of `not` and `eventually`.
- The operators `nextn`, `occurring` and `holding` are expanded to repeated applications of the appropriate operators. This translation is not implemented in the

current version of SALT^{XT}, so specifications containing these operators will currently not compile.

SALT^{--} The `CoreSaltToSaltMM` translation phase translates core salt into SALT^{--} (represented by the `SaltMMExpression` type). The SALT^{--} language contains the LTL operators, the SALT operators `rejecton` and `accepton` and the new operator `stopon`, which is used to express the various variations of the SALT operator `upto`. SALT^{--} does not contain modifiers, so modified versions of operators are expressed in terms of their plain version – with the exception of `until weak`, which is its own operator in SALT^{--}.

LTL The `SaltMMToLtl` translation phases translates SALT^{--} to LTL by translating the `rejecton`, `accepton` and `stopon` operators in plain LTL. It returns a `Specification <LtlExpression>`.

Output formats The phases `LtlToSMV` and `LtlToSpin` implement SALT^{XT}'s two currently supported output formats: SMV and Spin. They translate `LtlExpression`s to `String`s by expressing each LTL operation using its string representation in the given output format.

These phases implement the translation scheme described in appendix F of [Streit, 2006]. Except for `LtlToSpin`, none of the phases have requirements. The `LtlToSpin` phase has the requirement that the SALT specification may not contain past operators.

6.4 Predicate Plug-in

Certain translation phases can only be used under certain conditions. For example the Spin output format can currently only be used for specifications that don't use past operators[1]. Further certain translation phases may only be applicable if a specific domain specification plug-in is used.

To express these requirements, translation phases have a `requirements` method that returns `Predicate` objects. This section explains how these objects work.

[1] The SPIN format itself does not support past operators. Since every LTL formula with past operators can also be expressed without past operators [Gabbay et al., 1980], it would be possible to add a translation phase that replaces expressions involving past operators with equivalent expressions that don't use past operators, which would make it possible to use SPIN as an output format even if the specification contains past operators, but no such translation phase is currently implemented.

6.4.1 The Predicate Interface

Predicate plug-ins are created by writing a class that implements the `Predicate` interface. The `Predicate` interface has a single method `boolean isValid(Specification <CoreExpression>)`. The method takes a SALT specification and should return `true` or `false` depending on whether the given specification meets the predicate. The specification is the core SALT format because the transformations made when translating to core SALT are still light enough that, no matter which output format you're translating to, it always makes sense to translate to core SALT as an intermediate step, while at the same time core SALT removes enough redundancies in the language, that it becomes worthwhile to use it as you don't have to handle all the multiple ways to express the same thing that SALT sometimes allows.

6.4.2 The AbstractPredicate class

The `AbstractPredicate` class implements the `Predicate` interface. For every type of expression in the AST of a preprocessed SALT specification, the `AbstractPredicate` class has a method `boolean isValidExpression(ExpressionType)`. Further it has the method `isValidOperator(PrefixOperator)` and the method `isValidOperator(InfixOperator)`. These methods can be used to easily disallow certain types of operators without inspecting their operands. All of those methods have a default implementation (i.e. they're not abstract). In case of nodes that have children, the default implementation simply calls the appropriate `isValid` method for each of the node's children and then returns the conjunction of the results. For nodes that don't have children, it simply returns `true`.

It is expected that predicates will inherit from this class and then only handle the node types relevant to the predicate instead of implementing the `Predicate` directly.

6.4.3 Implemented Predicate Plug-ins

The following predicate plug-ins ship with the current version of SALT^{XT}:

The `NoPastOperators` predicate ensures that no past operators are used in the given specification. It does so by inheriting the `AbstractPredicate` class and overriding the `isValidOperator` methods to disallow any past operators as well as the `isValidExpression(RegularExpression)` method to disallow backward regular expressions. This predicate is a requirement of the `LtlToSpin` translation phase.

The `UsesDomainSpecification` predicate ensures that a specification uses a specific domain specification plug-in. It takes the class of the domain specification plug-in as a constructor argument. So to implement a translation phase that requires a domain

specification plug-in called `FooBar`, `requirements()` would return a list containing the following `Predicate` object:

```
new UsesDomainSpecification(FooBar.class)
```

6.5 Domain Specification Plug-ins

Specifications are written with the intent of verifying a system against the specification to ensure its correctness. For this to work it is, of course, imperative that the specification itself correctly represents the author's intent. A specification language like SALT should therefore be designed in a way that minimizes possible sources of mistakes and makes it easy to see when a specification contains mistakes. SALT accomplishes this by making it possible to define specifications in a structured way and using readable syntax. However there are still sources of mistakes in SALT. One such source is to misspell propositions in the specifications. For example a specification that requires that event A should always be followed by event B will be trivially – and thus uselessly – true if the name of event A is misspelled in the specification. To deal with this SALT allows you to specify valid propositions using the `declare` statement at the beginning of a specification. It will then mark any undeclared propositions as errors. However this approach has two major limitations:

- There is no guarantee that the propositions declared in the specification are actually meaningful in the context of the specified system. For example, when specifying the behavior of a sate machine, the propositions used in the specification might be the states of the state machine. However if you misremember the name of a state when declaring the states in the specification or if you rename the states of the state machine, but forget to apply the same renaming in the specification, the specification would become incorrect. SALT has no way of verifying whether the declared propositions correspond to states in the state machine because it doesn't know anything about the state machine. In other words SALT has no domain-specific knowledge and thus no way of knowing whether the declared propositions make sense in the context of the specified domain.
- Depending on the domain of the system to be specified and the properties to specify, there might be an infinite amount of reasonable propositions. That is a system might have a finite amount of variables that can be combined to propositions in an infinite number of ways (see the example below). Using the current approach the only way to write specifications would be to either declare each proposition you plan to use – not just each variable, but each proposition over those variables that you plan to use –, leading to a vast amount of declarations and ample opportunity to make mistakes in those declarations, or not declare any propositions at all and risk typos.

