STALIN: A plugin-based modular framework for program analysis

Daniel Dietsch
Departement of Computer Science

Chair of Software Engineering
Albert-Ludwigs-University Freiburg

First Advisor: Prof. Dr. Andreas Podeslski
Second Advisor: Prof. Dr. Christoph Scholl

Thesis submitted for the Degree of Bachelor of Science in the
Albert-Ludwigs-University Freiburg

· 2008 ·

Abstract

Modern program analysis and model checking algorithms are getting more and more demanding with regard to scalability. Current trends in this domains suggest a solution through combining existing approaches to maximize performance. This leads inevitable to interoperability issues caused by incompatibilities in languages, logics and abstractions. We need standardization to overcome those problems and concentrate again on the development of algorithms. Our work presents the STALIN framework to tackle this challenges by providing an open and extensible architecture as well as by proposing direct solutions to them.

Keywords: Program Analysis, Model Checking, Verification, Programming Languages, RCP, Frameworks, Program Modeling, Code Visualization, Abstraction
CR Categories: D.2.4 Model Checking, D.2.11 Languages, D.2.3 Pretty Printers, D.2.5 Code inspection and walk-throughs, D.3.2 Specialized application languages, D.3.3 Frameworks, F.3.2 Program analysis, E.1 Graphs and networks, I.6.5 Modeling methodologies

Acknowledgments

First of all I would like to thank the original STALIN team from the safer systems group 2007, Arend, Christian, Justus, Nicola, Robert and Peter. Without you, this work would not have been possible. You were a great team and I am sure your skills have found use in other compelling projects. My special thanks go out to Justus for his continuing great work with STALIN, especially the parser and language part, and to Christian, who knows everything about RCP and was always there when I had a question.
Furthermore I want to thank all members of the Chair of Software Engineering for the great atmosphere and the continuous support, especially Dr. Jochen Hoenicke, who could always answer all of my theoretical questions (I am still not sure how he does it; he knows everything), Stefan Maus, who gave me directions for my work and my studies and who was not afraid to call a spade a spade, Stephan Arlt, the undisputed master of Software Engineering, for the great time in Berlin and his professionalism, Martin Mehlmann for his advice on object oriented programming, class design and graph traversal and Evren Ermis for great times and great Baklava.
A special place in this list is reserved for Martin Schäf, who had the original idea and the name for the STALIN framework as well as endless knowledge about program analysis and software engineering. His friendship and support helped me not only with STALIN but with a great deal of my studies. He showed me that there is more to research than knowledge, expertise and endless work in dark rooms.
Of course I also want to thank my first advisor and Professor Andreas Podelski. His steady belief in my work and in STALIN as well as his ability to protect me from poor talks are unmatched. Also, I want to thank my second advisor Professor Christoph Scholl for he was willing to evaluate this work with such

short notice. Many thanks to both of them for their patience and flexibility in regard to this work.
Last but not least I want to thank my family for their unweary support despite their sacrifices and their uninterrupted understanding against my often harsh temper. You have given me more than any child could ever expect.

Contents

List of Figures

Chapter 1

Introduction

In the past decade static program analysis in general and especially software model checking made great achievements towards safer and more precise software. Many interesting approaches have been presented and many have been implemented in numerous tools (like SPIN [1], SLAM [2], BLAST [3], Bogor [4], NuSMV [5] and others) with all their strengths and weaknesses. Although those tools can prove or refute several important properties they all suffer from common issues like scalability, narrowness with respect to programming languages or to usable properties. Right now none of the known programs can prove that an arbitrary property holds for an industrial sized software and least of all in a fully automated way. Of course we cannot expect to solve those issues due to the complexity of our problem, but we can aid the research in this area. Many promising, new ideas are combinations of different known techniques, like execution with dynamic analysis [6] or the combination of termination proofs with given preconditions and precondition inference for correctness properties [7]. Those combinations suffer from compatibility issues in the representation of program ab-

stractions and the complexity of unifying different specification languages. Also, there is a huge amount of repetitive work associated with the implementing of only one combination of different approaches compared to the formal description of the same. This amount does not yield necessarily usable results and is thus more and more disregarded. To sketch some of the aspects of repetitive work consider this brief sketchy example:

We have a tool which is able to infer loop invariants very well and another tool which can prove heap assumptions if we supply loop invariants. Lets call the first one invariant finder and the second heap analyzer. Naturally, we want to combine those two tools. The easy process would, for instance, look like this: We write all invariants found by our invariant finder in a file and read it back with the heap analyzer. We only need a way of transferring between the formalism of both tools, which can be a lengthy process. It requires understanding all the rough edges of the formalism, writing an own parser, annotating the new information in our formalism at the desired locations and an error-prone integration in mostly non-modular code. But what if the invariant finder takes a special loop-description language as input and the heap analyzer wants to have an input program written in ANSI C? We have to write even more code, namely a converter which takes our C loops and converts them into the input language of the invariant finder. After all, we need to spend a lot of time and resources with a problem which does not cover the real issue.

There is even more to it. Model checking tools use commonly system calls to one of the different SMT [8] or SAT [9] solvers which in turn use different decision procedures [10, 11, 12] to decide if a given logical formula is satisfiable or not. Those solvers work with different logical theories and based on the problems at hand, different theories are useful. The question for the combination of differ-

ent tools is therefore also a question for the transitions between different logic theories.

To overcome those problems and be able to focus on the interesting, algorithmic part of the work we need standardization and especially modularization in several areas:

- We need a universal language to express different programming languages in a coherent way.
- We need parser which can translate at least the important programming languages to this language.
- We need fully specified and formalized models to abstract from this universal language. We also need tools which generate those models.
- We need mechanisms to store and manage models efficiently.
- We need easy and efficient access methods for the models.
- We need standardized interfaces for the interaction between different tools.
- We need transitions between different theorem prover and SMT solver theories.

In short, we need a framework which allows us to concentrate on the real work and combines the efforts of different experts from different domains with little to no extra work.
There exist several interesting approaches from different people to tackle one or more items from this wish list. One of them is the **C** **I**ntermediate **L**anguage

(CIL) [13]. CIL is not only a language but also comes with a set of tools for parsing, merging and transforming ANSI C, MS C and GNU C programs. It can normalize a given input program by transforming the more complex expressions from C to simpler ones and it also provides an interface to attach other, external tools. It is well-suited to deal with C programs but does not support other languages.

Another approach is proposed in [14, 15] by Michael Huth. It specifies an interaction method between different tools and how different abstractions can be conveyed into each other. It also proposes a generic specification language enclosed in a framework for abstraction and partly refinement. The described framework was to our knowledge never implemented which would have naturally provided an opportunity to evaluate its feasibility in the context of existing algorithms.

Both works were very inspiring to us and lead to the development of the STALIN framework which fills the gap. The STALIN framework simplifies experiments with and the implementation of concatenations of different tools, so-called toolchains. It is written in Java and based on the Eclipse **R**ich **C**lient **P**latform [16]. RCP is the basic platform for the Eclipse Integrated Development Environment [17] and many other development tools. We also build our application on top of this successful and approved platform. It provides platform independence, support for numerous UI features, plugin handling, automatic online update mechanisms and integration into the Eclipse IDE to name the most important features for our decision. But because the RCP framework is a large and compelling topic for itself, we will not discuss it in this work. If you are interested in getting to know RCP and its benefits we suggest you read [18] or take a look at the RCP community site at [19].

Our framework provides interfaces for different parsers and tools, manages the created models in a standardized way and abstracts a given input language to a given universal language. One can execute different tools consecutively and work on the same standardized model, be it a parse tree, a control flow graph, a call graph or any other graph based model. We also provide a set of tools to perform different commonly used transformations on our models and to aid development and debugging through graphical representations of the models. Besides tools we also deliver parser for different languages like Java, C#, ANSI C and C++.
The main contributions of this work are:

- A concrete architecture for a plugin-based framework which can support all aspects of our wish list.
- An underlying infrastructure enabling further extension of the framework.
- Different tools aiding users of our framework in their tool development.

Chapter 2

STALIN Framework

2.1 General Architecture

The architecture of our framework is loosely based on the Model-View-Controller Pattern [20], where the different tools are seen as view but can also change the content of a model through an access layer. In figure 2.1 you can see the general outline of our architecture:

The core works as extended controller for the user interface and connects the different views (tools and parser) with the model manager. An execution run starts with loading the Eclipse RCP base application. Every part of STALIN is integrated as a separate plugin into Eclipse RCP. The RCP class loader loads the STALIN core which, based on his arguments, loads an user interface controller and delegates the control to it. If the GUI part is loaded, the user now can select one or more files as input and the different tools he wants to run on this input. After this selection, the user-interface-controller returns control to the core which

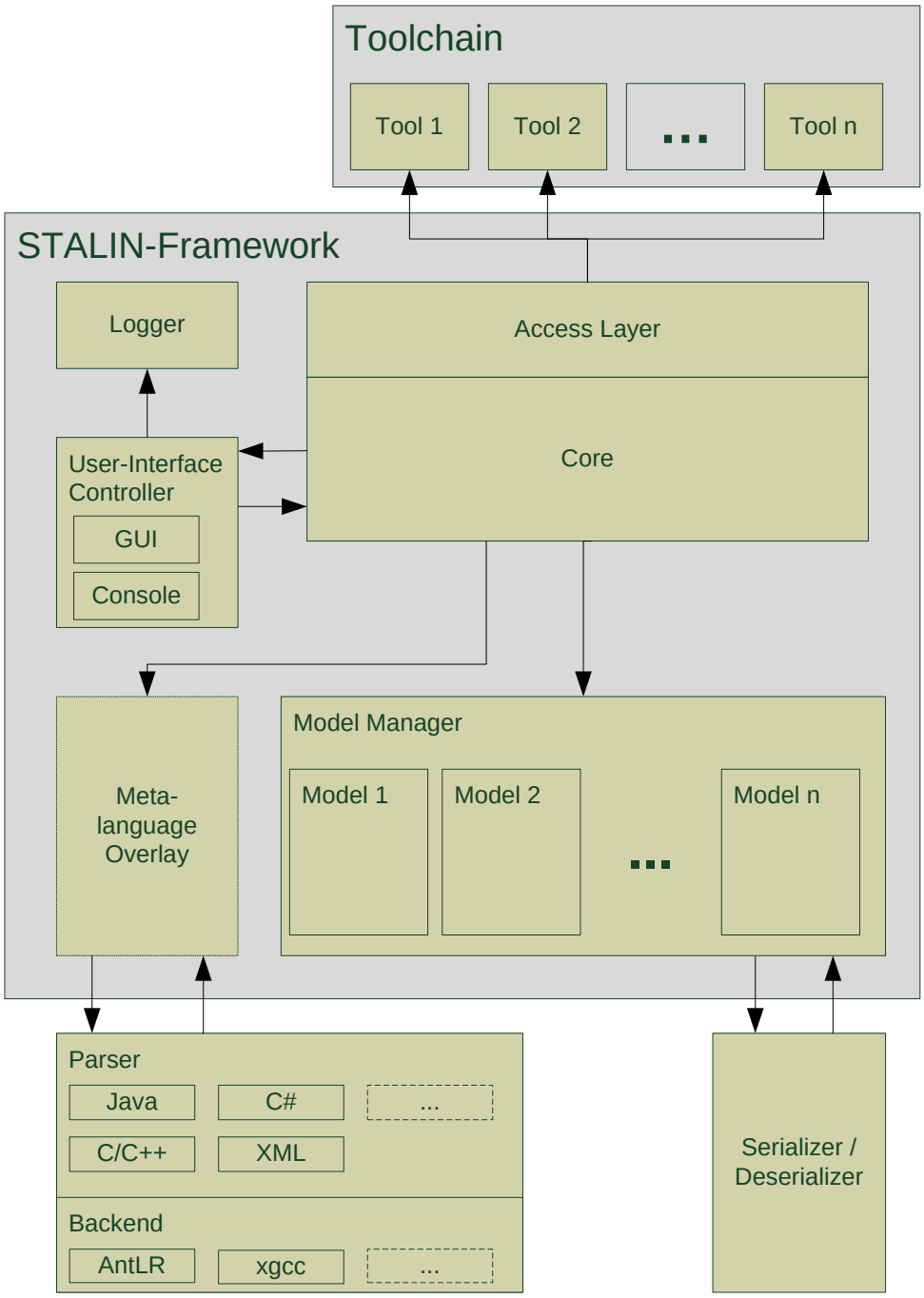

Figure 2.1: Modules of the STALIN-Framework.

loads the selected tools and a compatible parser (if more than one is present, the user will be requested to select one). During tool initialization the core determines the tool dependency and requests potential necessary parameters from the user. It also informs the model manager about the models it should later store and which tools are allowed to access or modify which of them. Now the core obtains a mapping from the parser token set to the internally used meta language token set and starts the parsing process for the given input. From this point on no more user interaction is required until the end of the toolchain execution. The different tools are called consecutively in the user-selected order by the core and are given their requested models node by node. Depending on their type, they are not able to modify or access all models stored in the model manager. During the execution of the toolchain our model manager can cache momentarily unused models to other media (hard disk or network for example) or swap parts of a larger model. The different tools can communicate through the annotations they made in the elements of the models they processed. They cannot call themselves dynamically to prevent excessive resource consumption.

In this chapter we outline the different parts that make up our framework. Our goal is allowing tool developers to focus on the algorithmic part of their ideas and to outsource some of the repetitive activities to our framework. We will briefly describe the most important aspects of our framework and how they work together while pointing out some of the challenges we had while designing and implementing it. Most of the time we follow an object-oriented approach and make extensive use of design patterns [21] where it is reasonable.

2.2 Meta Language

We stated before that we would need an universal language as a part of a relieving framework. STALIN supports such an universal language which we call a meta language. This meta language must fulfill the following requirements:

- Allow a well-defined transformation back to the input language.
- Preserve exact control flow data.
- Preserve type sizes.

Today there exist many intermediate languages for different environments and applications. They all share the goal of reducing multiple inputs to one common language which allows easy processing for special back ends. They are mostly used in compiler or interpreter specific applications. Examples of such intermediate languages include the Common Intermediate Language [22] for the .NET Framework, Java bytecode [23] for the Java Virtual Machine and the Register Transfer Language (RTL) [24] for the GNU Compiler Collection. All those intermediate languages are feasible in the context of compilation. They are used to reduce higher level languages to assembly languages and therefore serve their purpose well. But in the context of program analysis they can be rather useless because they do not allow a well-defined transformation back to the input language as we require. Without this requirement some tasks of program analysis become impossible. For example, imagine you want to develop a tool which corrects automatically certain runtime errors in program code. If this tool transforms the program code in one of the aforementioned languages and finds runtime errors there, how do you perform the automated correction? You have to keep

both models linked at all time and find transformations from the intermediate language back to all its possible input languages. This would be very tedious and would certainly prevail the benefits gained through the intermediate language.

