Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A Prototype Verifier for Weak Memory Reasoning

Bachelor's Thesis in Computer Science by

Christiane Goltz

March 20, 2017

Supervisors: Dr. Alexander J. Summers, Prof. Dr. Peter Müller

Contents

1. Introduction

Weak memory models are important models for real-world hardware behaviour. Unfortunately, they provide little consistency guarantees, which makes reasoning about concurrent programs running under them very hard. To tackle this challenge, program logics like relaxed separation logic [1] and fenced separation logic [2] have emerged. They employ abstract concepts such as ownership of memory locations and access permissions to handle the difficulties inherent to weak memory models. However, verifying programs based on these logics still requires a lot of manual effort, since their tool support is so far very limited.
This thesis provides a front-end to the Viper verification infrastructure to automate verification of weak memory programs. We developed a small input language that contains the essential operations to access memory present in the rules of the logics. It also includes an assertion language, very similar in notation to the assertions used in the presentation of the logic, that is used to supply user annotations in a few places. Programs in this input language are translated into the Viper intermediate verification language based on an encoding developed in the Programming Methodology Group at ETH Zurich [3]. Carbon, an existing backend verifier for Viper using verification condition generation, is used to verify the resulting Viper program.

2. Background

2.1. Relaxed Separation Logics

Relaxed Separation Logic (in short RSL) [1] is a program logic for reasoning about concurrent programs under the C11 memory model. It builds upon concurrent separation logic (CSL) [4] and introduces proof rules for the different atomic memory accesses from C11. These rules allow ownership of non-atomic locations to be transferred between threads upon specific accesses of an atomic location. This enables RSL to ensure race-freedom even in the presence of rather involved and daring concurrent access patterns.
The RSL proof rules for relaxed accesses, which have very weak guarantees on their own, allow no ownership transfer. To improve the usefulness of relaxed accesses, which are often less costly and therefore desirable, one needs to additionally use memory fences. Fenced separation logic (FSL) [2] introduces rules for fences as well as accompanying rules for relaxed accesses. There is also an extension of FSL, named FSL++ [5] that deals with compare and swap accesses in the presence of memory fences and the corresponding stronger rules for relaxed accesses.
The tool built for this thesis implements the encoding detailed in [3] which uses RSL rules for nonatomic and release/acquire accesses and FSL/FSL++ rules where fences and relaxed accesses are involved. We will therefore briefly summarize these rules in the following subsections. This is mostly meant to familiarize the reader with how these rules can be used, for a thorough reasoning about their correctness please refer to the respective papers mentioned above.
It is assumed that any memory location is either accessed solely through atomic or solely through non-atomic accesses. For atomic locations, there is also a distinction between locations that are only read using a read access and locations that are only read inside of a read-modify-write access.

2.1.1. Non-Atomic Accesses

There are no synchronisation guarantees for non-atomic accesses, therefore the logic needs to ensure that there are no races on non-atomic locations. To do this, the right to access a location is represented by an assertion of the form $hl \overset{k}{\mapsto} e$. We call this assertion hl *points to* e and regard it as a resource that can be passed around to model a changing ownership of the location in question. It expresses both the access right to the location and the fact that the location has been initialized and holds the value of e. If instead of an expression an underscore appears on the right hand side, it denotes the access right and the fact that the location has been initialized without specifying the value that is currently

held. The superscript k is the fraction of the access permission that is held. If k is 1, there is exclusive access, anything between 0 and 1 is shared access and 0 would be no access. We assume $k > 0$ from here on.
The rule for allocating a non-atomic location introduces the first RSL specific predicate, Uninit. It represents full access to the location inside and the information that it has not yet been initialized.

$$\frac{}{\vdash \{true\} l := \mathsf{alloc}_{\mathsf{NA}}() \{\mathsf{Uninit}(l)\}}$$

For a write, one needs full access, regardless of value or initialization status and afterwards the value will be stored according to what was written.

$$\frac{}{\vdash \{l \overset{1}{\mapsto} \_ \text{ or } \mathsf{Uninit}(l)\} [l]_{\mathsf{NA}} := e \{l \overset{1}{\mapsto} e\}}$$

To be able to read from the location we need some fraction of an access permission to it. We learn that the local variable we assign to contains afterwards the value that was stored in the location.

$$\frac{}{\vdash \{l \overset{k}{\mapsto} e\} x := [l]_{\mathsf{NA}} \{x = e * l \overset{k}{\mapsto} e\}}$$

Here we encounter for the first time the separating conjunction $*$ from basic separation logic. It is a conjunction that additionally expresses that the current heap can be split into two disjoint[1] heaps which each make one of the conjuncts true. Therefore, $l \overset{1}{\mapsto} e * l \overset{1}{\mapsto} e$ would be equivalent to false, even if $l \overset{1}{\mapsto} e$ was true, as l can't be part of both disjoint heaps.

2.1.2. Release/Acquire Atomics

When an atomic location is allocated, we associate with it a location invariant $\mathcal{Q}$. Conceptually, a location invariant is a function from values to assertions, but we represent it as an assertion parametrized by the special variable v. The location invariant associated with an atomic location expresses what ownership can be gained from reading a certain value from the location and correspondingly must be given up when writing it.

$$\frac{}{\vdash \{true\} l := \mathsf{alloc}_{\mathsf{AR}}(\mathcal{Q}) \{\mathsf{Rel}(l, \mathcal{Q}) * \mathsf{Acq}(l, \mathcal{Q})\}}$$

In the rule for Acq-Rel allocation, we encounter two new RSL predicates, the release predicate and the acquire predicate. The release predicate represents the right to write a value to the location (while giving up the ownership specified in the location invariant for the value written). Similarly, the acquire predicate allows reading a value from the location and gaining the location invariant for that value. An important difference is that the release predicate can be freely duplicated, while the acquire one can only be split into disjoint

[1]Disjoint in the presence of fractional permissions does not mean that a location can only appear in one conjunct. If it appears in multiple conjuncts however, the fractional permissions must add up to at most one and the values must be compatible.