For example one might consider a system of mutable numeric variables where we want to specify the temporal behavior of the properties of those variables using arithmetic expressions and comparisons. So if we wanted to express that the variable x must not stay less than y+42 forever and that y must never become negative, this could be expressed using the following specification:

```
assert "x < y + 42" implies eventually "x >= y + 42"
assert never "y < 0"
```

In such a specification a valid proposition could be any comparison operator applied to any pair of arithmetic expression where an arithmetic expression could be any sequence of arithmetic operators applied to any combination of the system's variables and numeric constants. Instead of declaring each such comparison individually, it would be much better if we could just declare the variables that we're going to use and let SALT apply domain specific knowledge to figure out in what ways those variables can be combined to propositions. So in the example above, we'd want to only declare that the system contains the variables x and y instead of having to declare each of the propositions ($x < y + 42$, $x >= y + 42$, $y < 0$ and every other proposition about which we'd like to reason) individually. In fact what we'd really want would be to not declare anything at all and instead let SALT use its domain-specific knowledge about the kind of system with which we're working to find out what the system's variables are.

It is thus becoming apparent that we need a way to let users supply SALT with domain-specific knowledge about the systems for which they want to write specifications. For this purpose let us consider domain specification plug-ins – a way to write plug-ins for the SALT compiler that imbue it with domain specific knowledge. The SALT language is also extended to allow the integration of such plug-ins.

6.5.1 Extensions to SALT

To integrate domain specification plug-ins into SALT, the syntax and semantics of SALT are extended as follows:

- A specification can optionally begin with a **using** statement. A **using** statement consists of the keyword **using** followed by the name of a domain specification plug-in.

- The domain specification plug-in can perform checks on the specification's abstract syntax tree to determine whether it is valid for the given domain. It can also perform substitutions on the tree to implement domain-specific macros or other custom features.

6.5.2 Writing Domain Specification Plug-ins

A domain specification plug-in is a class that inherits the abstract `DomainSpecification` class. It must define the method `Specification transformAndValidate(Expression)`. The `transformAndValidate` method is called on the specification's abstract syntax tree. The class also has a protected field `ValidationMessageAcceptor acceptor`, which will be initialized before the `transformAndValidate` method is called. It may return a transformed AST (to replace domain-specific macros with pure SALT) or it may simply check the expression for validity. If the expression contains is not valid according to the domain-specific rules, the `ValidationMessageAcceptor` will be invoked and compilation will abort with the given error message. The acceptor can also use to produce warning messages, which will not abort compilation.

The class also defines the following non-abstract methods:

- The method `Specification transform(Specification)` performs the plug-in's transformations on the given spec without performing any validations. It is implemented by calling `transformAndValidate` after setting `acceptor` to an acceptor that simply ignores all messages. This method is used during the compilation process when an already-validated specification needs to be transform during the code generation phase.

- The method `static Specification get(Specification)` takes a specification, finds out whether that specification uses a domain specification plug-in and if so, returns the `DomainSpecificationPlug-in` object corresponding to that plug-in. Otherwise it returns `null`.

- The method `static Specification transformAndValidate(Specification, ValidationMessageAcceptor)` finds the specification's domain specification plug-in using `get`, sets its validator to the given validator and then invokes that plug-in's non-static `transformAndValidate` method on the specification. If the specification does not use a domain specification plug-in, the specification is returned unchanged.

The `DomainSpecificationPlug-in` will be used in two places:

- During the validation phase of the compilation the static `transformAndValidate` method is applied to the specification. If it finishes without signalling any errors, validation continues on the (possibly) transformed AST returned by the method.

- During the code generation phase, the `get` method is called with the specification as its argument. The result of this is stored in the `Specification<T>` objects that are passed around between the translation phases. If it is not null, the `transform` method will be called before the preprocessing phase and its result will be used as input for that phase.

6.5.3 The AbstractValidator class

Domain specification plug-ins that only perform validations and no transformations can inherit from the `AbstractValidator` instead of directly from the `DomainSpecification Plug-in` class. The `AbstractValidator` defines a `validateExp` method for each type of SALT expression. Each one takes the abstract syntax tree of an expression as its first argument. It also takes an arbitrary object as its second argument and returns an object. This object can be used to pass around state information. The default implementation of each `validateExp` simply calls the appropriate `validateExp` method on each subexpression with the state object as the second argument. It then returns the state object it was given unchanged.

`AbstractValidator` inherits from `DomainSpecificationPlug-in` and implements the `transformAndValidate` method by calling its own `validateExp` methods on each macro body and each assertion in the specification. As the second argument it uses the result of the abstract method `newState`, which users of this class need to override to return an "empty" state object.

Users of this class should override the `validateExp` method for those types of expression, for which they want to perform custom validations. They can use the state object returned by calls of `validateExp` on subexpressions to gain knowledge about those subexpressions.