There also exist approaches for high level intermediate languages like the C Intermediate Language (CIL) [13] which also comes with a number of tools for program analysis. Unfortunately this language supports only ANSI C as input and possesses a non-modular architecture. Therefore it is out of scope in respect to modern, object-oriented programming languages. Other high level intermediate languages are BoogiePL [25] and the ESC/Java intermediate language [26]. They origin directly from the model checking and verification community and as such they are already known and accepted. BoogiePL is for instance the input language for Boogie [27], the Spec# [28] static program verifier.

The main reason why using a high level intermediate language is more promising to us is because we can maintain more information about an input program. This allows us to preserve the appearance of the original code after a transformation into the intermediate language and back, and thus allows developers to use tools based on STALIN on the fly.
To get a better understanding for the inner workings of such transformations we sought to create an own intermediate language. Instead reducing our input languages to a language with less instructions we tried to create a union of possible programming language constructs. We did not finish a formal definition of this intermediate language as its creation is a very demanding and error-prone task. This has to be subject to future work and we could as well decide to use or extend one of the already existing intermediate languages from the model checking and verification domain.

However we had to integrate such a language in the architecture of our framework. This was accomplished by providing an interface for mapping input language constructs to meta language constructs. Those meta language constructs are essentially nothing more than a set of tokens taken from the already existing parsers we have integrated so far. Those tokens will form the basis of a syntax for the meta language.

We will only outline the idea behind this mapping here while the more technical details are described in section 2.7.

As STALIN relies on graph based models as working base, it needs methods to transform an initial program code to such a graph based structure. This is done by a parser/lexer combination which generates a parse tree [29] or concrete syntax tree. Normally one would not concern oneself with writing own parser and lexer; the use of a parser generators is more feasible. Such parser generators, like ANTLR [30], Coco/R [31] or GOLD [32] produce a parse tree with the grammar and the tokens of the input language. As we stated before, we do not want to loose informations about the original language but we also do not want to write algorithms specific to one single language. To resolve this conflict, we introduce a mapping overlay over the structure of the parse tree generated by our parser. STALIN supplies a static `TokenMap` class which contains every token recognized by our meta language. An implementation of a parser now needs to specify which of its tokens corresponds to one of STALIN's tokens. This relation can be a one to one mapping but can also contain replacement rules which lead to a one to many or many to one mapping. This replacement rules are essentially expressed by lists of tuples containing a list of tokens and a value much like in a regular expression. STALIN applies those rules on the fly while it traverses the model

structure. This leads to a language-transparent tree structure with nodes that belong to one of the following three categories:

- Nodes are labeled with a token from the token set of the meta syntax for one to one relationships. Tools can consult the `TokenMap` class to look up the original token.
- Nodes are folded nodes labeled with a token from the token set of the original language for one to many relationships. While the model structure is traversed, STALIN unfolds this node automatically and traverses the contained subgraph as it were the actual model structure. This subgraph contains only tokens from the token set of the meta syntax. It notifies the affected tool of this process through a state parameter.
- Nodes are folded nodes labeled with a token from the token set of the meta syntax for many to one relationships. STALIN does not unfold this node while traversing the model structure but informs the affected tool that the supplied node is a folded one thus giving the opportunity to unfold it.

Our approach does neither contain any semantic information beyond the original input language nor any syntactical rules. The attached parser has to ensure that the supplied tree is conform to the parsed programming language. It does also ensure that there is an easy way to replace existing intermediate languages with different ones through simply adding an additional mapping layer.

STALIN contains in its actual version parser for the programming languages C#, Java, C and C++. We consider those languages as most important for current program analysis tools because they contain every aspect of modern imperative

and object-oriented languages. Future work will include a more closer examination of those directly attached languages to identify possibly equal semantic constructs and problem classes for program analysis.

2.3 Model

The capacity to generate and store standardized, graph-based models is one of STALIN's main aspects. Every tool needs a model on which it can work, but most of the time this model is not an important issue for solving a certain problem. It would be better if you could get one which suits your needs for free. Also, it is vital to the communication with other tools that you can rely on the same mechanisms of writing and reading tool-relevant informations as well as be able to share models between your tools. For this reason we will provide a number of plugins for creating standard model types like control flow graphs, call graphs, dependency graphs and the like in the near future.

The foundation of our models, as displayed in figure 2.2 is a carrier structure consisting of a node and an edge interface, called `INode` and `IEdge`. They represent nodes and edges in all of our data structures. Both of them are derived from the interface `IElement`. Enclosed in this interface is the container structure, represented by the interface `IPayload` which is described in section 2.3.1. This structure bears every piece of information of the node or edge. The concrete implementations make excessive use of the proxy pattern to save resources and as a result every subpart of `Payload` (which is the largest memory consumer) is not initialized until first use. Because we try to eliminate the concern for performance for tool developers we do not allow direct access to our carrier structures. As a

result, any tool that wants to manipulate the structure must use the provided access layers (see section 2.5.1 for detailed informations). This means essentially that a tool will only see objects of the type `IPayload` through a model traversal.

Our model also consists of a class called `GraphType`. This class describes the general structure of the model and contains generic informations about the state of the model and is described in section 2.3.3. Because this class only describes the structure of a model, it is visible to tools. `GraphType` is again enclosed in the class `ModelContainer`, our container for a single model. It features different search and access methods for the model and a list of entry points. A `ModelContainer` object itself is only accessible from the model manager, which contains all instances of `ModelContainer` and is responsible for the space-saving operations with respect to scalability.

Our architecture emphasizes the decoupling of information and structure. This gives us the possibility of implementing space or runtime efficient data structures in the future without performing changes on existing tools. For now, we have chosen a very simple carrier structure which does not contain any optimization in our current version of the framework. Because of its simplicity and the intended transparency for tool developers we do not describe the structure any further in this work.

2.3.1 IPayload

The interface `IPayload` contains a two-dimensional hash map which serves as annotation space for the different tools. Every tool should select a keyword (by convention the tool name) to reserve a bucket in the first dimension, which yields

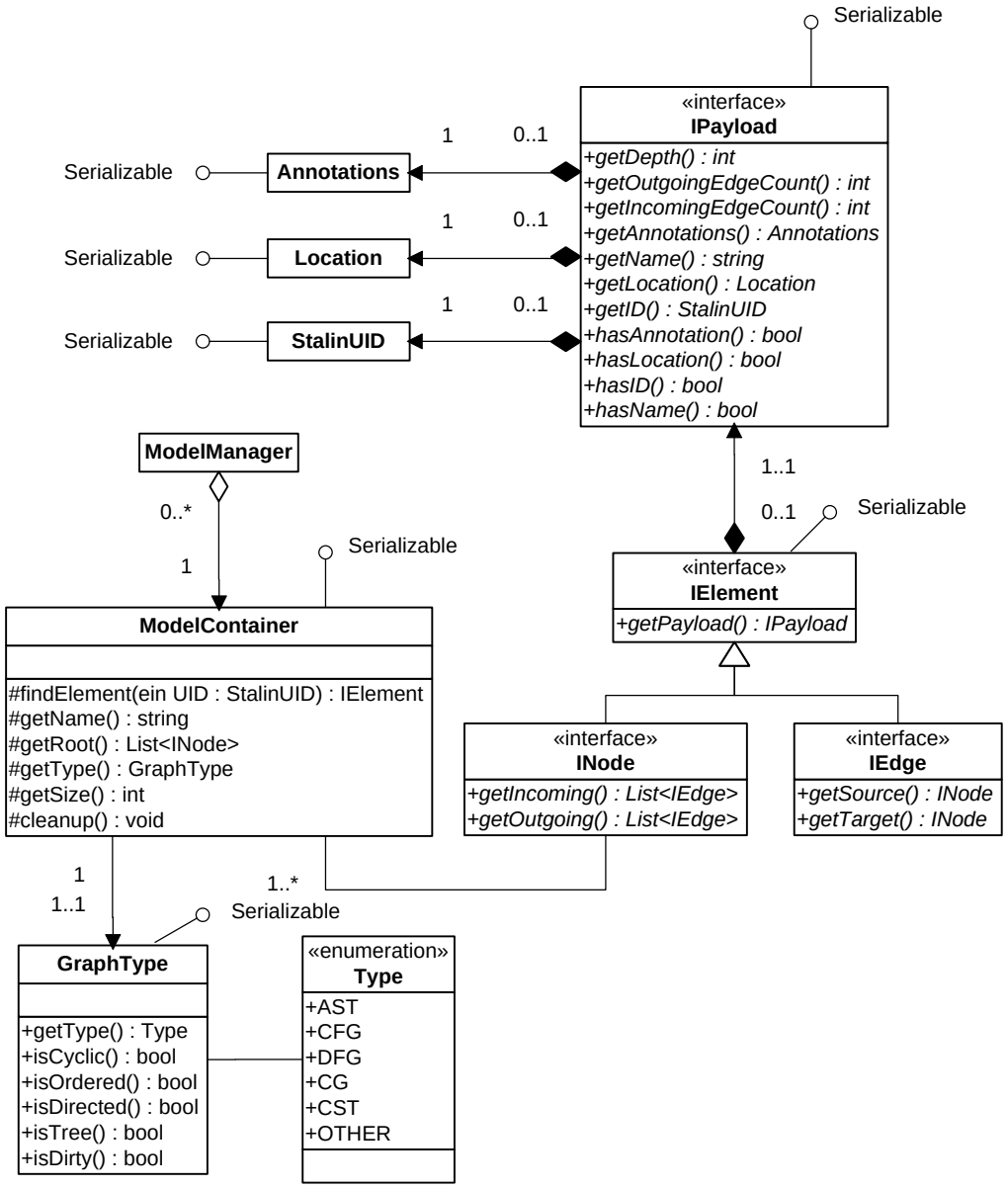

Figure 2.2: The model as an UML class diagram.

the second dimension, a normal hash map in which the tool can store every information it needs. The whole hash map is persistent during a toolchain execution, so every following tool or even the same tool, if it performs a multi-pass, can access and manipulate the informations stored in this map, in particular those informations stored by other tools. Right now we do not provide any standardized format for those annotations, but we are currently discussing the implementation of existing formats in the future (see chapter 4 for more information). STALIN does not control the information in the hash map and swaps only whole `IPayload` objects, so you gain full control over the information flow without any access layer. The same concept also applies to whole models, which are accompanied by a separate `IPayload` instance contained in the descriptive class `GraphType`. Additionally, every payload also contains a label and a number of structural informations like distance to entry point and the number of in- and outgoing edges. This can be important for tools with a heuristic approach or for measuring software metrics. Besides that a tool receives additional structural information through the callback methods of the `IObserver` interface which is part of our access layers. But for now we want to concentrate on the `IPayload` interface. The following is an overview of the methods provided by `IPayload`, which are accessible for all tools:

- `int getDepth()`

  Provides an integer which describes the distance to the current entry point of the model. This integer is the sum of nodes and edges of the shortest path from the current entry point to the node or edge carrying this `IPayload`. The current entry point refers to model types with multiple entry points, see section 2.3.3 for more details.

- `int getOutgoingEdgeCount()`

  Returns the number of outgoing edges for a node and -1 for an edge. Together with `getIncomingEdgeCount()` it yields the degree of a node. This information is computed by our factories and is never negative expect for edges. If the underlying model is undirected, both methods will return the same number, namely the count of all edges.

- `int getIncomingEdgeCount()`

  Similar to `getIncomingEdgeCount()` this method returns the number of incoming edges for a node and -1 for an edge. If the underlying model is a tree, every node returns 1 except the root node which returns zero.

- `HashMap<String, HashMap<String, Object>> getAnnotations()`

  This method returns the aforesaid two dimensional hash map for a node or edge. The method guarantees that the returned hash map is always initialized and that it is persistent through a toolchain exectuion. The hash map should be used to store annotations like predicate sets or connections to other models via their ids. The concrete implementation of this method follows the proxy pattern for `IPayload`.

- `String getName()`

  Returns the concrete label of a node or edge. This object will be displayed through our visualization tools (see section 2.4.3 for a description of those tools). It should be used for every label provided by a labeling function of a formal model or for the String representation of a token. If the label is not set, `getName()` will return an empty string.

- `Location getLocation()`

Provides a `Location` object which contains informations about the physical position of the node or edge in the input program. The `Location` object is described in section 2.3.2. The method guarantees that the returned `Location` object is always initialized and that it is persistent through a toolchain execution. The `Location` object is not writable for tools. If the structure of a model is changed in such a way that a change of the `Location` object is implied (for example due to a refactoring operation), STALIN takes care for the correct changes in the affected `Location` object. How this is achieved becomes more clear in the sections 2.3.2 and 2.5.1. The concrete implementation of this method follows the proxy pattern for `IPayload`.

- `StalinUID getID()`

  Returns a `StalinUID` object, which is essentially a wrapper for a Java `UID` object providing comparative methods. The method guarantees that the returned `StalinUID` object is always initialized and that it is persistent through a toolchain execution. Additionally, every assigned `StalinUID` is guaranteed to be unique, even after serialization and de-serialization. This is NOT guaranteed for a string representing the UID (in fact, every StalinUID of a model is recalculated when it gets de-serialized). The `StalinUID` object is always read-only, even for STALIN itself. The concrete implementation of this method follows the proxy pattern for `IPayload`.

- `boolean hasAnnotation()`

  Due to the implementation of `IPayload` as proxy, one will always get an initialized hash map when calling `getAnnotations()`. If there is no hash map at that time, `IPayload` will create an empty one and allocate resources

during this process. Because many operations need to check if there are annotations they can process you would jeopardize the suggested meaning of the proxy pattern if you check if `getAnnotations()` yields an empty hash map. On this account `hasAnnotation()` returns `true` if the hash map is already initialized and could therefore contain annotations and false if this is not the case. Please note that `hasAnnotation()` will also return `true` if the hash map is completely empty but initialized.

- `boolean hasLocation()`

  Similar to `hasAnnotations`, this method returns true if the `Location` object of `IPayload` is initialized and false if not.

- `boolean hasID()`

  Similar to `hasAnnotations`, this method returns true if the `StalinUID` object of `IPayload` is initialized and false if not.

- `boolean hasName()`

  Similar to `hasAnnotations`, this method returns true if the label of `IPayload` is initialized and false if not. Notice that an empty string counts as an initialized string.

2.3.2 Location

The `Location` object is enclosed by `IPayload` and describes the location of the corresponding element in a concrete file. Namely it contains the absolute path with regard to the machine and the operating system STALIN is running on, the package name if any and the line number where the element can be found.