parts to prevent multiple readers from unsoundly gaining overlapping ownership from the same write.

$$\mathsf{Rel}(l, \mathcal{Q}) \Leftrightarrow \mathsf{Rel}(l, \mathcal{Q}) * \mathsf{Rel}(l, \mathcal{Q})$$
$$\mathsf{Acq}(l, \mathcal{Q}_1 * \mathcal{Q}_2) \Leftrightarrow \mathsf{Acq}(l, \mathcal{Q}_1) * \mathsf{Acq}(l, \mathcal{Q}_2)$$

For a release write to a location, we consequently need its release predicate and whatever the location invariant specifies for the value we want to write. We give up the location invariant, but retain the release predicate and gain another predicate Init, which represents the information that the location is initialized.

$$\overline{\vdash \{\mathcal{Q}[e/\mathrm{v}] * \mathsf{Rel}(l, \mathcal{Q})\}[l]_{\mathsf{Rel}} := e\{\mathsf{Init}(l) * \mathsf{Rel}(l, \mathcal{Q})\}}$$

This is freely duplicable as well, as it does not by itself allow any ownership to be gained.

$$\mathsf{Init}(l) \Leftrightarrow \mathsf{Init}(l) * \mathsf{Init}(l)$$

For an acquire read, we need to know that the location has been initialized and we need to have the proper acquire predicate. We can then gain ownership from the location invariant according to the value we read. To make sure that we cannot gain the same ownership again from the same write, the acquire predicate that remains has a modified location invariant, which gives nothing for reading the value again.

$$\overline{\vdash \{\mathsf{Init}(l) * \mathsf{Acq}(l, \mathcal{Q})\}x := [l]_{\mathsf{Acq}}\{\mathcal{Q}[x/\mathrm{v}] * \mathsf{Acq}(l, \mathrm{v} \neq x \Rightarrow \mathcal{Q})\}}$$

2.1.3. Fences and Relaxed Accesses

In order to transfer ownership away through a relaxed write, there needs to be a release fence before the write and the locations whose ownership is to be transferred may not be accessed in between the fence and the write. To ensure this, FSL introduces the modality $\triangle$ (up) that protects an assertion between the fence and the write. A release fence simply puts an assertion under the modality.

$$\overline{\vdash \{A\}\mathsf{fence}_{\mathsf{Rel}}\{\triangle A\}}$$

A relaxed write works very much like a release write, except that it requires the location invariant to be under the modality to be able to give it away.

$$\overline{\vdash \{(\triangle \mathcal{Q}[e/\mathrm{v}]) * \mathsf{Rel}(l, \mathcal{Q})\}[l]_{\mathsf{Rlx}} := e\{\mathsf{Init}(l) * \mathsf{Rel}(l, \mathcal{Q})\}}$$

Similarly, a relaxed read can only gain ownership upon a following acquire fence. This is modelled by introducing a second modality $\triangledown$ (down), allowing the relaxed read to gain the location invariant under this modality.

$$\overline{\vdash \{\mathsf{Init}(l) * \mathsf{Acq}(l, \mathcal{Q})\}x := [l]_{\mathsf{Acq}}\{(\triangledown \mathcal{Q}[x/\mathrm{v}]) * \mathsf{Acq}(l, \mathrm{v} \neq x \Rightarrow \mathcal{Q})\}}$$

The modality needs to be removed to be able to use the gained information in any way, which is the task of the following acquire fence.

$$\overline{\vdash \{\triangledown A\}\mathsf{fence}_{\mathsf{Acq}}\{A\}}$$

2.1.4. Compare and Swap

To be able to use a location in CAS accesses, it needs to be allocated as a RMW location. This allocation is very similar to release-acquire allocation, only instead of the acquire predicate a RMW-acquire predicate is gained.

$$\frac{}{\vdash \{true\} l := \mathsf{alloc}_{\mathsf{RMW}}(\mathcal{Q}) \{\mathsf{Rel}(l, \mathcal{Q}) * \mathsf{RMWAcq}(l, \mathcal{Q})\}}$$

This works just like the acquire predicate, but there is one major difference. In contrast to the acquire predicate, the RMW-acquire predicate can be duplicated. This does not allow different threads to gain overlapping ownership by reading from the same write, since in a read-modify-write access the value read will atomically be overwritten again. This also means there is no need to track previously read values by changing the invariant in the predicate after a read as we did for acquire reads. If the same value is read again, there has always been some other write that wrote it back and correspondingly gave up the necessary ownership.

$$\mathsf{RMWAcq}(l, \mathcal{Q}) \Leftrightarrow \mathsf{RMWAcq}(l, \mathcal{Q}) * \mathsf{RMWAcq}(l, \mathcal{Q})$$

To attempt a compare and swap, we need to know that the location is initialized, we need to be allowed to write to it and we need to be allowed to read from it in a RMW way. A CAS access is of the form $x := \mathsf{CAS}_{rmwm}(l, e, e')$. It can either succeed, meaning we read the value e we expected from the location l and write the new value e' or fail, in which case we read some value different from e and do not write anything. The failing case is quite simple, no ownership is transferred and we retain exactly what we had before. In the successful case, conceptually we first gain a location invariant from the value read, just like for any other atomic read. After that we give up the location invariant for the value written, like in an atomic write. If the readmode rm or writemode wm is a relaxed access mode, the corresponding modality appears in the rule. Now it might be that the invariants gained from the read and given away by the write actually overlap. In the presence of modalities it would be useful if this overlapping portion (which we will call T) would not be transferred at all, since the modalities might make it impossible to transfer it away without a fence. We then only gain the remaining part of the location invariant (A) from the read and only need to give up the rest of the location invariant for the write (P). These actually transferred parts A and P are then the only ones interacting with the modalities[2].