6.6 Changes to the SALT Language

6.6.1 Macro Calls

The SALT language has the following grammar rules for non-nullary macro calls:

```
<macro_name>    ::= <ID>
<explicit_call> ::= <macro_name>
                    '(' <actual_param> (',' <actual_param>)* ')'
<prefix_call>   ::= <macro_name> <actual_parameter>
<infix_call>    ::= <actual_param> <macro_name> <actual_param>
                    (',' <actual_param>)*
<actual_param>  ::= <expression> | '@' <macro_name>
```

Nullary macros are defined using a syntax distinct from that of normal macros and are used simply by writing their names. Therefore they can be treated like variables in the parser. This makes them unproblematic and thus irrelevant to this discussion.

An expression can, among other things, be another macro call or the name of a variable. Variable names, like macro names, are simply identifiers. This makes the grammar for prefix_calls and infix_calls ambiguous. For example it is not clear from the grammar

whether abc is an infix macro call where b is a binary macro and a and c are variables, or whether there are two prefix macro calls where a and b are unary macros and c is a variable.

Further expressions like $afbgc, d$ are also ambiguous, even if we already decided that f and g are macros that we're calling using infix notation, because it is not clear whether d is an argument to the macro f or the macro g.

The previous version of the SALT language resolves these ambiguities by choosing from the possible parses that parse that will not treat a macro like a variable (or vice-versa) or call a macro with the wrong number of arguments. There will be at most one such parse and if no such parse exists, a parse error will be caused. For example $afbgc, d$ was parsed as $f(a, g(b, c, d))$ if f takes two arguments and g takes 3 or as $f(a, g(b, c), d)$ if f takes 3 arguments and g takes 2.

This approach makes it impossible to parse this syntax without keeping track of which macros exist and what their arities are. It can also be argued that it makes it harder for humans to read the code as they also need to keep track of those things to know which arguments belong to which macro.

The ANTLR parser used by the previous implementation of the language implements this behavior by embedding Java code in the grammar which creates a hash map that keeps track of which macros exist and what their arities are. It then makes parsing decisions based on the contents of that hash map.

Unlike ANTLR, Xtext does not allow parsing decisions to be based on embedded Java code like this – in fact Xtext does not allow embedded Java code in the grammar at all. Therefore it is not possible to implement these rules using an Xtext grammar. Because of this it was necessary to modify the SALT language in such a way that parsing decisions are based solely on the currently visible tokens, not the existence or arity of prior declarations.

One way to accomplish this would have been to simply disallow prefix and infix macro calls, so that all macro calls would have to use the explicit macro call notation. However this would have meant breaking backwards-compatibility in a major way. It also would have meant sacrificing some elegance of the syntax since user-defined macros could no longer be using the same syntax as built-in infix operators.

Instead an attempt has been made to come up with a set of rules that would be simple to understand and implement and that would lead to readable specifications while trying to minimize the number of previously correct SALT specifications that would cause errors using the new sets of rules. This resulted in the following rules:

- An expression of the form "<ID> <ID> <Expression>" will be interpreted as an infix macro call whose left operand is a variable. If the first id is not the name of a variable or the second id is not the name of binary macro, an error message is produced. To call two unary macros using prefix notation, the user needs to use

an expression of the form "macro1 (macro2 argument)".

- Similarly an expression of the form "<ID> <ID> <ID> <Expression>" will be parsed as an infix call whose right operand is an infix call.
- When nesting infix macro calls without parentheses all arguments will be interpreted to belong to the innermost macro call preceding them. So when given input of the form "a f b g c, d" the arguments of f will be a and the result of the second macro call and the arguments to g will be b and c. If f is not a binary macro or g is not a ternary macro, an error message will be produced. To call f with three arguments, the user needs to parenthesize the inner macro call, i.e. "a f (b g c), d".

6.6.2 Higher-Order Macros

In the previous version of SALT the syntax for defining non-nullary macros was as follows:

```
<macro_definition> ::= <ID> '(' <formal_param>
                       (',' <formal_param>)* ')'
                       ':=' <expression>
<formal_param>     ::= <ID>
```

In SALT it is possible to define higher-order macros, i.e. macros that take other macros as arguments. This means that an argument given to a macro can either be a SALT expression or another macro. To pass a macro as an argument, the name of the macro is prefixed with an @-character. In the previous version of the language there was no way to tell from a macro's signature which parameters were supposed to be expressions and which were supposed to be macros.

In the Xtext based implementation the definition of a formal parameter is now `<ID> | '@'<ID>` where a simple ID denotes a parameter that stands for an expression and an @-sign followed by an ID denotes a parameter that stands for a macro. This has the following benefits:

- When reading a SALT specification it is now possible to tell which parameters stand for expressions and which stand for macros just by looking at the macro's signature. This enhances readability.
- The formal parameter list of a macro now contains @-signs at the same positions at which the actual parameter list will also contain @-signs. So a macro that is called as "macro(@other_macro, var)" is now also defined as "macro(@m, x)". This makes the syntax look more consistent.
- Since the IDE knows which parameters are macros and which are not, it can offer more intelligent suggestions for auto-completion by only offering macro parameters as completions in positions where a macro name would be legal and non-macro parameters in positions where an expression would be legal.

6.6.3 Recursive Macros

Previously it was allowed to define macros that called themselves, i.e. macros that were recursive. Since the language offers no way to terminate such recursion, calling a recursive macro would always lead to non-termination. That is never desirable behavior and there is no use-case where a recursive macro would be useful without extending the language in such a way that would allow recursive macros to terminate, which would be outside the scope of this thesis (and not necessarily a good idea anyway). Therefore the new version of the SALT language no longer allows macros to be recursive.