Normally an `ISource` plugin (a parser, see section 2.7 for details) sets the initial location of every node and every edge. Because the output of such a plugin is typically an abstract syntax tree [29] or a parse tree, many nodes and edges will be marked with the exactly same `Location` object. Although this is not optimal in terms of memory usage, `Location` still remains a lightweight object consisting of two references and one primitive so that the memory load is negligible. We argue for its presence with some special cases where we need it and some general benefits through its usage. Special cases are for example the decoupling of merged files after you have used a whole project as input or some algorithms who need locations for their work. The binary reachability argument used by Terminator [33] for example requires cut points which are calculated with the help of a location information [34, 35, 36]. General benefits are for example easier reconstructing of changed input programs or connecting completely different models with the original source. Also, if you are planning to integrate an IDE into STALIN or vice versa, it is very useful to keep such an object around.

On the other hand it can be quite difficult to provide the necessary integrity of a location object. If a tool changes the structure of a parse tree, for example by inserting additional statements or by transforming complex ones to more simpler, STALIN needs to decide how it should update the line number. The normal rule of thumb is to update everything lower or more right than the changed element in the tree. But since such an operation requires in the worst case a complete walk through the model it can be a very costly update. Instead, STALIN updates every location in a model after the execution of one pass of a transforming tool. We assume for the update convenient formatting rules for source code and update the `Location` object accordingly. We do not limit the length of a line, since this

only serves human readability and we are concerned with practical handling for tool developers. You can find the descriptions for the tokens used in the rule listing in appendix A. They belong to our meta language which is described in section 2.2. The following rule listing describes our assumptions about the syntactical structure of the source code regarding line numbers. They are applied if a tool transforms the model structure in a way which needs the update of the `Location` object. If the respective tool supplies a location object for an element we will not perform an update for this element and base further update operations on this one. This means that this automatic location update will only step in if the transforming tool does not care for it by itself.

- Every *BLOCK* consists of one line for the opening bracket and one for the closing if any. Any statements or expressions contained in the block consist of at least one line for each.
- Every *FIELD* consists of one line containing the values for *MODIFIER*, *PRIMITIVE_TYPE* or *CLASS_TYPE*, *IDENTIFIER* and *VARIABLE_DECLARATION* if any.
- Every *METHOD* consists of one line containing the values for *MODIFIER*, *PRIMITIVE_TYPE* or *CLASS_TYPE*, *IDENTIFIER*, all *PARAMETERS* and additional language specific tokens (such as *THROW*), and at least one line for its block.
- Every *ASSIGN* is written in one line containing its *LEFT_HAND_SIDE* and *ASSIGNMENT_EXPRESSION*.
- Every *FOR* consists of one line for the keyword and all declarations, conditions and expressions and at least one line for its block.

- Every *WHILE* consists of one line for the keyword and the condition and at least one line for its block.
- Every *EXPRESSION* and every *STATEMENT* not captured by the rules above consists of one single line.

Please note we do not apply those rules during the parsing process, they are only used when a tool inserts new statements or transforms existing ones without declaring a new `Location` object. Our ProgramNormalizer tool (described in section 2.4.2) is intended for such situations and removes many of the more complex expressions and statements and tries to break up nested statements and expressions.

Nevertheless, there remain some unsolved problems with `Location`: Most of the algorithms for model checking do not work on the parse tree and therefore it is necessary to connect the `Location` object of the initial model, where we had a close relation to the `Location`, to the new, generated models. To achieve this, we exploit the ordering of a parse tree. Because generating tools must use our factories to generate new models and because we know the tokens they have seen in the tree when they call a generative method, we can connect new generated nodes with old `Location` objects. For one to one relationships between old and new model this is sufficient. But we also allow one to many and many to one relationships. The access layer contains special facilities to aid the generation of such relationships and their workings as well as the detection of a given relationship is described in section 2.5.3. They influence the `Location` object in the following ways:

- **One to many relationships:** The `Location` object of the source model

is simply referenced in all of the new elements.

- **Many to one relationships:** The new element is annotated with a `RangedLocation` object, which is a decorator for `Location` and contains collections for each field in `Location`. It retains all informations of the original `Location` objects and saves them in a collection consisting of triples of filenames, packages and a list of line numbers ordered by their appearance in the original model. It also saves the `Location` object where the generative method was called first and returns those values if accessed with the base class methods.

2.3.3 Supported Model Types

Program analysis is typically carried out on graph based data structures and therefore STALIN needs to support and distinguish numerous graph based models. The discrimination turned out to be more demanding in terms of implementation than expected. Because we also allow partial transformations on an existing model which change their type, it was necessary to abandon some principles of object-oriented programming to achieve maximal flexibility. Normally one would construct an object hierarchy to represent the different graph types. This would be for example, a graph object on top and derived from it directed and undirected graphs, which are superclasses to trees and undirected multigraphs and so on. The type of the graph with regard to graph theory would be inherent to the type of the underlying data structure. But problems arise if a change occurs to this structure that changes its inherent meaning. If, for example, a tool transforms a subtree of a tree structure through inserting a cyclic edge then the tree would be-

come a graph. Therefore, the type of the structure has to change, and not only in the subtree but in the whole model. We cannot solve this through a consolidation run at the end of the tool execution, because the current tree invariant forbids this transformation, so we have to change the whole model structure when the first tool operation violates the tree invariant. This operation is costly as it requires a rebuild of the whole structure, and thus we want to avoid it if possible. The solution is to separate the discriminating property of the model from the data structure and use a descriptive object for each model. This descriptive object is called `GraphType` and contains all discriminating informations. It uses numerous flags for the different attributes of a graph like cyclic, directed, multigraph and so on. Besides graph discrimination it is also used for some semantic informations that do not fit elsewhere like the order of the model or the finite property. For standardized models and easier access it also maintains an enumeration for common names of models and if not applicable a field for a custom name. The following list gives you an overview of the contained graph properties and the common names contained in the enumeration:

- Properties
  - *IsOrdered* describes if the ordering of the nodes and edges of the model serve a purpose or if they are arbitrary.
  - *IsDirected* distinguishes between directed and undirected graphs.
  - *IsCyclic* distinguishes between cyclic and acyclic graphs. Note that we do not guarantee that a model contains cycles if this property is true. It can also mean that we do not know if the model contains cycles.
  - *IsMultigraph* denotes that the graph is a multigraph, which means that

it can have multiple edges between two nodes or multiple loops on one node if it is also cyclic. Note that we do not guarantee that the model contains multiple edges between two nodes or loops on one node if this property is true. It can also mean that we do not know if the model contains multiple edges between a node or loops on one node.

- *IsFinite* marks the model as finite. If the model is marked infinite it can contain folded nodes which may contain themselves in a subgraph. Those cycles are unfolded while the model is traversed and can lead to an infinite sequence. Unfolded nodes are described in section 2.3.4. Note that we do not guarantee that the model is actually infinite when the property is false. It can also mean we do not know if it is finite.

- Common Names

  The properties used in this list refer to the STALIN properties with respect to their guarantees.

  - *AST* stands for abstract syntax tree. An AST is a finite and unordered tree.
  - *CG* stands for call graph. A CG is a finite, directed, cyclic and unordered graph.
  - *CFG* stands for control flow graph. A CFG is a finite, directed, cyclic and unordered multigraph.
  - *DFG* stands for data flow graph. A data flow graph is a finite, directed acyclic and unordered graph.
  - *CST* stands for concrete syntax tree (also known as parse tree [29]). A CST is a finite and ordered tree.

  - *TS* stands for transition system. A TS is a possibly infinite, directed, cyclic and unordered multigraph.
  - *PG* stands for program graph. A PG is a finite, directed, cyclic and unordered multigraph.
  - The keyword *OTHER* denotes that the current model is not one of the common models but a special one. If the OTHER keyword is set, you can obtain the name of the model (chosen by the model creator tool) by calling the `getName()` method of the `GraphType` object.

The following restrictions apply to our models:

- We do not support unconnected graphs in one graph structure. Every element of a graph structure must be reachable from one of the entry points of the model or it will be discarded. If you need unconnected graphs you will have to specify multiple entry points to your model and indicate their membership to another entry point in the annotations. We do allow not strongly connected directed graphs but there are some limitations while traversing them. The limitations are described in detail in section 2.5.1.
- Although we recognize possibly infinite data structures STALIN we do not guarantee termination on them. More precisely, if, for example, a transition system unrolls an infinite loop, STALIN will not terminate until the active tool sends a skip message or the system runs out of memory. You have to take that into account if you want to work on a possibly infinite model.
- STALIN does not support mixed graphs (graphs which contain directed and undirected edges).

- STALIN does not support hypergraphs (graphs where edges can have edges or "crossings").
- STALIN does not support loose edges, that is edges which have only one node attached to one of their endings.
- STALIN does not support null graphs or empty graphs. Every model must have at least one entry point or it will be discarded.

2.3.4 Folded Nodes

Folded nodes are a special case in our data structure. To enable support for infinite models and for certain lazy approaches we had to find a way of preventing STALIN of computing complete models in every pass. Folded nodes appear as normal nodes with adjacent edges in a model. But they also contain complete subgraphs or methods to inline subgraphs in an existing graph. The interface `IFoldedNode` is derived from the interface `INode` and extends `INode` with the `getUnfoldedNode()` method. This method returns the entry point to the contained subgraph. Every subgraph described by a folded node can have only one entry point. This is no real limitation because one can use several folded nodes to obtain multiple entry points for folded nodes. Nevertheless the implementation of the `getUnfoldedNode()` requires some care taking from a tool developer, because the exit points of the contained subgraph have to be specified. One has to explicitly connect the desired exit nodes to the outgoing edges of the folded node while bearing in mind that every adjacent edge of the folded node must be used as adjacent edge for the contained subgraph. You should also consider that it is possible to change the breadth-first order of the graph if there is more than

one exit node, that is a node with adjacent edges to nodes outside the subgraph. In figure 2.3 you can see the different breadth-first orders resulting from multiple exit nodes. You will notice that the original nodes 3 and 4 are now the nodes 7 and 6. Their order has been reversed.

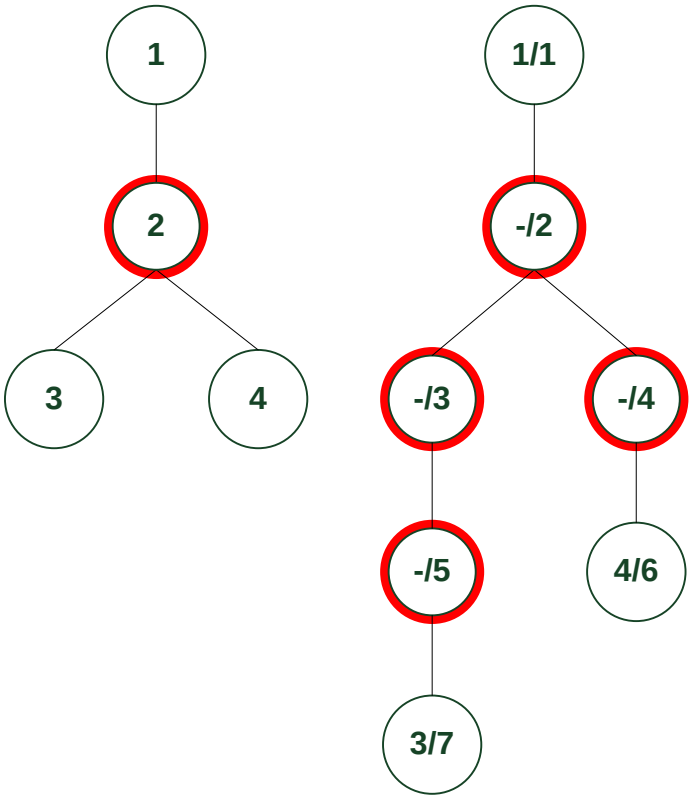

Figure 2.3: The left graph contains a folded node marked red, the right graph shows the same graph with the node unfolded. The original breadth-first order is the first number inside a node, the new order is the second.

When STALIN meets a folded node while traversing a graph, it asks the current tool what to do with it using a callback method. The tool can choose to unfold the node, to treat it as a normal node of the model or to access its unfolding rules or the contained subgraph. If it unfolds the subgraph the traversal is continued on the resulting new graph obeying the resulting traversal order. If the subgraph contains another folded node (possibly the same for infinite recursion) STALIN

will ask again if it should unfold the node or not. More information on the traversal of folded nodes can be found in section 2.5.1.

2.3.5 Model Manager

Until now we did not mention the inside workings of the model manager at all. His part is an important one, but as we address mainly tool developers it is not necessary to explain those workings in detail because it is sufficient to view it as a black box. Also, this part of the framework is currently undergoing rapid changes because it serves as abstraction layer between the STALIN core and the actual model representation. Therefore we do not explain the technical details of the model manager but give instead an overview of his tasks in STALIN.

The STALIN model manager has many roles: First of all, it holds all models present at runtime. It keeps track of the size of a model and decides if and when parts of it should be serialized. Second, it provides a variety of access methods to its models for the core. Third it provides static search functions for interactive tools. Those search functions are explained in section 2.5.2.

As we described before, the model manager encapsulates together with the factories the concrete implementation of our model structure. It is responsible for the access to those structures through the walkers. Every model forms through the class `ModelContainer` an entity which can be partly or as a whole serialized. The model manager utilizes this ability to cache the models dynamically through an attached `ISerializationPlugin` which you can see in figure 2.4. The interface provides methods for serializing and deserializing models or parts of it. The implementations of this serialization plugins decide on their own where or how

they store the received data.

The memory manager has different caching strategies based on the expected runtime for an traversal, the expected size of new models, the expected number of new models and the available memory. The model manager calculates the dependencies between the different tools contained in the toolchain based on the requested and needed models. The following caching strategies are currently implemented:

- If the expected size of all models together does not exceed the available memory there is no need to serialize a model. The model manager will keep every model in the memory and updates the calculation after every tool execution.
- When the expected size of all models exceeds the available memory, the model manager tries to serialize those models which will not be used anymore in the toolchain. If every model will be used, the model manager serializes the model which will be used last.
- Until a model reaches a size greater than a threshold value (a percentage of the total available memory) it is always serialized as a whole. Models greater than this threshold may be serialized partly while STALIN traverses them. Especially if a single model is larger than the available memory, parts of it may be serialized. The methods to partition the model are part of the serialization plugins.

2.4 Tools

In this section we describe the different plugin interfaces, their structure and how they are used to build a toolchain. Also we explain the categories and their associated interfaces which we use to divide tools based on their functionality. Because every STALIN plugin is also an Eclipse RCP plugin we will give a short introduction how you can connect a tool to the RCP extension points provided by STALIN using Eclipse and PDE [37].