$$\frac{\begin{array}{ll} x \notin FV(P) & \\ x \notin FV(e) & P' \equiv \begin{cases} P & \textit{if } wm \in \{Rel, SC\} \\ \triangle P & \textit{otherwise} \end{cases} \\ \mathcal{Q}[e/\mathrm{v}] \models A * T & \\ P * T \models \mathcal{Q}[e'/\mathrm{v}] & A' \equiv \begin{cases} A & \textit{if } rm \in \{Acq, SC\} \\ \triangledown A & \textit{otherwise} \end{cases} \end{array}}{\vdash \left\{ \begin{array}{c} \mathsf{Init}(l) * \mathsf{Rel}(l, \mathcal{Q}) * \\ \mathsf{RMWAcq}(l, \mathcal{Q}) * P' \end{array} \right\} x := \mathsf{CAS}_{rmwm}(l, e, e') \left\{ \begin{array}{c} (x = e \,?\, A' : P') * \mathsf{Init}(l) * \\ \mathsf{Rel}(l, \mathcal{Q}) * \mathsf{RMWAcq}(l, \mathcal{Q}) \end{array} \right\}}$$

[2]The unified rule for all access mode combinations presented here is taken from [3], which adapted the rules from [5]. The notation of the access modes is adapted to be more in line with the input language presented in chapter 3.

2.2. Viper

The verification infrastructure Viper presented in [6] was developed in the Programming Methodology Group at ETH Zurich. It includes powerful concepts for permission based reasoning that are at the heart of the encoding realized in our tool. We will present some of the most important and heavily used concepts of the intermediate language in the following.

2.2.1. Objects, Fields and Permissions

Viper does not have classes, instead fields are declared for the whole program. Every object in the program has all of these fields. References to objects can be stored in variables of the built-in type `Ref`. To control access to the program heap Viper uses *permissions*. A location may only be accessed if the necessary permission is held at the program point of the access. Permissions can be specified and transferred, for example through method pre- and postconditions, using *accessibility predicates*. The access to a field f of a reference x would be denoted by `acc`($x.f$). This simple form represents full access to the location, allowing writes to it. It is also possible to specify *fractional permissions*, like `acc`($x.f$, `1/2`), that only permit reads from the location. Instead of a concrete permission amount it is also possible to use a *wildcard* permission amount as in `acc`($x.f$, `wildcard`). This represents an arbitrary small fractional permission that can always be split into multiple wildcard permissions if necessary. We will use this splitting property in the encoding of the duplicable RSL predicates seen in section 2.1. It is also possible to refer to the permission amount held at a point in the program using `perm`($x.f$). Permission amounts can be stored in variables of the built-in type `Perm`. It is possible to use them inside accessibility predicates and even to perform comparisons and arithmetic operations on them.

2.2.2. Inhales and Exhales

Apart from method pre- and postconditions, information and permissions can be transferred at arbitrary program points using Viper's *inhale* and *exhale* statements. An inhale allows the verifier to assume all assertions contained in it and to use the permissions inside from this program point on. Conversely, an exhale obligates the verifier to assert that all assertions in it hold and the permissions inside are available at this point. They are then given up and are no longer available from this point onward.

2.2.3. Predicates

Viper allows the definition of *predicates* which can be held at a program point and transferred, similarly to permissions. They can have arguments and an optional body containing an assertion. It is also possible to specify fractional access to a predicate instance.

2.2.4. Old Expressions and Labels

In method postconditions, it is sometimes useful to refer to the old value of an expression from before the method was executed. To facilitate this, Viper has *old expressions*, written as `old`(e). This concept can be used in a more flexible way in combination with Viper's *labels*. These give a name l to a program point which can then be used in a *labelled old expression* such as `old`[l](e). The old expression then refers to the value of the expression at the point of the label.

2.2.5. Quantified Permissions

Viper supports *quantified permissions*, allowing specifications such as

```
forall r: Ref :: (r in refSet) ==> acc(r.val)
```

where `refSet` is a variable of type `Set[Ref]`. The expression above then denotes permission to the `val` fields of all references in this set.

3. Input Language

We designed the input language of the tool using the language from the examples in the original RSL paper [1] as a base. There are some differences, however, most notably the treatment of parallelism and local variables as well as the statement-based approach. For the full syntax, see Figure 3.1.

$$
\begin{aligned}
hl, x, x' &\ \text{are identifiers for a Heap Location and Local Variables resp.}\\
tk &\ \text{is an identifier used as a fork token}\\
m &\ \text{is an identifier used as a method name}\\
i, i' &\ \text{are integers}\\
op &\in \{+, *, -\},\ cop \in \{==, !=, <=, >=, <, >\}\\
rm &\in \{\text{Acq,Rlx,SC,NA}\},\ wm \in \{\text{Rel,Rlx,SC,NA}\},\ am \in \{\text{RMW,AR}\}\\
e, e' \in Aexp &::= x \mid \text{Int} \mid (e\ op\ e') \mid (b\ ?\ e\ :\ e')\\
&\mid x\ :=\ \mathtt{[}hl\mathtt{]}_{rm} \mid \mathtt{[}hl\mathtt{]}_{wm}\ :=\ e \mid x\ :=\ \mathtt{CAS}(hl, e, e')_{rmwm}\\
b, b' \in Bexp &::= \mathtt{not}\ b \mid (b\ \mathtt{or}\ b') \mid (b\ \mathtt{and}\ b') \mid (e\ cop\ e')\\
s, s' \in Stm &::= x\ :=\ e \mid \mathtt{var}\ x \mid \mathtt{skip} \mid [Stm;]^{+} \mid \mathtt{if}\ b\ \mathtt{then}\ \{s\}\ \mathtt{else}\ \{s'\}\\
&\mid \mathtt{while}\ b\ \mathtt{invariant}\ A\ \{s\} \mid \mathtt{while}\ b\ \{s\}\\
&\mid hl\ :=\ \mathtt{alloc}() \mid hl\ :=\ \mathtt{alloc}(A)_{am}\\
&\mid tk\ :=\ \mathtt{fork}(m, e, \ldots, e') \mid x\ :=\ \mathtt{join}(tk)\\
&\mid x\ :=\ \mathtt{[}hl\mathtt{]}_{rm} \mid \mathtt{[}hl\mathtt{]}_{wm}\ :=\ e \mid x\ :=\ \mathtt{CAS}(hl, e, e')_{rmwm}\\
&\mid \mathtt{FenceAcq} \mid \mathtt{FenceRel}(A)\\
Method &::= \mathtt{method}\ m(x, \ldots, x')\ \mathtt{pre}\ A\ \mathtt{post}\ A'\ \{s\}\\
Program &::= [Method]^{+}\\
a, a' \in AssertExp &::=\ {}^{*}hl \mid \text{Int} \mid \mathtt{True} \mid \mathtt{False} \mid x \mid (a\ op\ a')\\
&\mid \mathtt{not}\ a \mid (a\ \mathtt{or}\ a') \mid (a\ \mathtt{and}\ a') \mid (a\ cop\ a')\\
A, A' \in Assert &::= \mathtt{acc}(hl) \mid a \mid (A\ \&\&\ A') \mid (a\ \Rightarrow\ A) \mid (a\ ?\ A\ :\ A')\\
&\mid \mathtt{Rel}(hl, A) \mid \mathtt{Acq}(hl, A) \mid \mathtt{Acq}(hl, A\mathtt{[}i := \mathtt{emp}, \ldots, i' := \mathtt{emp}\mathtt{]})\\
&\mid \mathtt{RMWAcq}(hl, A) \mid \mathtt{Init}(hl) \mid \mathtt{Uninit}(hl) \mid \mathtt{Up}(A) \mid \mathtt{Down}(A)
\end{aligned}
$$