7 SALT IDE

Previously there has been no support for SALT in IDEs or text editors. That meant that SALT specifications had to be written without any tool support. Since certain features of IDEs and advanced text editors can greatly enhance the programmers' productivity and help them find mistakes earlier. Basic IDE support is considered a necessary feature for SALT^{XT}. The Xtext framework not only helps with implementing a compiler, but also with creating an Eclipse plug-in using the same infrastructure as the compiler. Therefore SALT^{XT}'s IDE support has been implemented as an Eclipse plug-in. This section will explain the features of that Eclipse plug-in and how they were implemented.

7.1 Features

The SALT^{XT} Eclipse plug-in has the following features:

- SALT source code is highlighted using basic token-based rules, which highlight keywords and operator names. The highlighting rules are generated from the token rules of the grammar.
- Variable and macro names, keywords and operator names can be automatically completed. A menu containing the list of possible completions will pop up as the user types a long name. Alternatively the user can press the Control and Space keys simultaneously to make the menu pop up right away. The list of possible completions is context-aware, i.e. it only lists names or keywords that would be syntactically legal in the current context. It also distinguishes between names of macros and variables, so in a context where a macro name would be legal, but a variable name would not be (after an `@` for example), only macro names will be listed. The menu will never list local variables that are not in scope at the point of completion.
- Macros and variables can be renamed using Eclipse's refactoring feature. This can be done by invoking Eclipse's rename action either on the definition of the macro or variable or at a use site. It will rename all references to that variable or macro. This replacement is aware of scope and does not rename variables or macros that have the same name as the one that is being renamed but live in a different scope.
- From the call site of a macro or variable the user can jump to its definition – either from the context menu or by pressing F3. This feature is also scope-aware and will

always jump to the correct definition even if there are multiple definitions with the same name.

- All errors are detected as the user is typing. The user does not have to invoke the compiler to find out whether the code will cause compilation errors. The errors will be marked in the code and listed in Eclipse's "Problems" view. Error messages can be viewed either in the "Problems" view or by hovering over the error marking in the code. This includes general language errors (like syntax errors, using variables or macros that don't exist or are not in scope or using a macro with the wrong number of arguments) as well as domain-specific errors that are detected by domain specification plug-ins (see section 6.5).
- The SALT compiler can be invoked from within the IDE to compile SALT specifications to the desired output format.

It should also be mentioned that all the features that Eclipse always offers regardless of which language is used, like integration with version control systems, are of course also available when using the SALT^{XT} Eclipse plug-in.

7.2 Implementation

Much of the functionality of the Eclipse plug-in is generated directly from the grammar by the Xtext framework: The syntax highlighting rules are determined from the token definitions of the grammar and require no additional code. The automatic completion rules for keywords and built-in operator names is similarly generated from the grammar. The remaining features, including completion rules for variables and macros, require some additional information: They need to know where macros and variables are defined, where they are used and how far their scope intends.

The first two types of required information can be encoded directly into the Xtext grammar: In places where a new macro or variable name is introduced, i.e. in `declare` statements, `define` statements and macro parameter lists, we don't just use the token type `IDENTIFIER`, but rather use the non-terminals `MacroName` or `VariableName` respectively. Those rules simply invoke the terminal rule IDENTIFIER and store its value in the special property `name`. The `name` property tells Xtext that the rule represents a named entity to which references can be made elsewhere in the grammar. In the places where macro or variable names are used, we write `[MacroName]` or `[VariableName]` in braces, which is Xtext syntax to signify that the next token must be an Identifier that is not just a legal identifier, but also equal to the `name` property of the given non-terminal rule and the name's scope must extend to the current location.

To determine the scope of names additional code is required outside of the grammar definition: The `SaltScopeProvider` class is called for every reference in the grammar and walks upwards from that reference through the AST to find which names of the given

kind are currently in scope. Together the information provided by the grammar and this class, enable SALT^{XT}'s scope-aware IDE features. In addition to enabling IDE features, the same information is also used by the compiler to match variable and macro names to their definitions, so using the Xtext framework allowed us to create both a compiler and an Eclipse plug-in without duplicating functionality like scope resolution.

8 Conclusion

This thesis described the SALT^{XT} compiler and Eclipse plug-in. It started by explaining the fundamentals of compilers and IDEs. It then gave an overview of the SALT language. Afterwards it explained the language changes made in SALT^{XT} and then proceeded to describe the implementation of the SALT^{XT} Eclipse plug-in and the SALT^{XT} compiler.

The contributions of this thesis are as follows:

The SALT^{XT} Eclipse plug-in that has been introduced in this thesis is the first tool to offer any kind of IDE or editor support for SALT. Through features like syntax highlighting and automatic completion, SALT specifications can now be written more efficiently and comfortably than before.

The SALT^{XT} compiler introduced in this thesis only uses Java-based technologies and can be distributed as a single jar file. It is thus more easily deployable than the existing reference implementation.

The SALT^{XT} compiler also distinguishes itself with its easily extensible, plug-in based architecture. Through the three types of plug-ins described in this thesis, it is easily possible to extend the SALT^{XT} compiler with alternative translation strategies, new optimizations, transformations and output formats and to add knowledge of specific domains, allowing the compiler to check that the given specification is not only syntactically valid, but also makes sense for the given domain.