First, consider figure 2.4: As you can see, the interface `IRCPPlugin` is the root of our inheritance hierarchy. It supplies the plugins with the methods `getName()`, `getPluginID()` and `init()`. As the name of the method `getName()` suggests, it provides a name for every plugin. This name is displayed in various dialogs and has no other influence on the framework. Different from that, the method `getPluginID()` is used to separate plugins from each other. This method should return an unique ID for each plugin loaded by STALIN and the same as used by the Eclipse RCP framework. The `init()` method is called by the STALIN core after it has loaded a plugin through the Eclipse RCP. The core itself does not implement this interface as he is the main application of the STALIN framework and thus loaded directly by the RCP classloader. He contributes to the `org.eclipse.core.runtime.applications` extension point provided by RCP. Derived from `IRCPPlugin` are four other interfaces, namely `ILogDisplayPlugin`, `ISerializationPlugin`, `IStalinPlugin` and `IControllerPlugin`. We will only discuss the function of the `IStalinPlugin` interface here, as the others are already covered in other sections. For the `IControllerPlugin` and `ILogDisplayPlugin` interfaces refer to section 2.6 while the `ISerializationPlugin` interface is de-

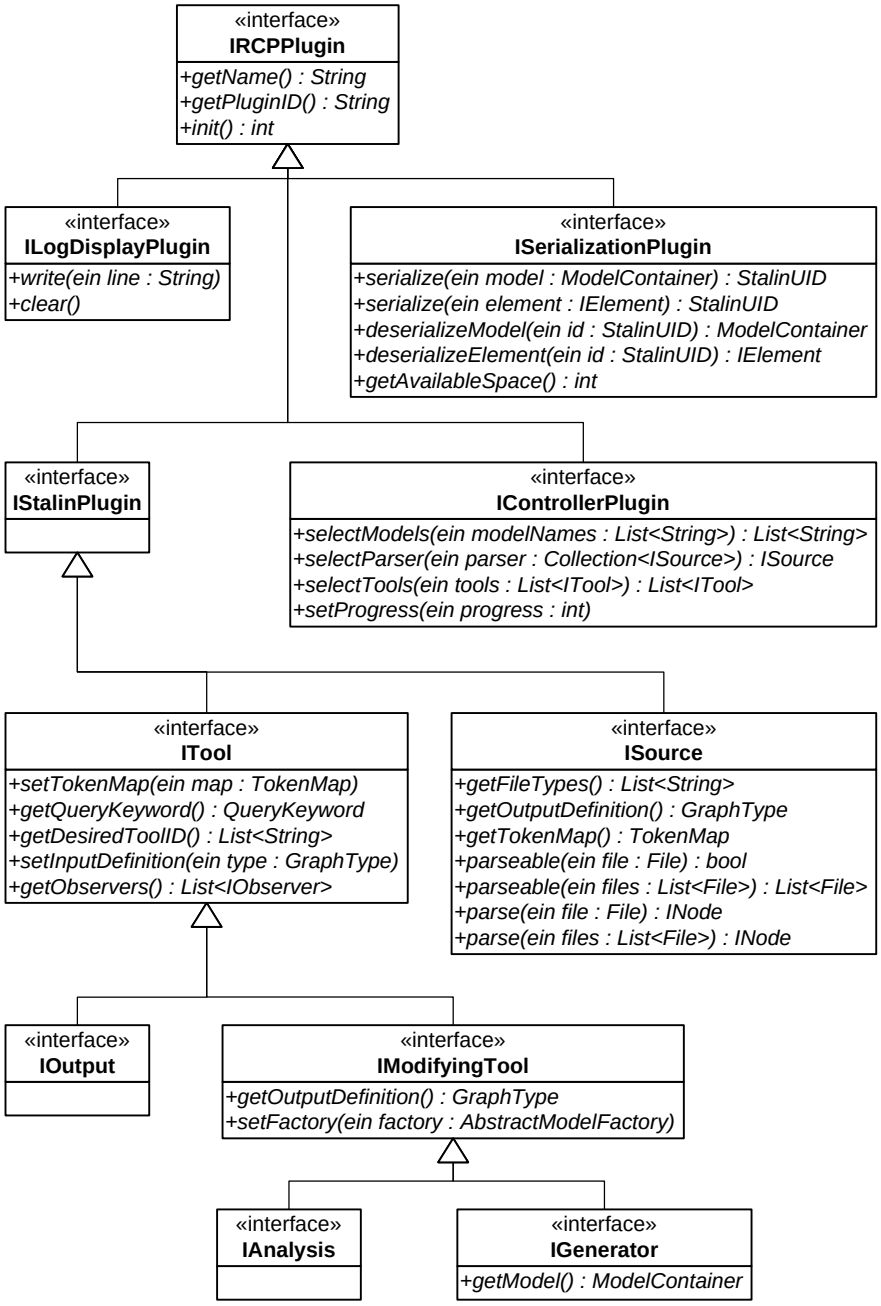

Figure 2.4: The different plugin interfaces as an UML class diagram.

scribed in section 2.3.5. `IStalinPlugin` itself does not add any methods to our plugins but instead classifies external contributions. Derived from it are the interfaces `ITool` and `ISource`. We do not mention the `ISource` interface here any further as it does not define a tool in the sense of our framework. It declares parser which can, of course, also be supplied by a tool developer. Why this is separated from our understanding of tools and what its inner workings are is discussed in section 2.7.

Our focus in this section lies on the `ITool` interface and its methods. Every plugin considered a tool by our philosophy implements `ITool` through one of his derived interfaces. A tool is by our understanding a part of code which consumes a present model, works on it and produces some sort of output. STALIN provides in this context the model and different ways to process the produced output. Regardless of what the work of a tool is, it does it by providing algorithms. Those algorithms have to be supplied to STALIN with minimal effort and maximal freedom for the tool developer. Because an algorithm deals with work in a step-by-step fashion, it is only natural to want something to do the single steps for one. There is no need to program traversal operations through a model, because certain standardized methods cover every possible traversal. STALIN provides those standardized methods and some more; they are described in section 2.5. The development of an algorithm to solve a certain problem is to our belief the core of every computer science researcher's work. Hence we want to eliminate the need of implementing unnecessary infrastructure components for a tool developer. One should be able to implement an algorithm like one would present it as pseudo code, as sequential and imperative structure. We apply the observer pattern to separate the implementation of the algorithm from the infrastructure tasks. This

application and the requirements for an `ITool` implementation as well as the intended usages of its methods are described in section 2.5.

The tool concept alone is too generic to handle all possible scenarios of tool applications with STALIN. We needed a way to express differences between tools and decided to categorize them into three groups:

- The model-generating tools represented by the `IGenerator` interface and referred to as "'generators"'. They cannot change the structure of the model they are working on but generate new models as output.
- Tools that do not generate a new model but can change the structure of the model they are working on are called analysis tools and represented by the `IAnalysis` interface.
- Tools that neither change the structure of the model they are working on nor produce new models as output are represented by the `IOutput` interface and referred to as output tools.

We choose those categories based on the operations which would be possible for each. The separating criterions are operations which change the model structure and the creation of new models. As both operation classes are very costly to maintain, we do not want to provide support for them if it is not strictly necessary. Note that although only analysis tools can change the structure of a model, every tool can modify the annotations of every element of a model. The execution of those tool is organized by a toolchain. A toolchain is a set of tools selected by the user. This set is ordered by the user, but this order is not necessarily respected by STALIN. If it is possible, STALIN will rearrange the execution of the tools

to provide a minimal memory footprint. STALIN tries to discard intermediate models whenever possible. Also, it tries to keep caching operations to a minimum. It does so through carefully evaluating the input and output definitions as well as the model requests provided by each tool. Those mechanisms are described in section 2.5 for tools and in section 2.3.5 for STALIN.

A toolchain execution is always a sequential execution of the contained tools. We do neither support nor encourage the alternating execution of tools. The reason behind this philosophy is again an efficient memory management and more controll over the resource allocation of single tools. We simply could not fulfill our goals if we would allow alternating calls, because they would lead to an unpredictable behavior of toolchains and thus to an unpredictable resource usage. Therefore we need to recognize some currently unresolved challenges in our approach. Many newer tools in the program analysis domain make extensive use of lazy approaches [38,39] to solve particular problems more efficiently. Although our current implementation supports the use of lazy techniques inside a single tool, it does not support those lazy approaches between multiple tools. The main reason for this is that we have right now no no feasible way of looping a toolchain or parts of it for an indefinite time. This restriction is subject to future work and possible solutions will be discussed in chapter 4.

The next subsections will contain a brief introduction to the three tool categories and their intended usage. They also contain some examples for tools from the respecting category and their workings. Because we discuss the inner workings of the different methods for accessing and generating models in the section 2.5, this section will only give a brief overview of the mechanisms involved. It should be sufficient to understand the principles behind it, but for a more technical

description refer to the section stated before.

2.4.1 Generators

The Generators category consist of tools which generate one or more new models. They have to implement the `IGenerator` interface which supplies a number of methods to aid this process. Generators have to state what kind of model structure they will generate by implementing the `getOutputDefinition()` method provided by the `IModifyingTool` interface. They receive an appropriate factory for creating the model they have specified through their output definition by the method `setFactory(...)`. After their execution, the core retrieves the created model through the `getModel()` method. The use of a callback method may seem odd at this point because STALIN oversees the creation process through the provided factories. But this method is needed to separate intermediate model formats from the actual model as one will see in section 2.5.3. Note that regardless of how many observers a generator tool provides, it can only contribute one new model to STALIN. This is done to encourage the separation of tools into smaller chunks. It should serve reusability and maintainability of the related code. Note that this does not restrict the numbers of graphs per model. Besides the stated methods there are no more differences to the general tool execution procedure.

At the present time there exists no final version of a tool from this category, but we are working on different ones which will create standardized models for STALIN. The responsible maintainer and lead developer for the following tools is Daniel Dietsch:

- **CSTtoCFG**: This generator creates an control flow graph (also known as flow graph [29]) from a concrete syntax tree. It takes besides an arbitrary CST a list of entry methods and constructs a CFG for every entry point in the list. Unresolvable calls, like system calls or library calls, are treated as atomic block in the control flow graph.

- **CSTtoDFG**: This generator creates a data flow graph [40, 41] from a concrete syntax tree. The data flow graph can be used to recognize stereotypical patterns in program code and is subject to future work.

- **CSTtoTS**: This generator creates a finite transition system [42] from a concrete syntax tree. It uses folded nodes to obtain a finite model of an arbitrary input program. Unresolvable calls are treated as atomic block in the transition system.

2.4.2 Analysis Tools

Analysis tools can change the structure of a model but do not generate a new one. As such, they fulfill the interface `IAnalysis` which is derived from `IModifiyingTool`. The `IAnalysis` interface does not contain any method but serves instead as a discriminator. Only analysis tools are allowed to change existing model structures. This restriction ensures a stricter separation and encapsulation of involved algorithms. In contrast to the generator tools they are allowed to change multiple models as long as they are matching their respective model selection criteria.

Like for the generator category, there are currently no final tools available for this

category. The following ones belong to this category and are under development:

- **ProgramNormalizer**: The ProgramNormalizer provides methods for syntactical transforming complex, language specific constructs in concrete syntax trees into simpler ones. Similar to CIL's program transformation, it tries to minimize a given input in such a way that it contains as few different constructs as possible. It does so by
  - transforming different loops to `while{true}` loops with explicit `break` statements,
  - transforming postfix operators to infix,
  - lifting all local variables to globals,
  - renaming every method and variable unique,
  - separating mixed assignment and declaration statements,
  - transforming all assignments which contain mixed expressions (like arithmetic operations mixed with function calls) to separate ones,
  - making references to different name spaces explicit.

  The tool is completely configurable, which means that every feature can be turned on or off. The development and maintenance of ProgramNormalizer is led by Justus Bisser.

- **StaticSlicer**: The StaticSlicer can generate static forward and backward slices [43] of arbitrary input programs. It relies on the data flow graph generated by CSTtoDFG. In doing so, it can produce two types of output: First, it can rely solely on the DFG and compute the program slice in a supported language while ignoring the original control flow up to semantic

equivalence. Second, it can take the preceding CST as additional input and produce a program slice with equal control flow in the original input language. The development and maintenance of StaticSlicer is led by Daniel Dietsch.

- **ATCML**: The **A**utomatic **T**ermination **C**hecker for **M**ultiple **L**anguages is a re-implementation of the existing Terminator tool which proves termination of C programs. The goal of this tool is to extend the set of input languages for Terminator with Java and possibly other languages. The development and maintenance of ATCML is led by Evren Ermis.

2.4.3 Output Tools

Output tools can neither change the model structure nor generate new models. Their intended usage includes pretty printers, file writers and input creation. Their interface `IOutput` does not any method to the `ITool` interface and thus they are the simplest tool category.

The following tools belong to this category, and except for JavaOut their development is beyond the beta stage:

- **Graphview2D**: This tool visualizes an arbitrary model structures with parts of its payload in 2D. It uses the Grappa library [44] in conjunction with system calls to the Graphviz tool dot [45]. It supports the following features:
  - displays every model structure supported by STALIN.

  - shows the complete payload for a selected element of the model in a separate RCP view.
  - can export dot files which represent a given model structure.
  - supports export to gif, jpg, svg and ps.
  - every feature supported by Grappa like zooming, panning, selections, etc.
  - customizable colors for different token groups.

  This tool was one of the early tools for STALIN and thus made it through various iterations of the framework. It has seen many architecture changes and thus suffers from scruffy programming and cluttered documentation. Currently, PrefuseVisualization is catching up with it and will hopefully replace it soon. The development and maintenance of Graphview2D is led by Daniel Dietsch.

- **PrefuseVisualization**: The PrefuseVisualization tool displays analogue to Graphview2D an arbitrary model structure with parts of its payload in 2D. It uses the Prefuse toolkit [46] to achieve this without utilizing system calls. PrefuseVisualization is considerably faster than Graphview2D and scales better. Nevertheless, it lacks some of the export features of Graphview2D and thus it is not superior to his older brother. It supports the following features:
  - displays every model structure supported by STALIN.
  - shows the complete payload for a selected element of the model in a separate RCP view.

  - provides different layout styles (horizontal, vertical and circular spreading).
  - allows expanding and folding of model structures in some layouts.
  - calculates only the visible part of the visualization and thus scales much better.
  - supports zooming, panning, selection, moving of elements, etc.
  - customizable colors for different token groups.
  - customizable font families, styles and sizes for every element.
  - customizable general color schemes.