Figure 3.1.: Syntax of the input language

3.1. Values, Variables and Heap Locations

For simplicity, all values are integers and all local variables therefore implicitly have type integer. A local variable needs to be declared inside the method it is used in, but there is no further scoping available. It does not matter where inside the method it is declared. This is again partly for simplicity, but also due to the fact that the Viper language does not natively support scoping.
Heap locations also only store integer values and are treated differently from local variables in a number of ways. The most important difference is that a heap location is globally available if it is allocated anywhere. We do not model references as local information that needs to be passed around between methods through arguments of reference types. Any heap location that is allocated somewhere in the program is available using the corresponding identifier anywhere in the program. This makes examples look more similar to the ones from [1] (where parallelism happens seamlessly inside the code) and simplifies the implementation of methods. Access rights to the heap location on the other hand are only available locally after allocation. They need to be explicitly given to another method via its precondition. The allocation site of a heap location determines whether it is an atomic location and if it is, whether it will be accessed using acquire reads or RMW statements. Accessing a heap location in a manner that doesn't match with this information, for example doing a non-atomic write on an atomic heap location, is an error and the tool will reject the program.

3.2. Programs and Methods

A program in the input language consists of a sequence of methods. These can have a sequence of formal arguments that are implicitly of type integer and available inside the method body as a local variable.
Method names have to be unique throughout the program. A method has a pre- and a postcondition, given as an $Assert$. These are mostly used to specify access rights to heap locations using RSL style predicates that the method should have, but they can also talk about the formal arguments and the result of the method. Each method has an implicitly declared `Result` variable and the value this variable has at the end of the body is the return value of the method. If it has a meaningful value, this should be described in the postcondition, as in the actual verification only the postcondition of a method is learned after its execution. The return value is assumed to be an arbitrary integer if the postcondition doesn't give any better information about it.

3.3. Fork and Join

To model parallelism, there are fork and join statements, where the fork statement takes a method name and a number of actual arguments corresponding to the formal arguments of the method. The fork statement has to contain a unique string token, written

as tk := **fork**(...). While this looks a lot like assignment to a local variable, it is fundamentally different. The token need not be declared in any way and cannot be reused (apart from the intended use in a corresponding join statement, of course). When forking a method, access rights to heap locations required by its precondition need to be available in the enclosing method and are given away upon forking it.
The join statement x := **join**(tk) takes a token and assigns the return value of the method that was forked on the token to a local variable. The method in which the join happens gains all information and access rights available from the postcondition of the joined method. In fact, this is the only information gained, so the return value should be described by the method postcondition if it is meaningful.
Forking is the only way to execute a method from another one, as normal method calls were not needed in any example, but they could most likely be added with little additional work.

3.4. Assertion Language

The assertion language is a closed subset of the input language and used in two similar but different ways.
For one, it is used to specify method pre- and postconditions and loop invariants. Here the full set of assertions can and will be used.
The second use is inside atomic allocation and in Rel and Acq predicates. We will call this use case a location invariant, as inside these constructs, there is always a specific location - the one being allocated or the one specified in the predicate before the assertion - that the assertion is associated with. The value of this location can be talked about inside such an assertion by using the reserved identifier v. In a Rel predicate this represents the value that is being written to the location, in an Acq or RMWAcq it represents the value read. Often a location invariant will be a conditional on this value, for example ((v == 0) ? **True** : (**acc**(a) && (*a == 7))). This example says that if the value read or written is 0, no information is gained or given up. If it is any other value, a write has to give up access to the nonatomic location a and ensure that its value is 7. The conjunction of **acc**(x) and ($*x$ == e) corresponds to a points-to assertion in RSL. In fact, this example is the location invariant $\mathcal{Q}(x)$ used in Figure 8 [1], where it is written as $\mathcal{Q}(x) \overset{\text{def}}{=}$ **if** $x = 0$ **then** emp **else** $a \mapsto 7$. Splitting the access right and information about the value into two conjuncts is closer to the way this information is represented in the Viper language after the translation and at the same time allows for a little more flexibility in specifying the possible values.
Inside a location invariant only a subset of the assertion language is allowed. The RSL predicates as well as the modalities cannot occur inside of them.
Assertions inside a **FenceRel** or explicitly under a modality would typically be a part of a location invariant, as a release fence prepares the assertion to be given away via a write to a location and modalities occur because of fences. If we wanted to use a relaxed write to write 1 to the location with the example invariant we looked at above, we would use **FenceRel**((**acc**(a) && (*a == 7))) before the write. Therefore only assertions