8.1 Future Work

8.1.1 Compiler

Feature-complete SALT Implementation

The version of the SALT^{XT} compiler presented in this thesis does not support regular expressions or timed operators. These could be implemented in a way similar to the operators that are supported with comparatively little effort. Support for regular expressions has since been added independently of this thesis.

Translation Phases That Add Predicates

One other feature that is currently not implemented, but that would be worthwhile to implement in the future is the ability to add predicates to specifications using translation phases:

Currently the check whether a specification meets a predicate is performed once after the specification has been translated to Core LTL. If a specification does not meet a predicate at that point, no translation phase that requires that predicate will be performed on the specification. The proposed feature would enable a translation phase to add a predicate to a specification, so that after running that translation phase on a specification, other translation phases that require the predicate can now be run on that translation.

An example where this would be useful would be a translation phase that replaced expressions involving past operators with equivalent expressions that don't use past operators, which is possible because, for every LTL formula containing past operators there is an equivalent formula that does not contain past operators [Gabbay et al., 1980]. Such a translation phase could add the `NoPastOperators` predicate to a specification (because after that translation phase is finished, the specification will indeed not contain any past operators anymore), allowing the specification to be compiled to SPIN, which does not support past operators.

This feature can be implemented by storing the information which predicates are met by a specification as part of the specification's `Specification<T>` object – as opposed to storing that information locally in the method that invokes the translation phases, as is the case now.

Any translation phase can then add predicates to the `Specification` object when it returns the new `Specification` object representing the translated specification. Such a change could be made relatively easily. This has not been implemented as part of this thesis, but has since been implemented indepently of this thesis.

More Translation Phases and Other Plug-ins

Jonathan Streit's SALT compiler can produce LTL formulas in symbolic notation as LaTeX code. Such an output plug-in (or one that directly produces PDFs by feeding the LaTeX code into a LaTeX compiler), could also be implemented for SALT^{XT}. Other output formats that support additional model checkers and runtime verifiers can also be written. For tools that understand some dialect of LTL, this would be very little work using the `LtlBasedOutputFormat` class. Tools that don't use LTL can be supported as well due to SALT^{XT}'s modular design. This might even include tools whose specification languages are less powerful than SALT. For those predicate plug-ins could be written to ensure that specifications only use those features that the target language is powerful enough to express.

In addition to new output formats, new intermediary translation phases could also be introduced. One such phase could be a phase that removes past operators as indicated earlier. Another example would be optimization phases that could implement both high-level optimizations, optimizing SALT or Core SALT expressions, or low-level optimizations, optimizing LTL or SALT^{--} expressions (or other intermediate representations that might be introduced by further plug-ins).

Currently the only domain specification plug-in that comes with SALT^{XT} is a small example plug-in that is of very little practical use, other than showing plug-in writers how a domain specification plug-in might be written. Real domain specification plug-ins could be written in the future.

User Friendliness of Translation Path Selection

Currently the dialog to select the translation path is very bare-bones and not user-friendly. This will be even more true, the more translation phase plug-ins are added to the compiler. The user also needs to remake his selection every time a specification is compiled, which is not ideal considering that a user will most likely want to target the same output format as before when re-compiling a specification. It would therefore be a good idea to create a more user-friendly dialog for selecting compilation chains that displays the available options in a more organized fashion remembers the user's previous selection. One way to simplify the interface would be to first only ask the user which output format he wants to target and then let him select from the compilation paths leading to that output format only.

When a given compilation path cannot be chosen due to its requirements, the current interface simply does not list it as an option. If a user intended to use a certain output format and is not aware that he's using features that are not supported by that output format, the fact that no compilation path leads to his intended output format will likely confuse the user. Therefore an improved version of the compilation path selection dialog should make clear which translation paths are not available and why (that is which requirements are not fulfilled). This could be achieved by graying out target formats or intermediate phases that are not available and show in a tooltip which requirements are not met. The interface might also include a "more information" button that, when clicked, would show the user exactly where his specification does not meet the requirements (e.g. where he's using an operator that is not supported by the given output format).

It should also be possible to select compilation paths via command line switches, so that the compiler can be used in a headless environment or by an automated build system.

A command line interface that allows compilation paths to be selected using command line switches has since been implemented independently of this thesis.

8.1.2 Eclipse Plug-in

The following additional features for the SALT^{XT} Eclipse plug-in could be implemented in a future release of SALT^{XT} with relatively little effort. All of these features would require some code to be written in the Eclipse-specific part of the project and would not affect the behavior of the compiler part of the project.

Various UI Fixes

- Syntax highlighting could be used to distinguish macro names from variables and local macros or variables from global macros or variables.
- Auto completion could be improved to not list unary macros when completing an infix macro call.
- Currently the names of domain specification plug-ins are not auto-completed when writing a `use` statement. This could be fixed.
- The outline view could be improved, so that it provides a more useful outline of the specification's structure.

Integration of Compilation Path Selection

When compiling from within Eclipse, it would be more consistent if the user could select his intended output format (and the compilation path to get there) through Eclipse's built-in "Run As" system rather than SALT^{XT}'s own selection dialog. This would make the process feel more integrated and it would not interrupt the user's workflow by making a dialog pop up.