  The development of PrefuseVizualization was led by Roman Matthias Keil. The ongoing development and maintenance positions are currently vacant and are in the meantime taken by Daniel Dietsch.

- **ConsoleOut**: ConsoleOut prints tree model structures to a console. It can also print the complete payload of every element. It will send its output to any attached `IILogDisplayPlugin` instance (see section 2.6 for a description). Currently, we do not support the output of other structures than trees with this tool. The development and maintenance of ConsoleOut is led by Daniel Dietsch.

- **JavaOut**: The JavaOut output tool is designed to convert an arbitrary input language to Java 6 code while preserving semantic equivalence between the programs. It depends on the ProgramNormalizer tool to reduce complex language constructs. It cannot translate calls to external libraries, especially it can currently not translate framework based methods or types. To some extend this will be possible through simple mapping operations.

We currently do not know if there exist feasible transitions between all Java and .NET language elements and framework features. The IKVM.NET [47] project could provide a good solution which has yet to be evaluated. Other, currently only partly solved challenges are the transitions from languages supporting pointers, using explicit memory allocation or multiple inheritance like C and C++. The development and maintenance of JavaOut is led by Justus Bisser.

2.4.4 Extension Points

This subsection completes our overview over the different tool categories with some informations about the RCP extension points and their usage in STALIN. For more detailed information about RCP refer to [18, 16].

The STALIN core propagates RCP extension points to provide loose coupling between the different components of our architecture. Extension points are an integral concept for the RCP framework. They are provided by the OSGi specification [48] which servers as component model for the Eclipse RCP framework. Every contribution or extension of RCP through new code, like STALIN, must dock to an existing extension point and can provide own extension points. A declared extension point consists of an entry in the plugin.xml of the providing class (in our case the core) with a reference to a schema file. This schema file contains informations and constraints for each extension point. This includes descriptions and names as well as which interfaces have to be present in an implementing class. We use those constraints to expose our interfaces to tool developers. Furthermore, we receive some useful benefits from the extension point mechanism

like lazy loading and better scalability for a large plugin base. We defined so far seven different extension points for STALIN. They are bundled in the package `local.stalin.ep` and use equally named schema definition files. You can see their binding with our interfaces in table 2.1.

Extension point ID	**Required interface**
`local.stalin.ep.analysis`	`IAnalysis`
`local.stalin.ep.generator`	`IGenerator`
`local.stalin.ep.source`	`ISource`
`local.stalin.ep.output`	`IOutput`
`local.stalin.ep.logdisplay`	`ILogDisplayPlugin`
`local.stalin.ep.controller`	`IControllerPlugin`
`local.stalin.ep.serialization`	`ISerializationPlugin`

Table 2.1: Relation between extension points and plugin interfaces

2.5 Access Layer

In this section we give details of the access methods that complement our model structure from section 2.3. During the design process we found that there is a conflict between maximal interoperability among tools and providing efficient data structures. The toolchain idea seems to limit some aspects of direct collaboration among tools like mutual calling, lazy approaches and multi-pass algorithms. In our approach the access methods compensate for this conflict and retain a preferably optimal balance. At the same time it provides easy access to our models. We will start with a short overview of the access layer structure and then describe the chosen methods with their problems and benefits.

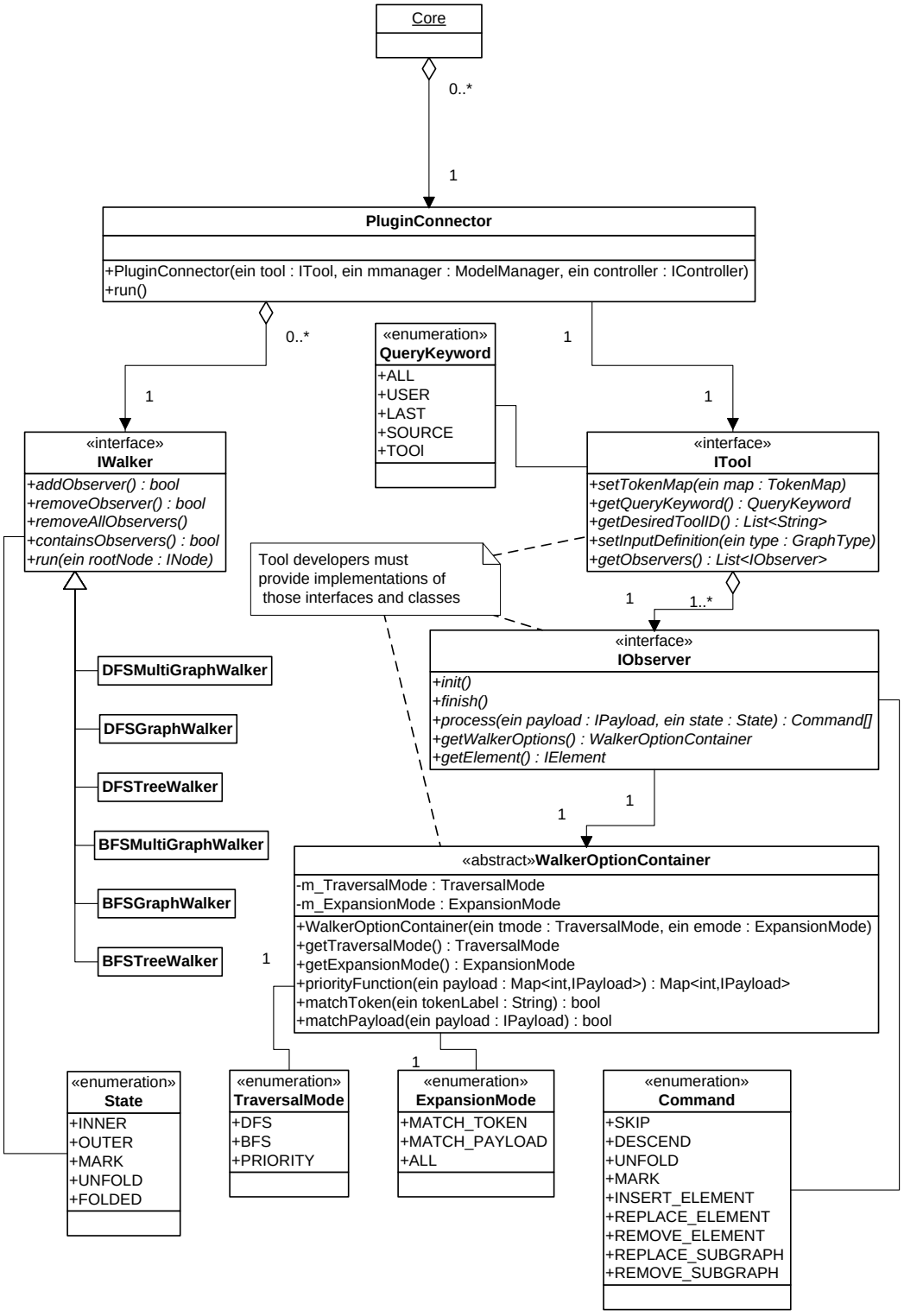

Figure 2.5: The access layer as UML class diagram.

2.5.1 Traversing Models

Figure 2.5 shows the class structure of our access layer. The central class is called `PluginConnector`. This class connects each tool with the core and handles the selection of the appropriate `IWalker` implementation (referred to as walker). The order of the tool execution is controlled by the core while `PluginConnector` performs the execution for each tool. This is done by initializing a `PluginConnector` with references to the tool in question, the current model manager and the current UI controller. The controller is only used for displaying error messages and selection dialogs. Its workings are described in section 2.6. The core calls then the `run()` method of the `PluginConnector` to start the tool execution. Afterwards, `PluginConnector` calls the `getQueryKeywords()` method of its tool to decide which model should be used for this tool. It receives one of the following elements of `QueryKeyword` and acts accordingly:

- *ALL*: This keyword tells the `PluginConnector` to use the tool on all present models. `PluginConnector` will execute the tool for each model provided by the model manager in the ascending order of their modification or creation. If this keyword is given, STALIN will obey the execution order of the toolchain with respect to the current tool. It will execute every tool before the current tool in an arbitrary order but guarantees that every previous tool has finished its execution before calling the current one.

- *USER*: The `PluginConnector` will open a dialog through the supplied UI controller and present all available models to the user. The user can then select one or more models from the list on which the `PluginConnector` will execute its tool in the order of the selection.

- *LAST*: The `PluginConnector` will execute its tool on the model which was created or modified last. If this keyword is selected STALIN will obey the execution order of the toolchain with respect to the current tool. It will guarantee that the delivered model is the model created or modified last in the order of the toolchain.

- *SOURCE*: The `PluginConnector` will execute its tool on all models created by `ISource` plugins (see section 2.7 for a definition of `ISource`). If there are multiple, the criterion matching models present the tool will be executed on them in the ascending order of their modification or creation.

- *TOOL*: If the *TOOL* keyword is specified, the `PluginConnector` will call the `ITool` method `getDesiredToolID()`. This method should return a list of tool IDs which specify other tools which create or modify models and are present in the toolchain. The `PluginConnector` will then execute its tool on those models in the order given by the tool ID list. If a tool from the list has created or modified multiple models then they will be processed in ascending order of their modification or creation. STALIN will ignore any tool on the tool ID list if they are not in the toolchain.

The `PluginConnector` will throw an exception if the criterions supplied by those keywords lead to an empty set of models.

After the determination of the input models for the tools the `PluginConnector` calls the `ITool` method `getObservers()`. The method returns a list of `IObserver` instances. We call such an instance of `IObserver` just observer. Those observers are the workhorses of each STALIN tool. They contain a `WalkerOptionContainer` object which specifies the traversal modalities and con-

tains callback methods for different modes. For each observer this object is retrieved by the `PluginConnector` who in return initializes the appropriate walker and links the observer to it. Which walker is selected depends upon the keywords returned by the `getTraversalMode()` and `getExpansionMode()` methods. First, the `getTraversalMode()` method is called and returns a keyword which determines the traversal mode. We currently support three different modes:

- *DFS*: The DFS mode uses a depth-first search to traverse the model structure.
- *BFS*: The BFS mode uses a breadth-first search to traverse the model structure.
- *PRIORITY*: The priority mode uses a priority function to examine the payload of elements of the model structure before they are explored. This priority function has to be provided by the `priorityFunction(...)` method. It has to assign every explorable model element an integer value. Explorable model elements are those that are adjacent to the last explored element that does not already has an assigned value. STALIN will then explore the element with the smallest value. If multiple elements share the smallest value, STALIN explores the one who has received his value first.

Only one of those modes can be active in one traversal. After the traversal mode has been set, the `PluginConnector` determines the expansion mode with the keyword combination returned by the `getExpansionMode()` method. This setting determines if STALIN performs a nested traversal or not. Again, there are currently three different keywords:

- *ALL*: The ALL mode is the default expansion mode. STALIN explores every element of the model structure and does not perform a nested traversal.
- *MATCH_TOKEN*: The match-token mode allows to control which elements of a model are presented to the observer. The `matchToken(...)` method is used to decide this. It takes a String representation of the current element's token as input and returns a boolean value. STALIN performs a traversal of the model according to the traversal mode determined before. It applies the method to every element label on its way and presents an element to the observer if the method returns true. We refer to this traversal as the outer traversal. After finding an element, STALIN starts a new traversal called inner traversal. Again, this traversal is determined by the current traversal mode. The element which led to the invocation of the inner traversal is called the transit element.
- *MATCH_PAYLOAD*:The match-payload mode works similar to the match-token mode. The only difference is the method used to decide if an element matches or not. This mode uses the `matchPayload(...)` method which receives instead the token of an element the whole payload.

There is more to the two modes match-token and match-payload. They both perform a nested traversal of the model structure and have therefore a different behavior. First, if one of the decision criteria of an element is not present (i.e. if the payload or the label of an element is null), STALIN assumes that it does not match and continues its search. Second, the transit element is presented twice to the observer. This is done to enable the steering of the outer traversal through a tool. We explain the workings of this special case in this section after we have described the general traversal steering facilities of our observers.

The `IObserver` interface provides five methods. We have already described the `getWalkerOptions()` method which returns the `WalkerOptionContainer` instance. Before an observer is executed through a walker, STALIN calls the `init()` method of the observer. Similar, after the execution is finished, the `finish()` method is called. STALIN does not require those methods, but start and end notifications are useful to prepare the observer with precomputed informations in a multi pass algorithm or to consolidate the processed informations after a completed traversal and before the final output is requested. The essential method of each observer is the `process(...)` method. A currently active walker calls this method of the attached observer and supplies it with its current state as well as the payload of the current element. The method delivers in return a keyword which steers the traversal of the model structure. This keyword is called a walker command. The following commands are available to all tool categories:

- *SKIP* informs the walker that it should not expand the subgraph of the current element. This command refers in an undirected graph to the unexplored part of the graph reachable only from the current element. This command cannot be combined with the *DESCEND* command.
- *DESCEND* informs the walker that it should continue its descend into the graph structure. This command cannot be combined with the *SKIP* command.
- *UNFOLD* requests the unfolding of an folded node. The walker supplies the element which would be the next in the model with the unfolded graph present in the next step. This command is only valid if the state of the walker is *FOLDED*.

- *MARK* tells the walker to mark the current element. After processing the subgraph of the current element, the walker will switch into the *MARK* state and present this element again. There can be multiple *MARK* operations before the walker switches to this state. In this case the elements are presented in the order they were marked. This command cannot be combined with the *REMOVE_ELEMENT* or the *REMOVE_SUBGRAPH* commands.

Our analysis tools need a way to change the structure of a given model. Therefore we have some additional keywords available for them:

- *INSERT_ELEMENT* instructs the walker to call the `getElement()` method of the observer. The method returns an element, which is inserted into the model structure as unexplored, adjacent element of the current one. The position is determined by the return value of the observer method `getPosition()`. This position is relative to the depth-first search order. This command cannot be combined with one of the other structural changing commands.

- *REPLACE_ELEMENT* instructs the walker to replace only the current element but not its connections to other elements. Again the walker calls the `getElement()` method to receive the substitute for the current element. The new element can be the entry point for a subgraph, and in this case the `getPosition()` method determines where the new connections are inserted. This command cannot be combined with one of the other structural changing commands.

- *REMOVE_ELEMENT* instructs the walker to remove only the current element. Its connections are attached to an adjacent element determined by the `getPosition()` method with respect to the current search order. This command cannot be combined with one of the other structural changing commands.

- *REPLACE_SUBGRAPH* is similar to the *REPLACE_ELEMENT* command. But instead of preserving it discards the connections of the current element. This command cannot be combined with one of the other structural changing commands.