allowed in a location invariant should occur in these places, but as they are not location invariants themselves, `v` is not available here. It is not needed either, since the assertion would generally be a variant of a location invariant for a specific value of `v`.
The separating conjunction $*$ is written as $\&\&$. This is in-keeping with the implicit dynamic frames style syntax used in the Viper language and avoids confusion with the multiplication operator.
An acquire predicate where some values have already been read can be explicitly expressed instead of the assertion being rewritten. For example, let's assume we want to express the acquire predicate for location `a` which has the location invariant we have seen above and where the value 2 has already been read:

$$\texttt{\textbf{Acq}}(\texttt{a}, ((\texttt{v} \ == \ 0)\ ?\ \texttt{\textbf{True}} \ : \ (\texttt{\textbf{acc}}(\texttt{a})\ \&\&\ (\ {}^{*}\texttt{a} \ == \ 7)))\,\texttt{\textbf{[}}2 := \texttt{\textbf{emp]}}\,)$$

This is useful, since it means that no syntactically different assertions will show up because of already read values. It is easy to detect that the assertion is the same as the one used during allocation. The information about previously-read values can be encoded separately.

4. Tool Overview

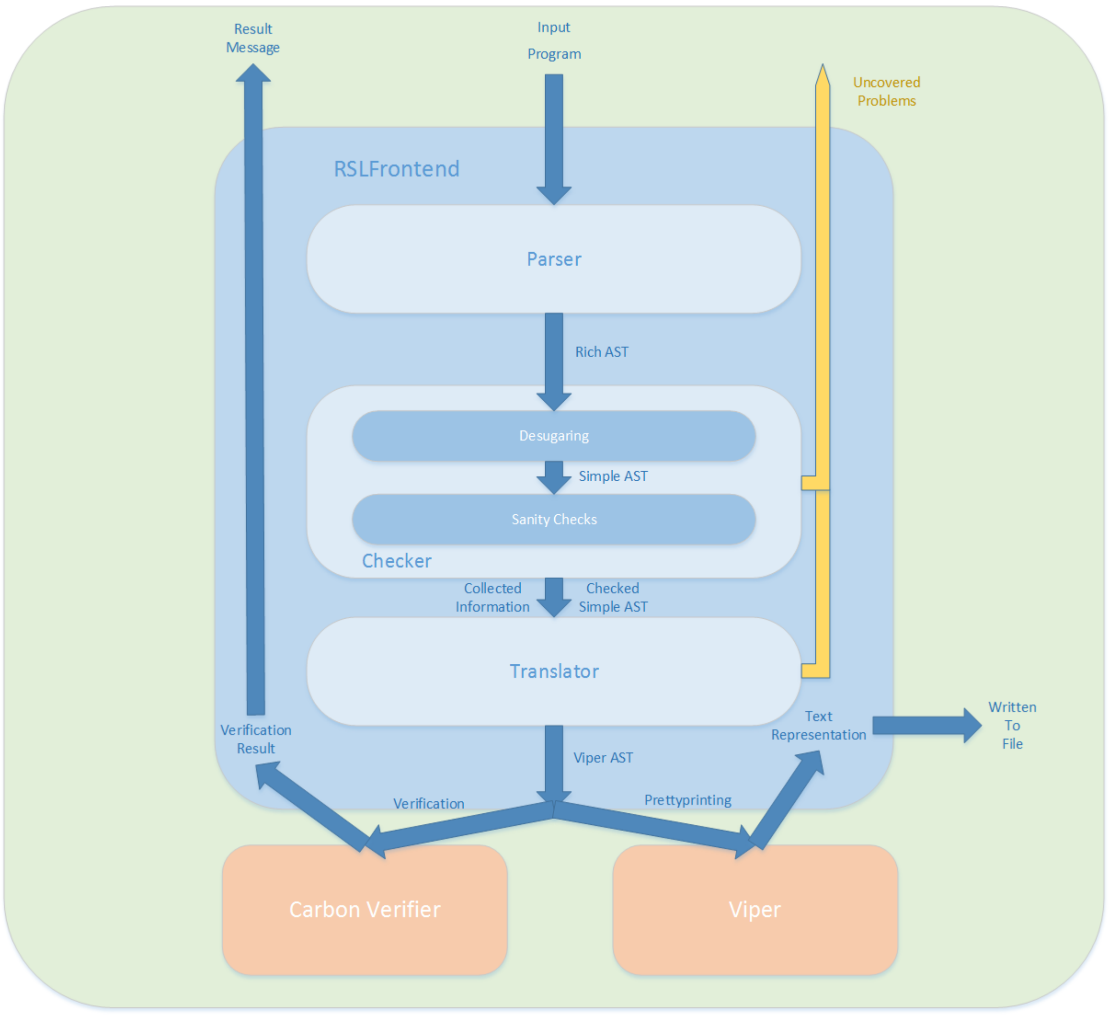

Figure 4.1.: Tool Overview

The RSL frontend tool takes a program written in the input language described in chapter 3 and tries to translate it into a Viper AST. The user has the choice of whether he wants to verify the program or whether he wants a text representation of the resulting AST.
Internally, the tool consists of three main stages. First is the parser, which creates an AST representing the program in the full input language. This AST is given to the Checker stage, where it gets transformed to eliminate certain expressions and replace them with statements and simpler expressions. We call this process the Desugaring. We will refer

to the subset of the expression language that remains after the Desugaring as ”simple expressions”. Afterwards, certain checks are performed on the simplified AST, for example to make sure that atomic locations are not used in non-atomic accesses. If problems are encountered, these are reported to the user and no translation is attempted. On top of this, the Checker collects additional information about the program, such as a mapping of fork tokens to the methods that were forked on them.
This information together with the simplified AST is then given to the Translator stage, if checking was successful. This stage starts creating the Viper AST, but there are also a few checks that are performed in this stage. This is because they are easy to do using some context that the translation needs to keep anyway, whereas the Checker would need to do additional work. If the translation is completed successfully, the Viper AST is either given to a Carbon Verifier instance for verification or a text representation is generated using the pretty-printing functionality provided by Viper. In case of an unsuccessful verification, the error messages from Carbon are displayed, together with the position and statement in the original input language file in whose translation Carbon reports the problem. We will examine the first two stages in the following subsections; for the details of the actual translation see chapter 5. The tool itself is implemented in Scala.