SALT Projects

Currently the SALT^{XT} Eclipse plug-in does not add a project type for SALT projects – the user has to create a different type of project (like a Java project) and then add SALT files to it by adding a "misc" file and then giving it a ".salt" extension. This is counter intuitive. Thus a SALT project and file type should be added. When creating a new SALT project, a SALT file should be created along with it and creating a SALT file should invoke a wizard that asks whether a domain specification plug-in should be used and, if so, lets the user choose between the available domain specification plug-ins.

The project's settings could also offer the option to select a specific compilation path. Any use of features that violate that path's requirements could then directly be marked as errors by Eclipse's on-the-fly error detection.

A SALT Syntax

This appendix will give a continuous EBNF grammar describing the syntax of the SALT language as described in [Streit, 2006], excluding timed operators and regular expressions, which have not been implemented in this thesis and are thus of no relevance to it.

A.1 Grammar

```
<spec>              ::= <declaration>*
                        <macrodef>*
                        <assertion>+

<declaration>       ::= 'declare' <id> (',' <id> )*

<macrodef>          ::= 'define' <id> '(' <id> ( ',' <id> )* ')' ':=' <expr>
                      | 'define' <id> ':=' <expr>

<assertion>         ::= 'assert' <expr>

<expr>              ::= <atomic_proposition>
                      | <operator_expr>
                      | <quantifier> <list> 'as' <id> 'in' <expr>
                      | 'if' <expr> 'then' <expr> 'else' <expr>
                      | '(' <expr> ')'

<atomic_proposition> ::= 'true'
                       | 'false'
                       | <id>
                       | <string>

<operator_expr>     ::= <prefix_operator> <expr>
                      | <expr> <infix_operator> <expr> ( ',' <expr> )*
                      | <operand> <operator> '(' <operand> ( ',' <operand> )* ')'
                      | <counted_operator> '[' <count> ']' <expr>

<prefix_operator>   ::= '!'
                      | <id>

<infix_operator>    ::= '->'
                      | '<->'
                      | '&'
                      | '|'
                      | <id>
```

```
<operator>          ::= <prefix_operator>
                      | <infix_operator>

<counted_operator>::= 'nextn' | 'nextninpast' | 'previousinpast'
                      | 'occurring' | 'occurringinpast'
                      | 'holding' | 'holdinginpast'

<operand>           ::= <modifier>? <expression>

<modifier>          ::= <wro> <inclexcl>
                      | <inclexcl> <wro>
                      | <inclexcl>
                      | <wro>

<wro>               ::= 'weak'
                      | 'required'
                      | 'optional'

<inclexcl>          ::= 'inclusive'
                      | 'exclusive'

<count>             ::= <int>
                      | <rel_op> <int>
                      | <int> .. <int>

<rel_op>            ::= '<' | '=' | '>' | '<=' | '>='

<quantifier>        ::= 'allof'
                      | 'noneof'
                      | 'someof'
                      | 'exactlyoneof'

<list>              ::= 'list' '[' <expr> ( ',' <expr> )* ']'
                      | 'enumerate' '[' <int> '..' <int> ']'
                      | <list> 'with' <expr>
                      | <list> 'without' <expr>
```

A.2 Rules for Identifiers

An identifier can be any alphanumeric identifier that starts with a letter and can contain underscores – just like in most programming languages. When identifiers and strings are used inside a macro definition or loop, they may include, surrounded by dollar signs, any of the macro's parameters or the loop's variables. In that case they are composite identifiers or composite strings respectively. There are the following restrictions on using identifiers (in case of composite identifiers, the restrictions apply to its expanded version):

- If an identifier is used as a prefix operator, it must either be the name of one of the prefix operators listed in appendix B or the name of a macro that has been defined

with exactly one parameter.

- If an identifier is used as an infix operator it must either be the name of one of the infix operators listed in appendix B or the name of a macro that has been defined with two or more parameters.
- If an identifier is used as an atomic proposition and the specification contains one or more declarations statements, the identifier must currently be bound according to the following rules:
  - `declare` statements and macro definitions without parameters bind the given identifiers for the entire specification.
  - Macro definitions with parameters bind the parameters inside the expression to the right of the `:=` and any of its sub-expressions.
  - Loops bind the identifier following the `as` keyword inside the expression to the right of the `in` keyword and any of its sub-expressions.

  If the identifier is used as an atomic proposition and the atomic proposition is used directly as an operand to a macro, the id may also be the name of an operator or previously defined macro. Note that this has been changed in SALT^{XT} as explained in section 6.6.2.

  If the specification does not contain any declarations, any identifier can be used as an atomic proposition without restrictions.

B List of SALT Operators

This appendix lists all operators that exist in the SALT language and gives a short explanation for each of them. For a more in-depth explanation see [Streit, 2006].

If an operator's description does not mention modifiers, no modifiers are allowed on any of its operands.

In the following descriptions all times will be relative to the current step, e.g. a phrase like "the first step at which x holds" does not consider any steps prior to the current step. So if x holds at steps 2, 4 and 5 and the current step is 3, the first step at which x holds would refer to step 4, not 2. Likewise "the last step at which x held" would be 2, not 5.