- *REMOVE_SUBGRAPH* is similar to the *REMOVE_ELEMENT*. But instead of removing only the current element, it removes the current element and its complete subgraph. In an undirected graph the subgraph is the unexplored part of the graph reachable only from the current element. This command cannot be combined with one of the other structural changing commands.

STALIN will throw an `InvalidCommandException` if those commands are supplied by a tool from one of the other categories. The commands can be mixed if they are not contradictory. For example, a combination of the skip and the descent command makes no sense and consequently, STALIN will throw an `InvalidCommandException` if this combination is actually used.
The structural commands can result in a situation where the replacement for a node is requested but instead an edge is delivered. If this is the case, STALIN will throw an `InvalidElementException`. A tool developer can determine if the needed element is an edge or an node by calling the `getIncomingEdgeCount()` method of the current payload .

The walker commands are one aspect of the traversal control provided by STALIN. The other aspect is the state of the walker which is also supplied with every call to the `process(...)` method. They signal important changes in the model structure which are otherwise not noticeable by the observers. The following commands can be received from a walker:

- The *INNER* state can occur if a expansion mode with nested search was selected. The state tells the observer that the walker presents a transit element and performs the inner search. This indicates that the returned walker command will refer to the inner search. This state can be combined with one of the other states except *OUTER*.
- The *OUTER* state can occur if a expansion mode with nested search was selected. The state tells the observer that the walker presents a transit element and performs the outer search. This indicates that the returned walker command will refer to the outer search. This state can be combined with one of the other states except *INNER*.
- The *MARK* state notifies the observer that the current element is an previously marked element. This also indicates that the element was already seen and that the exploration of its subgraph is complete. This state can be combined with one of the other states.
- The *UNFOLDED* state informs the observer that the current element results from the unfolding of an previously encountered, folded node. This state can be combined with one of the other states.
- The *FOLDED* state informs the observer that the current element is a folded node. The observer can now respond with the *UNFOLD* command.

This state can be combined with one of the other states.

- The *EXPLORE* state is the normal state. It is used in absence of every other state.

Now we are aware of the different traversal steering methods STALIN provides. While it is clear how we can control one traversal through the return values of the observer, this is not the case for a nested traversal. This is the reason why we present the transit element in an nested traversal twice to the observer. First it is presented with the outer and second with the inner state keyword. Now tools can respond to the outer traversal with a skip or a descent command. The descent command will tell STALIN to proceed the outer traversal into the unexplored model part adjacent to the transit element while the skip keyword prevents it. Note that there exist only those two options regardless of the category a tool belongs to. If another keyword is presented while the walker is in the outer state, STALIN will throw an `InvalidCommandException`. The second presentation in the inner state works then completely normal.

The interaction between command and state keywords now allows a differentiated control of the model traversal. We are able to repeat paths through the model, modify the model structure or switch between the input language and the meta language. Nevertheless there are some missing components for tools which require interactive access to a model. Some of our output tools like Graphview2D and PrefuseVisualization use selections on the displayed model to show extended informations, namely the payload of an model element. Because they cannot access the model structure as they want we had to supply methods to search in the model for certain elements. This is discussed in the next section.

2.5.2 Searching for Model Elements

Tools with the need for interactive access to the model structure like editors or visualizations can use the different search methods statically provided by the model manager. The search methods return the payload of a certain model element or initiate a traversal starting from an arbitrary entry point of the model. Because those access methods are strictly for interactive use there are certain restrictions in effect. First of all, a tool can only access those methods when there is no toolchain execution in progress. Second, a traversal of a model is always a blocking operation, that means that no other operation can be executed until this traversal is complete. Finally, the restrictions implied through the tool categories sustain.

Because there is no possibility that a tool can execute an operation outside a toolchain those methods can only be accessed through user interaction. This means that a tool has to add controls to the RCP UI extension points which then can in return access the search methods of the model manager. The model manager provides the following access methods:

- `List<GraphType> getModels()`

  The method returns a list of `GraphType` objects which represent the currently available models. The list contains only the models which comply to the tool's model selection. If there is no such model, an empty list is returned.

- `IPayload search(GraphType model, StalinUID uid)`

  The method returns the payload of the element of the specified model that has the specified UID. If there is no such element, `null` is returned.

- `List<IPayload> search(GraphType model,`

  `String outerKey, String innerKey, Object value)`

  The method searches for elements whose annotations match the given parameters. The parameters are:

  - `outerKey` specifies the outer key of the two dimensional hash map.
  - `innerKey` specifies the inner key of the two dimensional hash map.
  - `value` specifies the value of the bucket which corresponds to the keys.

  If the method does not match any elements it will return an empty list. There are two more methods that work similar like this one. The first removes the `value` parameter and matches every element who has any entry in the bucket. The second removes additionally the `innerKey` parameter and thus matches every element which has an entry in the `outerKey` bucket.

- `void traverse(GraphType model, StalinUID uid,`

  `List<IObserver> observer)`

  This method starts a model traversal using the specified model and the specified element as entry point. Only the initializing of the traversal is different, everything else works exactly as in the normal traversal.

2.5.3 Creating and Modifying Models

We have seen so far how STALIN traverses models. This section describes how tools belonging to the analysis and generator categories can create or modify models. We use the abstract factory pattern in conjunction with the graph description object `GraphType`. The typing of the factories is completely transparent

to the tools. Every factory supports the same methods; they only differ in the graph structure they create.

In figure 2.4 we saw that every tool that belongs to the modifying categories implements the method `setFactory(...)`. This method is used by the core to supply an appropriate factory to the respective tool. The core determines based on the obtained graph description which type this factory has and thus which kind of graph structure it produces.

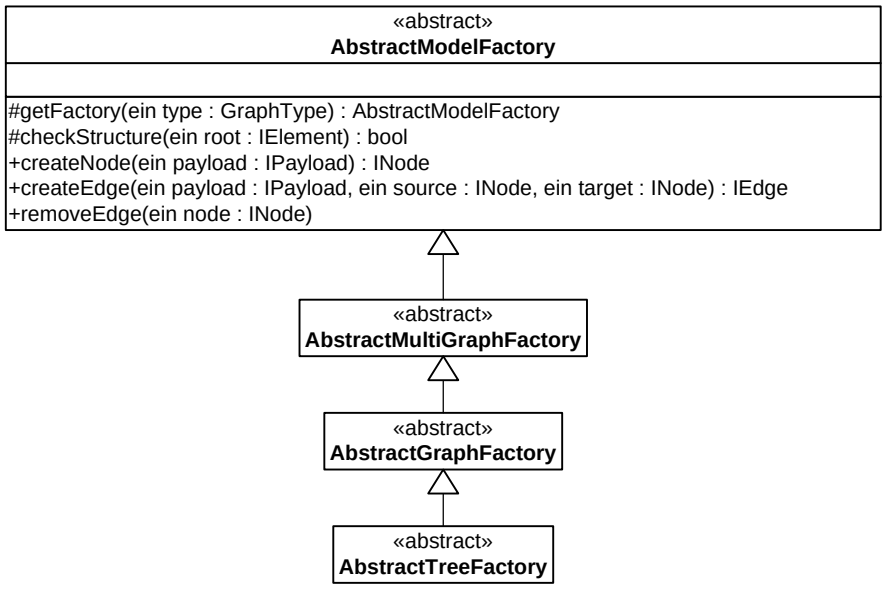

Figure 2.6: The figure shows STALIN's factories as UML class diagram.

Figure 2.6 shows what factories are available in STALIN. The abstract class on top of the hierarchy is named `AbstractModelFactory`. This class defines the static and protected method `getFactory(...)` which takes a `GraphType` object as parameter and returns an object of its own type. This object is already the real factory, as the `getFactory(...)` method of the top factory decides which specialized factory should be called. All other classes contain anonymous inner classes for their factory type. They also implement all abstract methods of `AbstractModelFactory`. Those methods are then available to the modifying

tool. Before we explain the workings of those methods we want to discuss the different factory types.

As you can see we only provide three different factories. For each general graph type there is an own factory. We do not separate between directed and undirected graphs as they do not differ in the used graph structure but only in the interpretation of the sources and targets of the edge implementations. Because we keep the information about the graph structure separate from the carrier structure we do not need to encode any difference between those graph types here. Theoretically we could have used only one factory, but it showed that the implicit separation of those graph types and their creation lead to substantial performance increases.

The tools, however, are not aware of the different factories as they receive only one type. This type provides the following public methods to them:

- `INode createNode(IPayload payload)`

  This method creates a node with an attached payload. It returns an object of type `INode` and takes an `IPayload` instance as parameter. If a tool does supply null as payload, the method will return an element without initialized payload.

- `IEdge createEdge(IPayload payload, INode source, INode target)`

  This method creates an edge between two nodes. It takes a source, a target and a payload object as parameters. Again, if null is supplied as payload the method will return an edge without initialized payload. However, if one of the source or target arguments is missing, it will throw a `NullPointerException`. There is no implicit meaning in the names of the

parameter source and target if the current graph structure is an undirected graph. In this case the edge makes the two supplied nodes simply adjacent.

- `void removeEdge(IEdge edge)`

  This method removes an edge from the graph. Parts of the graph that become unreachable after the operation are discarded. If the supplied argument is null this method will throw a `NullPointerException`.

We choose to separate the factories from the graph definition to keep the informations about the type of a data structure separate from its real type. We did not want to encode this information implicit into the typing of a generated data structure to gain more freedom for a performance sensitive implementation. Our graph structure does not enforce constraints on itself as the normal approach would be. The structural invariants are completely enforced by the factories and the core. Every factory provides the static and protected `checkStructure(...)` method which validates a given data structure from its entry point. It is applied on the output of every `getModel()` method of a tool and every `getElement()` method of an observer. If a constraint violation is detected, the method throws an `ConstraintViolationException` which is in turn thrown by the core.
This behavior allows our analysis tools to change the graph structure of a model without the need to explicitly change the type of all elements in this structure and therefore yields a great performance benefit.

2.6 User Interface

The STALIN user interface is driven by RCP. As such, it needs only little intervention from our side. It is already extensible through the RCP API and the Standard Widget Toolkit (SWT) [49]. Because the main STALIN application needs some kind of GUI hook, we created the interface `IControllerPlugin` as seen in figure 2.4. The core instantiates an `IControllerPlugin` object as UI controller. Because there is no sense in multiple UI controllers, only one object at a time may be present.

`IControllerPlugin` provides the `init(...)` method which takes a reference to the current core as argument. It also provides different callback methods for the core. This kind of mutual access is needed because the core must obtain informations from the user, like explicit model, parser or tool selections, while the UI controller needs to specify the core as main thread to the RCP workbench advisor. Due to thread safety issues both of them have to be separate instances. We also provide the interface `ILogDisplayPlugin` which is responsible for encapsulating different log display facilities. STALIN's log output is distributed to all attached instances of `ILogDisplayPlugin`, which enables us to provide visual logging as well as file logging or remote logging monitors. Currently, we write log output only to a view in the standard STALIN GUI.

There is no need for a developer to implement any of the here briefly described interfaces, as their implementations are all provided by STALIN itself.

2.6.1 Logging

Many tools need to output logging messages to inform developers and users alike about the current status of the tool. STALIN makes no exception and provides logging facilities for itself and for tool developers. We use the log4j [50] library provided by the Apache Software Foundation. To benefit from the log4j methods one has only to...

- include the import of the logging facilities with
  `import org.apache.log4j.Logger;`
- get the root logger instance of log4j with
  `Logger s_Logger=Logger.getRootLogger();`
- and use the logger with, for instance,
  `s_Logger.info(message);`.

The logger support various log levels accessible through the methods fatal, error, warn, info, debug and trace. They represent different logging levels ranging from error-logging to verbose output. STALIN allows you to set the log level in its preferences. We support the same levels as the log4j library. As the general usage of the library is fairly straight forward, we do not explain it here. Please refer to the aforementioned log4j website for any questions.

2.7 Parser

Parser are an important aspect of every program analysis tool and STALIN does not make any difference here. Parser generate usually a parse tree or an abstract syntax tree. But because most of the time a developer would choose to use a parser generator, i.e. ANTLR [30], over hand-coding an own parser, the output format of the parser has to be transformed to the graph format of STALIN. This is not as complicated as it sounds; most parser generators are aware of the output format issue and provide methods to adapt their code easily to another format.

The `ISource` interface defines the different attachable parser. It contains the following methods:

- `List<String> getFileTypes()`

  Defines which file endings STALIN should show in his input file dialog.

- `GraphType getOutputDefinition()`

  Is called when STALIN loads the parser to determine which kind of model the parser will provide.

- `TokenMap getTokenMap()`

  Returns a `TokenMap` object, which contains the paired mapping of the parsed language tokens to the meta language tokens (see section 2.2 for details about the meta language).

- `boolean parseable(File file)`

  STALIN calls this method for single file selections to determine if the parser can handle this specific file. If multiple parsers return true for this call,

STALIN will ask the user which parser it should use.

- `boolean parseable(List<File> files)`

  STALIN calls this method if the user has selected multiple files from the input file dialog. The parser should return a list containing all files he can handle. STALIN also expects the parser to merge the returned files into one graph structure. If the parser does not support any of the presented files or cannot merge them properly it should return an empty list or null. If multiple parsers can handle the whole file selection, STALIN will ask the user which parser should be used. If no parser can handle the whole file selection, STALIN will ask the user if it should proceed and if so, iterate the parsers file by file and generate single models for each file.

- `INode parse(File file)`

  This method is the main parsing method. It receives a single file and should return a graph structure corresponding to the definition supplied by `getOutputDefinition()`. It is only called with a previously presented file that was considered parseable by the parser.

- `INode parse(List<File> files)`

  This method will be called if multiple files were selected and the parser signaled support for all of those. It receives the list of files previously presented to the parser. STALIN expects the parser to generate one graph structure corresponding to the definition supplied by `getOutputDefinition()`.

Note that every parse function is allowed to throw exceptions. If a parser throws an exception while parsing, STALIN aborts the current toolchain execution and

displays the exception.