4.1. Parser

The parser is implemented using Scala parser combinators[7]. Compared to a parser generator framework, the resulting parser is not as efficient, but for the scale of examples this tool is expected to be used on that is not really relevant. Their advantage is ease of use and quite readable resulting code. The explanations on [8] proved particularly useful in understanding the parser combinators. The specification of the parser can be found in the appendix section A.1 on page 32.

4.2. Desugaring

The input language described in chapter 3 contains some expressions that could actually be viewed more as statements in expression positions. To simplify the translation later on, there is a simpler version of the input language that is used internally, with fewer expressions. The first step of checking a given program is desugaring it from the richer input language into the simpler one. The expressions that are eliminated in the desugaring are the conditional expression and all kinds of memory accesses. It is then possible to translate all remaining expressions directly into Viper expressions, whereas memory accesses would always need to be translated into a sequence of statements.

4.2.1. Statements

To desugar the program, the AST is traversed recursively and a new AST containing only simple expressions is built up. Whenever a statement containing an expression is encountered, the expression is desugared into a statement and a simple expression. In place of the old statement, a sequence statement containing the statement obtained from desugaring the expression and the old statement with the simple expression instead of the old expression is inserted into the new AST. We minimize unnecessary nesting of sequence statements by concatenating the inner sequences instead of wrapping sequence statements inside each other. For general loops, the statement obtained from desugaring the condition is also added to the end of the loop body. During the desugaring we also detect certain kinds of loops that will receive special treatment in the translation. See subsection 4.2.3 for details.

4.2.2. Expressions

The interesting part of the desugaring is the desugaring of the expressions themselves, so we will give a little more detail on that. It is performed in a recursive function **desugar** that takes an expression and returns a statement and a simple expression.
For literals and local variables, a sequence statement containing an empty sequence is returned together with the original expression. These unneeded statements nicely disappear when enclosing sequence statements are built using the nesting avoidance mentioned earlier.
For arithmetic, boolean and comparison operations, **desugar** simply calls itself on the children and returns the sequence of statements from the recursive calls together with a new version of itself where the children are replaced by the simple expressions that were returned for them.
In place of a conditional expression, a fresh local variable is declared and an if statement mirroring the form of the expression is created. The statement returned for $(b\ ?\ e\ :\ e')$ would look like:

```
var freshvar;
stmfromdesugaring($b$);
if expfromdesugaring($b$)
 then {
stmfromdesugaring($e$);
freshvar := expfromdesugaring($e$);
} else {
stmfromdesugaring($e'$);
freshvar := expfromdesugaring($e'$);
}
```

The expression returned is then simply `freshvar`. The reason we remove conditional expressions (despite the fact that the Viper language has conditional expressions), is that the expressions in the branches might be non-simple. Statements from desugaring e and e' should be executed conditionally - simply putting them into the returned statement and

returning a conditional expression again would not in general make sense. Memory accesses that occur in a branch should only happen if the branch is taken and they might not even be allowed for a branch that is not taken.
The different expressions accessing memory are all fairly simple to desugar. They each correspond to a statement that does the same memory access. This statement is returned, preceded by statements from the desugaring of expressions that occur inside and with all expressions replaced by the desugared simple expressions. The expression is either the local target that was assigned in the memory access statement or in case of a write access the desugared expression used as the RHS in the write.

4.2.3. Wait on Atomic Loops

Loops that just wait on an atomic access in a simple way can be encoded in a simpler form than general loops. This has the additional advantage that no loop invariant needs to be specified by the user for this encoding to work. Since the desugaring would transform the loops into a more complicated form, it is convenient to detect them during this process. Loops of the form

$$\mathbf{while}\,(\mathtt{target} := \mathtt{[loc]}_{\text{Acq/Rlx}} == \textit{simpleExp})\ \{\mathbf{skip}\}$$

are replaced by a special **WaitOnAtomicRead** node. Similarly, loops of the form

$$\mathbf{while}\,(\mathtt{target} := \mathbf{CAS}(\mathtt{loc}, ov, nv)_{any}\ != ov)\ \{\mathbf{skip}\}$$

are replaced by a **WaitOnCAS** node if ov and nv are simple expressions.

4.3. Checks

After the desugaring is completed, the Checker first finds all allocation nodes and constructs a mapping of heap location names to their allocation site. This is used as a reference for what the intended usage mode of a heap location is in later checks. Heap location nodes have two flags **isAtomic** and **isRMW** to record their intended usage mode. In some places where heap locations occur, the parser cannot fully determine how to set the flags, so in that case it will set them to the nonsensical combination **!isAtomic** and **isRMW**. When such a heap location is encountered in a check, the Checker sets these flags according to the allocation site it finds in the map. A similar mapping from method names to the method nodes is built in the beginning of the following traversal, used for example when checking a fork statement to make sure the method exists and the arguments match. During the checking traversal, a map from fork tokens to the respective fork statement nodes is also built, to be used when encountering a join statement. The method and token maps are also given to the translator stage for further use. The properties checked are the following:

- The identifier used in an allocation statement is only used in one.