B.1 Prefix Operators

This is a list of all operators that are used with exactly one expression as their operand. These operators can be invoked either using prefix syntax (`operator operand`) or function call syntax (`operator( operand )`).

not The `not` operator negates its operand. It can also be written as `!`.

next The `next` operator holds iff its operand holds in the next step. The right operand can optionally be modified using the `weak` modifier. If there is no next step (which can be the case if `next` is used inside an `upto` expression), the expression will hold iff the `weak` modifier has been used.

eventually The `eventually` operators holds iff its operand holds in the current or any subsequent step.

always The `globally` operator holds iff its operand holds in the current and all subsequent steps. It can also be written as `globally`.

never The `never` operator is the opposite of the `always` operator. It holds iff its operand holds neither in the current step nor in any of the subsequent steps.

nextinpast The `nextinpast` operators holds iff its operand holds in the previous step. The right operand can optionally be modified using the `weak` modifier. If there is no previous step (which can be the case if `next` is used inside an `uptoinpast` expression), the expression will hold iff the `weak` modifier has been used. The `nextinpast` operator can also be written as `previous`.

eventuallyinpast The `eventuallyinpast` operators holds iff its operand holds in the current or any previous step. It can also be written as `once`.

alwaysinpast The `alwaysinpast` operator holds iff its operand holds in the current and all previous steps. It can also be written as `historically`.

neverinpast The `neverinpast` operator is the opposite of the `alwaysinpast` operator. It holds iff its operand holds neither in the current step nor in any of the previous steps.

B.2 Infix Operators

This is a list of all operators that are used with two or more expressions as their operands. Most of these operators are used with exactly two operands, so the number of operands will only be mentioned when it is not two. These operators can be invoked either using infix syntax (`operand1 operator other_operands`) or function call syntax (`operator(operands)`).

and The `and` operator holds iff both its operands hold. It can also be written as `&`.

or The `and` operator holds iff at least one of its operands holds. It can also be written as `|`.

implies The `implies` operator holds iff both its operands hold or its left operand does not hold. It can also be written as `->`.

equals The `equals` operators holds iff both its operands hold or both its operands do not hold. It can also be written as `<->`.

until The `until` operator in SALT is an extended version of the U and W operators in LTL. Its second operator can optionally be modified using the modifiers `weak`, `required` or `optional` and/or `exclusive` or `inclusive`. If none of the modifiers `weak`, `required` or `optional` are used, it acts as if the modifier `required` had been used. If neither `inclusive` nor `exclusive` are used, it acts as if `exclusive` had been specified.

When used with the modifiers `required` and `exclusive`, `until` is equivalent to the U operator in LTL and will hold iff the right operand holds eventually and the left operand holds during every step before then (not considering any steps before the current step). When used with the modifiers `weak` and `exclusive`, it is equivalent to the W operator and holds iff the right operand holds eventually and the left operand holds during every step before then or the right operand never holds and the left operand always holds. When `inclusive` is used instead of `exclusive`, the left operand must still be true during the step in which the right operand first becomes true (whereas it usually would only need to be true during every step before then). When `optional` is used instead of `weak` or `required`, it will behave the same as `weak` except that it will always be true if the right operand never becomes true (even if the left operand is false during any or all of the steps).

releases The `releases` operator holds iff its right operand remains true until the left operand first becomes true (if the left operand never becomes true, the right operand must always remain true).

upto The `upto` operator holds iff its left operand holds when only considering the steps up to the step where the right operand first becomes true. The right operand has to be modified using either `inclusive` or `exclusive` and either `weak`, `optional` or `required`. When `exclusive` is used, only the steps before the right operand first becomes true are considered. When `inclusive` is used, the step at which the right operand first becomes true is considered as well. When `required` is used, the expression does not hold if the right operand never holds. When `weak` is used and the right operand never holds, the expression holds iff the left operand holds. When `optional` is used and the right operand never holds, the expression holds regardless of whether the left operand holds. When `exclusive` is used on the right operand, either `weak` or `required` can (and, in some cases, must) be used on the left operand. When `inclusive` is used, the left operand must not be modified. If `exclusive` is used and the right operand holds in the current step, the rules determining whether the expression holds depend on the form of the left operand and the modifier used on the left operand. These rules are explained in [Streit, 2006] and will not be repeated here for conciseness. The right operand needs to be a purely Boolean (i.e. not temporal) proposition. The `upto` operator can also be written as `before`.

from The right operand of the `from` operator must be modified with the same modifiers as the right operand of `upto`. The left operand must not be modified. The `from` operator

holds iff its left operand holds at (if the `inclusive` modifier is used) or after (if the `exclusive` modifier is used) the step where the right operand first becomes true. The meanings of the other modifiers are like those of `upto`. Like `upto` the `from` operator requires the right operand to be a purely Boolean proposition. The `from` operator can also be written as `after`.

between The `between` operator is applied to three operands. It is a combination of the `upto` and `from` operators. It holds if its first operand holds when only considering the steps from the step where the second operand first becomes true and up to the step where the third operand first becomes true. The allowed and required modifiers for the first operand as well as their meanings are the same as for `upto`. The allowed and required modifiers for the second and third operands as well as their meanings are the same as those the right operand of `upto`. The second and the third operand must be purely Boolean propositions.

accepton The `accepton` operator holds iff the left operand holds when only considering the time before the step during which the right operand first holds or the right operand holds at a step before anything has happened that would mean that the left operand does not hold. For example `a until b accepton c` holds iff `a until b` holds when only considering the time before the step at which `c` first holds or `c` holds before any step at which `a` does not hold. If the right operand never holds, the expression holds iff the left operand holds. The right operand of `accepton` must be a purely Boolean proposition.

rejecton The `rejecton` operator holds unless and only unless the left operand does not hold or the right operand holds before the step at which it can be determined that the left operand definitely holds. The right operand of `rejecton` must be a purely Boolean proposition.

untilinpast The `untilinpast` operator holds iff the left operand holds in the current and every previous step until the step at which the right operand last held. It has the same modifiers with the same meanings as `until`. It can also be written as `since`.

releasesinpast The `releasesinpast` operator holds iff the right operand has held ever since the left operand has last been true. It can also be written as `triggered`.

uptoinpast The `uptoinpast` operator holds iff the left operand holds when only considering the time after the right operand has last been true. It can be modified in the same way as the `upto` operator. The right operand needs to be a purely Boolean proposition.

frominpast The `frominpast` operator holds iff the left operand held at the step where the right operand has last been true (if the `inclusive` modifier is used) or at the step before it (if the `exclusive` modifier is used). It can be modified in the same way as the `from` operator. The right operand needs to be a purely Boolean proposition.