2.7.1 Integrating an ANTLR-based Parser

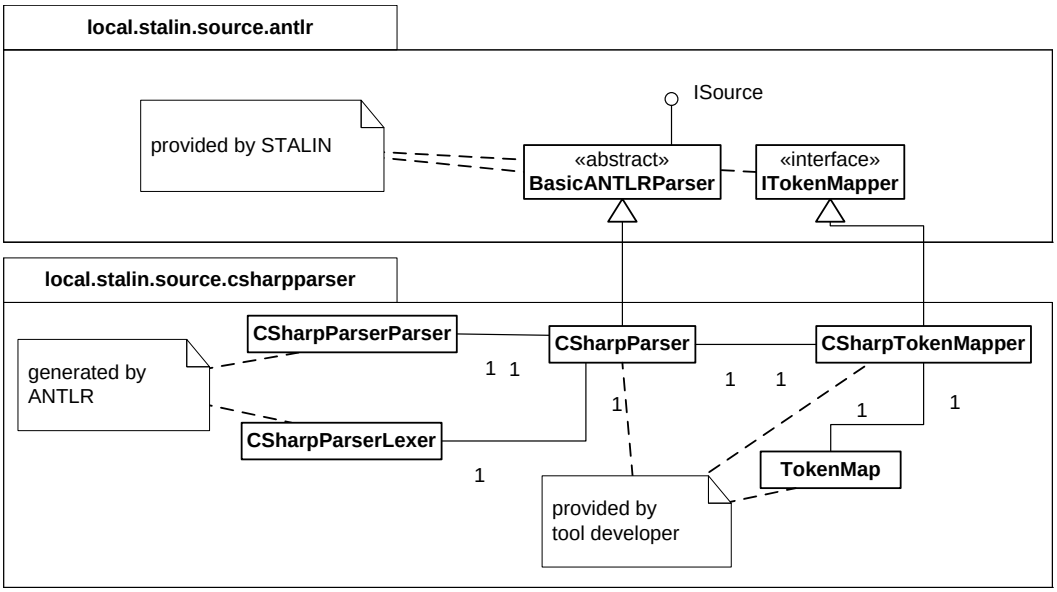

Figure 2.7: The figure shows the integration of our C# parser with the provided `local.stalin.source.antlr` package as UML class diagram.

We provide an already adapted package for the parser generator ANTLR named `local.stalin.source.antlr`. It includes the ANTLR library, the abstract class `BasicANTLRParser`, the interface `ITokenMapper` and other, for the correct output necessary, classes. Figure 2.7 shows the integration of our C# parser, which is based on ANTLR, into the framework. As you can see, one has only to define the classes `CSharpParser` and `CSharpTokenMapper`. The first class supplies only references to the necessary classes. STALIN uses then reflections to call the actual methods when the parser should be used. Therefore, when using our ANTLR package, one has only to implement the following override methods of

`BasicANTLRParser`:

- `String getPluginID()`

  The same as in `IRCPPlugin` (see section 2.4 for details).

- `String getName()`

  The same as in `IRCPPlugin` (see section 2.4 for details).

- `String getEntryPoint()`

  Has to return the name of the starting production of the ANTLR grammar. In our example this production is named "'compilationUnit"'. Every ANTLR grammar has only one starting production.

- `Class getLexer()`

  Has to return the class type of the attached lexer which will be invoked through reflections. In our example this class type is represented by `CSharpParserLexer.class`.

- `Class getParser()`

  Has to return the class type of the attached parser which will be invoked through reflections. In our example this class type is represented by `CSharpParserParser.class`.

- `Class getTokenMapper()`

  Has to return the class type of the attached token mapper which will be invoked through reflections. In our example this class type is represented by `CSharpTokenMapper.class`.

The second class is the mentioned token mapper. It is an implementation of the `ITokenMapper` interface. This interface carries only two methods, namely `getTokenMap()` and `getTokenSet()`. `getTokenMap()` returns a token map which maps the tokens from the input language to tokens from our universal syntax as we have described in section 2.2. `getTokenSet()` only returns a list of all tokens used by this parser.

2.7.2 Existing Parsers

STALIN does currently feature the following parser:

- Two parser for Java 6. The first is based on the described ANTLR package and in the final stage. The second uses the javac compiler directly and is still in beta stage.
- Two parser for C# with .NET 2.0. Again, the first is based on the ANTLR package and in the final stage. The second uses the monoc compiler direclty and is still in beta stage.
- A parser which uses tools of the GNU Compiler Collection (GCC) to directly access different intermediate representations of the GCC. It supports every language the GCC supports. It is still in beta stage.
- A XML parser based on Xerces [51] in final stage.
- Different ANTLR-based parsers for special inputs provided by different tools from members of the Chair of Software Engineering of the University Freiburg.

Every parser in this listing is a complex program for itself and it would go beyond the scope of this work to describe them any further. Justus Bisser's bachelor thesis is devoted to this part of STALIN and should appear shortly after this work.

Chapter 3

Related Work

There exists various direct and indirect approaches to unify the world of program analysis in frameworks. We have already mentioned CIL [13], which comes with a set of tools for parsing, merging and transforming ANSI C, MS C and GNU C programs. It also contains mechanisms and tools to analyze the control and data flow of C programs. Unfortunately its transformations of the input language are not configurable. This makes it impossible to look at the source exactly as it appears. It is also, as the name suggests, limited to C programs which we are not.

The theoretical approach of Michael Huth in [14, 15] was also mentioned. It is mainly concerned with finding an appropriate mediation between different levels of abstraction and different logics which describe those. This is important to our problem but does not convey it as a whole.

From the domain of model checking comes Bogor [52, 53, 4], a fully fledged and extensible framework for software model checking. Bogor is also based on Eclipse RCP and uses its extensible nature to support numerous model checking algo-

rithms. It uses a revised version of the Bandera Intermediate Representation (BIR) as input language, which is originally part of the Bandera model checker for Java [54]. Although this language can express most constructs of modern programming languages, it still does not free a developer from the concern for language translation. One has still to translate a standard program or program artifact in BIR and back to create a tool which can handle industrial software.

The perhaps most appealing approach to our problem comes from Microsoft Research with the Phoenix Framework [55]. Phoenix is an extensible and fast-growing framework which can handle all aspects of program analysis for programming languages supported by Microsoft. It is designed as back end for future MS compilers and programming tools. It supports already a wide range of abstractions and analyses as well as infrastructure for external tools. Its only drawback is the lack of language independence and the currently closed sources.

Chapter 4

Conclusion and Future Work

4.1 Future Work

While we outlined the implementation of our framework it has become clear that there were many challenges; some of them we could not address in this work. This leaves much space for future work. The current road map focuses on the following aspects of STALIN:

Justus Bisser will provide solutions for the meta language issue in his bachelor thesis. He will decide if we continue our work on an own intermediate language or use one of the already existing like BoogiePL or the ESC/Java intermediate language. He will also push the development of the ProgramNormalizer further, as we expect to gain more insight into he rough edges of the language issue through trying to transform certain expressions into others. This should eventually lead to the identification of certain problem classes in modern programming languages. Such problem classes could serve similar to SMT problem divisions [56, 57, 58] as

comparator between the efficiency of different tools.

At the same time we have to push the development of the model generating tools described in section 2.4.1 beyond their beta stages. Solid model generators will enable us to focus on implementing already known program analysis and model checking algorithms into STALIN. Then we can decide where our architecture lacks usability or feasibility and adapt those areas. The ATCML development will serve us as an example in this task.

After finishing this phase we have to care for the missing features in our architecture. We identified the following issues which will be addressed as soon as our current road map allows us in the order of the list:

- We have to find a feasible method to integrate existing SMT and SAT solvers to be accessed directly through STALIN. To achieve this we have not only to provide interfaces for tools and solvers but also handle the representation of logic formulas in different theories.

- We need to find a solution for lazy calling between tools; that means a tool should be able to mark elements which in turn should be processed by another tool. Afterward STALIN should resume the original tool to continue his work.

- We need to integrate the support for annotation languages like the Java Modeling Language (JML) [59] and Spec# [28]. Annotations from this language should be integrated directly into the annotations provided by STALIN.

- We have to integrate feasible benchmarks to measure the performance of every part of the framework, be it tools, parsers, solvers, the memory manager

or the carrier structure. The benchmarks should be completely transparent and configurable. They should measure runtime as well as memory consumption.

4.2 Conclusion

This thesis is dedicated to simplify the increasingly complex process of implementing program analysis and model checking algorithms. We recognized that the greater part of the work currently performed by experts in the field is not about finding algorithms but about combining and implementing existing ones. Program analysis and model checking have evolved beyond the stages of fundamental research to concrete applications in industrial scale environments. Therefore scalability gains greater importance, and with it the combination of different approaches. This demands in return the usage and knowledge of many different branches of this huge research area. This cannot be accomplished without standardization of program abstractions and language transformations. We presented a framework which aids both tool developers and researchers alike with tackling this tasks. STALIN provides a feasible infrastructure while still keeping things simple for developers. Although we could not meet all of our high objectives we delivered a conclusive architecture and sketched the road map for future developments without constraining the original idea.

Bibliography

[1] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[2] Thomas Ball and Sriram K. Rajamani. The SLAM Toolkit. In *CAV '01: Proceedings of the 13th International Conference on Computer Aided Verification*, pages 260–264, London, UK, 2001. Springer-Verlag.

[3] Tom Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software Verification with BLAST. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, pages 235–239. Lecture Notes in Computer Science 2648, Springer-Verlag, January 2003.

[4] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 267–276, New York, NY, USA, 2003. ACM.

[5] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A New Symbolic Model Checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[6] Matthew B. Dwyer, John Hatcliff, Robby Robby, Corina S. Pasareanu, and Willem Visser. Formal Software Analysis Emerging Trends in Software Model Checking. In *FOSE '07: 2007 Future of Software Engineering*, pages 120–136, Washington, DC, USA, 2007. IEEE Computer Society.

[7] Andreas Podelski, Andrey Rybalchenko, and Thomas Wies. Heap Assumptions on Demand. In *Computer Aided Verification, CAV'08*, 2008. to appear.

[8] Paul B. Jackson, Bill J. Ellis, and Kathleen Sharp. Using smt solvers to verify high-integrity programs. In *AFM '07: Proceedings of the second workshop on Automated formal methods*, pages 60–68, New York, NY, USA, 2007. ACM.

[9] I. Gent and T. Walsh. The search for satisfaction, 1999.

[10] J. Gu, P. Purdom, J. Franco, and B. Wah. Algorithms for the satisfiability (sat) problem: a survey, 1996.

[11] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, 2006.

[12] Roberto Sebastiani. Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation, JSAT.*, 3:141–224, 2007.

[13] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

[14] Michael Huth. A Unifying Framework for Model Checking Labeled Kripke Structures, Modal Transition Systems and Interval Transition Systems. In *FSTTCS*, pages 369–380, 1999.

[15] Michael Huth. Domains of View: A Foundation for Specification and Analysis. Technical report, Kansas State University, October 2000.

[16] Rich client platform (RCP) applications:. http://www.eclipse.org/community/rcp.php.

[17] The official website for the Eclipse Integrated Development Environment:. http://www.eclipse.org/.

[18] Jeff McAffer and Jean-Michel Lemieux. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications*. Addison-Wesley Professional, 2005.

[19] The official website for the Eclipse Rich Client Platform:. http://wiki.eclipse.org/Rich_Client_Platform.

[20] Steve Burbeck. Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC). http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[22] ECMA C# and Common Language Infrastructure Standards, Common Language Infrastructure (CLI), Partition III, CIL Instruction Set. http://msdn2.microsoft.com/en-us/netframework/aa569283.aspx.

[23] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.

[24] GNU Compiler Collection Internals, Chapter 12: RTL Representation:. http://gcc.gnu.org/onlinedocs/gccint/RTL.html.

[25] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical report, Microsoft Research, 2005.

[26] James B. Saxe K. Rustan M. Leino and Raymie Stata. Checking Java programs via guarded commands. Technical Report 1999-002, COMPAQ Systems Research Center, Palo Alto, CA, 1999.

[27] M. Barnett, R. DeLine, M. Fahndrich, K. Rustan, M. Leino, and W. Schulte. Verification of object-oriented programs with invariants, 2003.

[28] Spec# at Microsoft Research:. http://research.microsoft.com/SpecSharp/.

[29] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[30] The offical website for ANTLR, ANother Tool for Language Recognition:. http://www.antlr.org/.

[31] The Compiler Generator Coco/R:. http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/.

[32] GOLD Parsing System:. http://devincook.com/goldparser/.

[33] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination, 2005.

[34] B. Cook, A. Podelski, and A. Rybalchenko. Termination Proofs for Systems Code, 2006.

[35] B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond safety, 2006.

[36] TERMINATOR at Microsoft Research:. http://research.microsoft.com/TERMINATOR/default.htm.

[37] The Plug-in Development Environment (PDE):. http://www.eclipse.org/pde/.

[38] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy Abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.

[39] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Lazy Shape Analysis. In T. Ball and R.B. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006, Seattle, WA, August 16-20)*, LNCS 4144, pages 532–546. Springer-Verlag, Berlin, 2006.

[40] Linda Mary Wills. *Automated Program Recognition by Graph Parsing*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1992.

[41] Evren Ermis. Graphendarstellung von Quelltext als Ansatz zur Erkennung von stereotypischen Programmteilen. Master's thesis, Universität des Saarlandes, September 2007.

[42] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[43] Mark Weiser. Program Slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[44] Grappa - A Java Graph Package:. http://www.research.att.com/~john/Grappa/.

[45] Graphviz - Graph Visualization Software:. http://www.graphviz.org/.

[46] Prefuse - an interactive information visualization toolkit:. http://prefuse.org/.

[47] IKVM.NET - an implementation of Java for Mono and the Microsoft .NET Framework:. http://www.ikvm.net/.

[48] Osgi Alliance. *Osgi Service Platform, Release 3*. IOS Press, Inc., 2003.

[49] SWT: The Standard Widget Toolkit. http://www.eclipse.org/swt/.

[50] Apache Software Foundation - log4j:. http://logging.apache.org/log4j/1.2/.

[51] Xerces, a part of the Apache XML project:. http://xerces.apache.org/.

[52] Bogor - A Software Model Checking Framework:. http://bogor.projects.cis.ksu.edu/.

[53] Matthew B. Dwyer, John Hatcliff Robby, and Matthew Hoosier. Supporting model checking education using BOGOR/Eclipse. In *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 88–92, New York, NY, USA, 2004. ACM.

[54] Bandera tool set for model checking:. http://bandera.projects.cis.ksu.edu/.

[55] Phoenix at Microsoft Research:. http://research.microsoft.com/Phoenix/.

[56] C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability Modulo Theories Competition. In K. Etessami and S. Rajamani, editors, *17th International Conference on Computer Aided Verification*, pages 20–23. Springer, 2005.

[57] C. Barrett, L. de Moura, and A. Stump. Design and Results of the 1st Satisfiability Modulo Theories Competition (SMT-COMP 2005). *Journal of Automated Reasoning*, 35(4):373–390, 2005.

[58] Clark Barrett, Leonardo Moura, and Aaron Stump. Design and results of the 2nd annual satisfiability modulo theories competition (SMT-COMP 2006). *Form. Methods Syst. Des.*, 31(3):221–239, 2007.

[59] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An overview of JML tools and applications, 2003.