- The identifier used in an allocation statement is not one of the reserved ones (`v` and `Result`)
- Different methods have different method names.
- Reserved identifiers `Result` and `v` are not used as formal arguments in a method.
- Heap locations that are used are allocated somewhere (they are in the heap location - allocation site mapping).
- Inside atomic accesses and predicates over atomic locations the heap location used was allocated as atomic and with the proper kind of access mode (RMW or Release-Acquire).
- Inside non-atomic accesses and predicates over non-atomic locations the heap location used was allocated as non-atomic.
- Local variables are not named `Result` or `v` in their declaration statement.
- Each fork token is only used in one fork statement.
- The method name in a fork statement is the name of a defined method (it is in the method name - method mapping).
- `Result` is not used as an actual argument in a fork statement. This simplifies the translation of join and is not really a limitation, as it is always possible to introduce an auxiliary variable.
- The number of actual arguments supplied in a fork statement matches the number of formal arguments of the forked method.
- The token used in a join statement is actually a token that a thread was forked on (it is in the token - fork mapping).
- The RSL predicates (Rel, Acq, RMWAcq, Init, Uninit, Up, Down) are not nested inside each other.
- `v` is only used inside location invariants.

Additionally, the following checks are performed by the translator stage, but we also list them here for an easier overview:

- Local variables that are used were declared before, either explicitly or by being a formal argument of the enclosing method.
- Location invariants contain no local variable accesses, as they would not make sense in a different method.

5. Translation

The Translator stage gets the simple AST that the Checker created, along with the **ThreadTokens** and **MethodDecls** maps. It builds up a Viper AST according to the encoding detailed in [3]. We will explain important parts of the encoding as we go, but for more in-depth reasoning on why the encoding works please refer to the cited paper. The first thing the Translator does even before starting to build the new AST is finding all location invariants that appear in the program and assigning them a numbering. This will be used to model the RSL predicates that contain location invariants. The numbering is done by a simple recursive function that traverses the AST and pattern matches on a trait **ContainsInv** that is used to mark all nodes that contain a location invariant. These are allocation statements and the Rel/Acq/RMWAcq assert nodes. The invariant is then assigned one number for the whole invariant (used in the encoding of Rel later) and a set of numbers for all immediate conjuncts, which is useful for supporting the splitting of Acq we have seen in section 2.1.2. If the invariant or some of its conjuncts have already been encountered before, no new number is assigned for these parts.
The actual translation is done in four methods that recursively call each other: **translateMethod**, **translateStm**, **translateExp** and **translateAssert**. **translateMethod** returns a Viper method, **translateStm** returns a Viper statement and the other two return a Viper expression. Here we really use that the AST was desugared before, because otherwise not all input language expressions could be translated into Viper expressions.
Heap locations are modelled by references with fields `val`, `init`, `rel` and `acq`, so these fields are added to the Viper program node that is the root of the resulting AST. The `val` field is of type `Int` and used to model the value stored for non-atomic locations. For atomic locations, the value is not actually modelled, wherever it would be relevant it is assumed to be an arbitrary integer because some other thread might have written to it. The `init` field is of type `Bool` and represents whether a location has been initialized. For non-atomic locations we use the value of the `init` field to encode the initialization status. In contrast, the initialization of an atomic location is represented by a wildcard permission to the `init` field and not its value. The `rel` and `acq` fields are only used for atomic locations. The `rel` field is of type `Int` and is used to store the number we assigned to the location invariant the heap location was allocated with. The `acq` field is a `Bool` and is true for release-acquire locations and false for RMW-acquire locations.
We also add the uninterpreted predicate

```
predicate AcqConjunct(l: Ref, idx: Int)
```

and the uninterpreted function

```
function valsRead(l: Ref, i: Int): Set[Int]
        requires AcqConjunct(l, i)
```

to the program, which will be used to model the acquire predicates from RSL later on.
To be able to model a variable getting an arbitrary value, the methods `havocedInt`, `havocedBool`, `havocedRefSet` and `havocedIntSet` are added to the program. These are very simple methods that don't take any arguments and just have one formal return variable of the respective type they are supposed to produce havoced values for. On top of these, we iteratively call **translateMethod** on all the methods of the input program and add the resulting Viper methods to the sequence of methods in the root of the new AST.

5.1. Methods

Local variable declarations are represented in Viper by a sequence of **LocalVarDecl** nodes that is held directly in the method node. Since the local declarations in the input language can occur somewhere in the body, **translateMethod** initializes a global map **currentLocals** to an empty map and the translation of the body will add the needed variables to it. It maps variable names to Viper types, since heap locations allocated in the method will also be represented by a local variable in Viper. Similarly, heap locations that are allocated in a different method but accessed in the one being translated need to be collected and added to the formal arguments of the Viper method. For this a global set of strings **currentNeededVars** is initialized to an empty set. After the translation of the body, pre- and postconditions it contains the names of all heap locations needed in the current method. For every name that is not also in **currentLocals** a declaration is then added to the formal arguments of the Viper method. On top of that, the formal arguments of the input method and the `Result` variable are put in a global set **currentFormals**. This makes it easy to check if a local variable used has been declared by checking that it is in **currentLocals** or **currentFormals**.
The precondition of the Viper method is simply the precondition of the input language method, translated using **translateAssert**. The postcondition would be the translated postcondition of the input language method as well, but because post conditions may mention heap locations that were allocated inside the body and these are local variables to Viper this does not work as easily. Instead, we put the translated postcondition in an exhale and attach it to the end of the method body. After that we add another exhale statement

```
exhale (forperm [val] NonLeakingCheckVar :: false)
```

that basically checks that no permission is left over, to make sure that no method is leaking permission. The body of the input method is given to **translateStm** for translation.

5.2. Statements

The **translateStm** method relies a lot on helper functions that mimic the macros that have been previously used for hand-encoding examples. Macros are not actually present at the AST level in Viper, otherwise we would have generated macros as well to improve the readability of the program after the translation. We will not always mention when part of the translation is done by using such a helper method, only when it seems of special interest because the same helper is used frequently. The **translateStm** method pattern matches on the different possible statement nodes. We will examine the different statements in the following subsections.