B.3 Counted Operators

This is a list of all operators that are used with a count and an expression as their operands. These operators can be invoked using the counted operator syntax: `operator [count] operand`.

nextn `nextn [n..m] f` is true iff, for any i between n and m (inclusive), the formula `f` holds at step $c+i$, where c is the current step. `nextn [>= n] f` is true iff, for any $i \geq n$, the formula `f` holds at step $c+i$.

holding `holding [n..m] f` is true iff, for any i between n and m (inclusive), the formula `f` holds exactly during i steps. `nextn [>= n] f` is true iff f holds during at least n steps.

occurring `occurring [n..m] f` is true iff, for any i between n and m (inclusive), the formula `f` occurs exactly i times. `nextn [>= n] f` is true iff f occurs at least n times.

The difference between `holding` and `occurring` is that, if a formula holds for n consecutive steps, that counts as n steps during which it holds, but only as one single occurrence.

nextninpast `nextninpast [n..m] f` is true if, for any i between n and m (inclusive), the formula `f` holds at step $c-i$, where c is the current step. `nextn [>= n] f` is true if, for any $i \geq n$, the formula `f` holds at step $c-i$. It can also be written as `previousn`.

holdinginpast `holdinginpast [n..m] f` is true if, for any i between n and m (inclusive), the formula `f` holds exactly during i steps. `nextn [>= n] f` is true if f holds during at least n steps.

occurringinpast `occurringinpast [n..m] f` is true if, for any i between n and m (inclusive), the formula `f` occurs exactly i times. `nextn [>= n] f` is true if f occurs at least n times.

For all of counted operators `[n]` and `[=n]` are equivalent to `n..n`, `[<= n]` is equivalent to `[0.. n]`, `[< n]` to `[0 .. (n-1)]` and `[> n]` to `[>= (n+1)]` for all integers `n`.

References

[Accellera, 2004] Accellera (2004). Property Specification Language Reference Manual Ver- sion 1.1.

[Aho et al., 1986] Aho, A. V., Sethi, R. and Ullman, J. D. (1986). Compilers: Principles, Techniques, & Tools. Addison-Wesley, Reading, MA.

[Armoni et al., 2002] Armoni, R., Fix, L., Flaisher, A., Gerth, R., Ginsburg, B., Kanza, T., Landver, A., Mador-Haim, S., Singerman, E., Tiemeyer, A., Vardi, M. Y. and Zbar, Y. (2002). The ForSpec temporal logic: A new temporal property-specification language. In Tools and Algorithms for Construction and Analysis of Systems pp. 196–211,.

[Bauer et al., 2006] Bauer, A., Leucker, M. and Streit, J. (2006). SALT—Structured Assertion Language for Temporal Logic. Technical Report TUM-I0604 Institut für Informatik, Technische Universität München.

[clang, 2013] clang (2013). http://clang.llvm.org. [Online; accessed August 20, 2014].

[Eclipse, 2013] Eclipse (2013). http://www.eclipse.org/. [Online; accessed August 20, 2014].

[Emacs, 2013] Emacs (2013). http://www.gnu.org/software/emacs/. [Online; accessed August 20, 2014].

[Gabbay et al., 1980] Gabbay, D. M., Pnueli, A., Shelah, S. and Stavi, J. (1980). On the Temporal Basis of Fairness. In POPL, (Abrahams, P. W., Lipton, R. J. and Bourne, S. R., eds), pp. 163–173, ACM Press.

[gcc, 2013] gcc (2013). http://gcc.gnu.org/. [Online; accessed August 20, 2014].

[Haskell, 2013] Haskell (2013). http://www.haskell.org/. [Online; accessed August 20, 2014].

[Netbeans, 2013] Netbeans (2013). http://www.netbeans.org/. [Online; accessed August 20, 2014].

[Pnueli, 1977] Pnueli, A. (1977). The Temporal Logic of Programs. In FOCS pp. 46–57, IEEE Computer Society.

[Python, 2013] Python (2013). http://www.python.org/. [Online; accessed August 20, 2014].

[Raskin, 1999] Raskin, J.-F. (1999). Logics, Automata and Classical Theories for Deciding Real-Time. PhD thesis, Universite de Namur Namur, Belgium.

[Reps, 1998] Reps, T. (1998). "Maximal-Munch" Tokenization in Linear Time. ACM Transactions on Programming Languages and Systems *20*, 2008.

[Streit, 2006] Streit, J. (2006). Development of a programming-language-like temporal logic specification language. Diploma thesis Munich University of Technology.

[Visual Studio, 2013] Visual Studio (2013). http://www.microsoft.com/visualstudio/. [Online; accessed August 20, 2014].