Appendix

Appendix A

Meta Language

The description of the token set for the meta language was extracted from STALIN's javadoc documentation. It was created by Justus Bisser as he is the main researcher for the meta language aspects of STALIN. The list is sorted alphabetically.

A.1 Token Set

- **ADDASSIGN**: This token is an assignment of the sum the old value of its first child and the expression being the second child.
- **ANDASSIGN**: This token is an assignment of the bitwise AND operation the old value of its first child and the expression being the second child.
- **ARGUMENTS**: This token indicates that its child tokens are either passed as arguments to a function or constructor call.

- **ARITHDIV**: This token indicates that its child tokens are divided arithmetically. The first child being the dividend the second child being the divisor.
- **ARITHMINUS**: This token indicates the the second child token is subtracted from the first one.
- **ARITHMOD**: This token indicates a modular division of the two child tokens.
- **ARITHMUL**: This token indicates that its child tokens are multiplied arithmetically. Each of the two child tokens is a factor.
- **ARITHNEGATIVE**: This token indicates that the child token is negative.
- **ARITHPLUS**: This token indicates that its child tokens are added. The two children being the summands.
- **ARITHPOSITIVE**: This token indicates that the child token is positive. However there could be constructs like `(+(-(5))`.
- **ARRAY**: This token indicates the creation of an array. Its first child being the type of the array and its second child being an expression usually the size of the array.
- **ARRAY_ACCESS**: This token is an array access. Its first child being the name of the array and its second child being the index of the accessed field.
- **ARRAY_TYPE**: This token indicates that the variable is an array. The variable is the first child of this token.

- **ARRAY_VARIABLE**: This token indicates that the variable is an array variable.
- **ASSIGN**: This token indicates an assignment of the second token called ASSIGNMENT_EXPRESSION being the expression assigned to the first child named LEFT_HAND_SIDE.
- **BITAND**: This token indicates a bitwise AND operation of its two children.
- **BITNEG**: This token indicates a bitwise NOT operation on its child.
- **BITOR**: This token indicates a bitwise OR operation of its two children.
- **BITSHIFTLEFT**: This token indicates a bitwise left shift operator.
- **BITSHIFTRIGHT**: This token indicates a bitwise right shift operator.
- **BITSHIFTRIGHTUNSIGNED**: This token indicates an unsigned right shift operator.
- **BITXOR**: This token indicates a bitwise XOR operation of its two children.
- **BLOCK**: This token indicates a block.
- **BOOLEAN_LITERAL**: This token indicates that its child is a boolean value.
- **BREAK**: This token indicates a break of e.g. a loop.
- **CAST**: This token indicates a cast to another type.
- **CATCH**: This token indicates the catch block in a try-catch block.

- **CHAR_LITERAL**: This token indicates that its child is a char literal.
- **CLASS**: This token indicates that its child is a class.
- **CLASS_BODY**: This token indicates the class body.
- **CLASS_LITERAL**: This token indicates that its child is a class literal.
- **CLASS_TYPE**: This token indicates that its child references a class.
- **COMPAQ**: This token indicates an equivalence check of its two children.
- **COMPGEQ**: This token indicates a check if the first child is greater or equal to its second child.
- **COMPGT**: This token indicates a check if the first child is greater than its second child.
- **COMPLEQ**: This token indicates a check if the first child is less or equal to its second child.
- **COMPLT**: This token indicates a check if the first child is less than its second child.
- **COMPNEQ**: This token indicates a not equivalence check of its two children.
- **COMPOSED_EXPRESSION**: This token indicates a composed expression.
- **CONDITIONAL_EXPRESSION**: This token indicates a conditional expression, usually in combination with an IF token.
- **CONSTRUCTOR**: This token indicates that its child is a constructor.

- **CONTINUE**: This token indicates that the rest of a loop should be skipped.
- **DECIMAL_LITERAL**: This token indicates a that its child is a decimal literal.
- **DECREMENT**: This token indicates that its child is decreased by one.
- **DIM_EXPR**: This token is indicates the the known or unknown size of an array. In the latter case it does not have a child.
- **DIVASSIGN**: This token is an assignment of the division the old value of its first child and the expression being the second child.
- **DOWHILE**: This token indicates a DO-WHILE loop.
- **ELSE**: This token indicates the else block of an IF statement.
- **EXPRESSION**: This token indicates that its child is an expression.
- **EXPRESSIONS**: This child indicates that its children are expressions. This is used for easier handling.
- **EXTENDS**: This token indicates that a class inherits from another class.
- **FIELD**: This token indicates a field in a class.
- **FINALLY**: This token indicates the finally block in a try-catch-block
- **FLOAT_LITERAL**: This token indicates that its child is a float literal.
- **FOR**: This token indicates a for-loop.
- **FOR_INIT**: This token indicates the initialization of a for-loop.

- **FOR_UPDATE**: This token indicates an update which is executed after each pass of a for-loop.
- **FOREACH**: This token indicates a `foreach` loop, usually used for iterating over a list or an array.
- **GUARD**: This token identifies the guard in a conditional expression.
- **HEX_LITERAL**: This token indicates that its child is a hexadecimal literal.
- **IDENT**: This token identifies an identifier.
- **IDENTIFIER**: This token indicates an identifier.
- **IF**: This token indicates an if-statement.
- **IMPLEMENTS**: This token indicates that a class implements one or more interfaces.
- **INCREMENT**: This token indicates that its child is incremented by one.
- **INITIALIZER**: This token indicates a variable or array initializer.
- **INSTANCEOF**: This token indicates a test if an object is an instance of a specific type or its type extends this type.
- **INT_LITERAL**: This token indicates that its child is an integer literal. It does not specify the length of the integer type. This should be done by some mapping.
- **LABEL**: This token indicates a jump label.

- **LEFTSHIFT_ASSIGN**: This token indicates a bitwise left shift assign operator.
- **LITERAL**: This token indicates that its child is a literal. Usually its child also specifies what kind of literal it is.
- **LOGICAND**: This token indicates a logical AND of its two children.
- **LOGICNEG**: This token indicates a logical negation of its child.
- **LOGICOR**: This token indicates a logical OR operation of its children.
- **MEMBER**: This token indicates a class member but is currently not used.
- **METHOD**: This token indicates a method. It children define its signature and body.
- **METHOD_BODY**: This token identifies the body of a method.
- **METHOD_CALL**: This token indicates a method call its children difene the name of the called method and the arguments.
- **MODASSIGN**: This token is an assignment of the modular division the old value of its first child and the expression being the second child.
- **MODIFIER**: This token indicates modifiers of a class a function or else.
- **MULASSIGN**: This token is an assignment of the multiplication the old value of its first child and the expression being the second child.
- **NULL**: This token stands for the nil value.
- **ORASSIGN**: This token is an assignment of the bitwise OR operation the old value of its first child and the expression being the second child.

- **PACKAGE**: This token indicates a package a class is in. Each child is a part of the complete package name.
- **PARAMETER**: This token identifies the parameters given to a function or a constructor call.
- **PARAMETERS**: This token's children identify the parameter signature of a method or constructor.
- **POSTFIX_OPERATOR**: This token indicates a unary postfix operator.
- **PREFIX_OPERATOR**: This token indicates a unary prefix operator.
- **PRIMITIVE_TYPE**: This token indicates a primitive type versus a class type.
- **PROJECT**: This token is the root node of a project. Its children aure of the type UNIT.
- **QUALIFIED_NAME**: This token indicates a fully qualified name of a method, a field or a class.
- **RETURN**: This token indicates a return statement. It can have a child specifying the return value.
- **RIGHTRIGHTSHIFT_ASSIGN**: This token indicates a bitwise unsigned right shift assign operator.
- **RIGHTSHIFT_ASSIGN**: This token indicates a bitwise right shift assign operator.
- **STATEMENT**: This token indicates a statement which is specified by its child.

- **STRING_LITERAL**: This token indicates that a literal is a string literal.
- **SUBASSIGN**: This token is an assignment of the subtraction the old value of its first child and the expression being the second child.
- **THEN**: This token indicates the statement which is executed when an if-check succeeds.
- **THROW**: This token indicates an explicit exception definition of a function.
- **TRY**: This token indicates the start of a try-catch block.
- **UNDEFINED**: This token is used for mapping of language constructs undefined in the Stalin metalanguage.
- **UNIT**: This token identifies a single compilation unit usually a single file.
- **VARIABLE_DECLARATION**: This token indicates a variable declaration.
- **VARIABLE_INITIALIZER**: This token indicates a variable initializer.
- **VOID**: This token indicates the void, usually as return type of a function.
- **WHILE**: This token indicates a while-true-loop.
- **XORASSIGN**: This token is an assignment of the bitwise XOR operation the old value of its first child and the expression being the second child.

Appendix B

Legal Statement

I hereby declare that I have written the Bachelor's Thesis on my own, and used no other than the stated sources and aids. I have duly acknowledged all words, phrases, or passages taken from other publications. Furthermore, I declare that neither this thesis nor a similar version of it has been submitted to any other institution for a degree or for publication.

Freiburg, March 31, 2008 Daniel Dietsch

Appendix C

German Summary

Diese Arbeit beschreibt eine Möglichkeit, den immer schwieriger werdenden Prozess der Implementierung von Methoden der Programmanalyse und des Model Checking zu vereinfachen. Wir erkennen das aktuell der größte Teil der von Experten aus diesem Gebiet geleisteten Forschungsarbeit nicht der Findung von neuen Algorithmen sondern viel mehr der Kombination von Existierenden und deren Implementierung dient. Dieser Trend zur Kombination ist auch aus den gestiegenen Anforderungen gegenüber der Skalierbarkeit einzelner Ansätze erwachsen. Das gesamte Gebiet hat sich in den letzen Jahren immer mehr aus der Grundlagenforschung gelöst und sich konkreten Aufgaben im industriellen Maßstab zugewandt. Dies kann nur mit großem Aufwand ohne die Standardisierung von Programmabstraktion, Logik und Sprachtransformationen geschehen.

Kurz gesagt, wir benötigen ein Framework das es erlaubt, uns auf die wirkliche Arbeit zu konzentrieren und das Wissen verschiedener Experten auf verschiedenen Gebieten mit wenig bis gar keinem zusätzlichen Aufwand zu vereini-

gen. Wir stellen in dieser Arbeit das STALIN Framework vor um dieses Problem zu bewältigen. STALIN stellt eine geeignete und einfach zu bedienende Infrastruktur dar, die durch ihre offene und erweiterbare Architektur auch zukünftigen Anforderungen gewachsen ist.
Wir besprechen in dieser Arbeit vor allem wie STALIN beschaffen ist, welche Hindernisse es auf dem Weg der Entwicklung gab und welche Werkzeuge wie auch welche Teile der Infrastruktur bereits fertiggestellt worden sind und welche uns noch als Aufgaben für die Zukunft bleiben.
Wir orientieren uns dabei an einer von uns festgelegten Liste von Eigenschaften die ein solches Framework unserer Überzeugung nach in sich vereinigen muss, um den anstehenden Anforderungen gerecht zu werden:

- Wir benötigen eine universelle Sprache die verschiedene moderne Programmiersprachen in einer geeigneten Weise beschreiben kann.
- Wir benötigen Parser die zumindest die wichtigen modernen Programmiersprachen in diese Sprache übersetzen können.
- Wir benötigen komplett spezifizierte und formalisierte Modelle sowie geeignete Werkzeuge um Abstraktionen unserer universellen Sprache zu erzeugen.
- Wir brauchen verschiedene Methoden um diese Modelle effizient zu speichern und zu verwalten.
- Wir benötigen einfache und effiziente Zugangsmethoden zu diesen Modellen.
- Wir benötigen standardisierte Schnittstellen für die Interaktion zwischen verschiedenen Tools.

- Wir benötigen Übergänge zwischen den verschiedenen Theorien der Theorembeweiser und SMT Solver.

Verschiedene Arbeiten haben sich bereits der Herausforderung gestellt und dabei teilweise ähnliche Frameworks geschaffen. Die C Intermediate Language (CIL) [13] ist ein Beispiel für solch ein Programmanalyse-Framework. CIL beinhaltet Werkzeuge für das Parsen, Verändern und Vereinigen von Programmen in ANSI C, MS C oder GNU C. CIL transformiert außerdem Eingabeprogramme in eine Untermenge von C, indem es komplexere Sprachkonstrukte reduziert und ein semantisch äquivalentes Programm erzeugt. Leider sind die vorgenommenen Transformationen nicht konfigurierbar, sodass eine Rückführung in den ursprünglichen Code nicht mehr möglich ist. Außerdem ist CIL, wie bereits der Name suggeriert, auf C als Eingabesprache begrenzt.
Ein weiterer Ansatz wird von Michael Huth in [15, 14] vorgeschlagen. Er beschreibt eine Interaktion zwischen verschiedenen Tools und wie verschiedene Abstraktionen in einander überführt werden können. Die Arbeit beschreibt ein formales Framework für Abstraktion und Verfeinerung welches eine generische Spezifikationssprache beinhaltet. Nach unserem aktuellen Kenntnisstand wurde dieses Framework jedoch nie implementiert, sodass wir seine Eignung für die Entwicklung von Algorithmen nicht geeignet abschätzen können.
Zusammen ergaben beide Arbeiten den Ausschlag für die Entwicklung von STALIN. Wir verwenden die Eclipse Rich Client Platform (Eclipse RCP) [18, 17, 19, 16] als Basis unseres Frameworks, da diese als sehr robuste und zuverlässige Platform bereits in zahlreichen anderen Projekten zum Einsatz kam und die von uns benötigten Funktionen mehr als ausreichend zur Verfügung stellt.
Wir beschreiben in dieser Arbeit was für verschiedene Schnittstellen für Parser

und Werkzeuge durch unser Framework bereitgestellt werden, wie STALIN Modelle verwaltet und wie von einer gegebene Eingabesprache zu unserer Universalsprache transformiert wird. STALIN bietet verschiedene Schnittstellen für Entwickler von Werkzeugen für Programanalyse, die die Kommunikation zwischen diesen Werkzeugen und mit STALIN selbst ermöglichen. Diese Werkzeuge, von uns Tools genannt, können nacheinander auf denselben, standardisierten Modellen einer Eingabesprache ausgeführt werden, sei es auf Kontrollflussgraphen, Datenflussgraphen oder auf abstrakten Syntaxbäumen. Wir stellen außerdem Werkzeuge und Methoden bereit, die verschiedene gebräuchliche Operationen auf diesen Modellen sowie graphische Ausgaben der Modelle ermöglichen und damit die Arbeit der Entwickler entschieden erleichtern. Neben diesen Tools stellen wir außerdem Parser für verschiedene Sprachen wie Java, C#, ANSI C und C++ bereit.