5.2.1. Local Variables and Sequential Control Flow

A local declaration does not actually show up in the body of a Viper method, so we return an empty sequence statement for these. To reflect that the local has been declared, we instead add its name and the type `Int` to the **currentLocals** map explained in the previous section. For a local assignment statement, we correspondingly check that the variable is contained in **currentLocals** or alternatively in **currentFormals**. If it is in **currentLocals**, we also make sure that the type of it is `Int`, otherwise it would be a heap location allocated in the method and not actually a local variable of the input language. The same check is also made in all the other statements that assign to a local variable, like atomic reads and joins. We will omit mentioning it there, since it works in exactly the same way. The local assignment is then translated into a Viper **LocalVarAssign** using **translateExp** to translate the right hand side.
The skip statement of the input language is simply translated into an empty sequence statement. For a sequence statement, we map **translateStm** over the contained sequence and return a Viper sequence statement containing the resulting sequence of Viper statements. We initially tried avoiding unnecessary nesting here as well, like we did during desugaring, for example to get rid of the empty sequence statements generated by local declarations and skips. This proved problematic however, when including comments that help the readability of the pretty-printed representation of the translation result. These are usually stored in the enclosing sequence statement for statements that get translated into more than one Viper statement, so the comments might be lost if we simply removed sequence statements. In contrast to the desugaring, where every simple expression encountered would actually generate an unnecessary empty sequence statement, here these are few and other nesting corresponds to desirable grouping of logically related statements. We therefore forgo any nesting avoidance in the translation of sequence statements.
If and while statements are structurally very similar to the corresponding Viper statements. Their conditions are given to **translateExp**, the body of the while as well as the branches of the if are translated by a recursive call to **translateStm** and the invariant of the while is translated by **translateAssert**. The respective Viper statement is then assembled from the results. While statements without invariant from the input language don't receive any special treatment, they are parsed as a normal while statement with a true literal as the invariant. The special cases of while statements waiting on an atomic access that were detected during desugaring will be treated in the subsections that explain the atomic accesses.

5.2.2. Fork/Join

We encode a fork conceptually by simply exhaling the precondition of the forked method with the formal arguments replaced by the actual arguments. To enable the join to access the values of the actual arguments at this point, we insert a label generated from the fork token before the exhale. Since we checked that fork tokens are unique and the join statement contains the token as well, we do not need to keep track of these labels, the join can simply generate it in the same way. We find the declaration of the forked method using the **MethDecls** map generated by the Checker and look up the precondition we need to exhale. We want to use **translateAssert** to translate the precondition, but **translateAssert** includes checks to make sure local variables it encounters are declared in the current method. The formal arguments of the forked method are not declared in the current method, but they might be mentioned in the precondition. Since we will replace them afterwards, we do not want **translateAssert** to complain about them, so we use a flag **replacementToHappen** to disable the checks here. Since we know that the forked method exists in the program, the precondition will be translated with proper checking when the forked method itself is translated, thus we can be sure not to miss any problem by omitting the checks when forking. We then use the replace functionality implemented for Viper nodes to replace occurences of the formal arguments by the translations of the corresponding actual arguments. To translate the actual arguments, **translateExp** is used.
To model a join, we havoc the local variable that the result is assigned to and then inhale the information from the postcondition of the joined method. Here we need to replace the formal arguments of the method by the values of the actual arguments at the point the method was forked. For this we use labelled old expressions with the label generated from the fork token and the translation of the actual arguments of the fork inside. After that, we need to replace occurrences of `Result` in the postcondition by the local variable we conceptually assign the return value to. We disallowed using `Result` as an actual argument in a fork to prevent erroneously replacing it inside an actual argument here. Doing the replacement of the result variable first (eliminating this danger) would have introduced another possible naming clash. If the variable we replaced the result with happened to have the same name as a formal argument of the forked method, we might accidentally replace it afterwards with an actual argument, which would be a problem. Therefore disallowing the use of `Result` inside actual arguments seemed to be an easy and not overly restrictive way of avoiding the problem. To find the forked method and the actual arguments we use the **ThreadTokens** map and to find the formal arguments and the postcondition of the method again the **MethodDecls** map. The postcondition is translated before the replacement using **translateAssert** and the **replacementToHappen** flag just like in the fork case. Havocing the local variable is achieved by a method call to the `havocedInt` method described in the beginning of this chapter. Inhaling information about the value of the variable after the havoc allows the verifier to assume that the value the variable got from the havoc must have been some value that is compatible with this new information.

5.2.3. Non-atomic Locations and Accesses

To allocate a non-atomic heap location, we add its name to **currentLocals** with type `Ref` to declare a local variable of reference type to represent it. We assign a new reference to this variable by generating a Viper **NewStmt**. After that, we inhale the encoding of the Uninit predicate. For this, a helper method taking the name of the location is used, that will also be reused in **translateAssert**. Uninit(l) is encoded as:

```
acc(l.val) && acc(l.init) && !l.init
```

The first two conjuncts ensure that we have full access to the relevant fields for our new non-atomic location. The last conjunct represents the fact that the location has not been initialized yet. This whole expression is returned by the Uninit helper and inhaled in the translation of the non-atomic allocation statement.
For a non-atomic write, we assign the translation of the right hand side (done by a call to **translateExp**) to the `val` field of the location. Afterwards we assign `true` to the `init` field to represent the fact that the location is now initialized. The necessary access rights are all automatically checked by Viper upon a field access, so we do not need to explicitly encode checking them. To make sure that the location will be available in the enclosing method, we add its name to **currentNeededVars**.
For a non-atomic read, we also need to make sure the location we are reading from is available, so we add it to **currentNeededVars** as well. To be able to read from a location, we need to make sure that it was initialized, so we add an assertion ensuring that the `init` field is `true` before assigning the value of the `val` field to the local variable that is the target of the read. The access right to the fields will again be implicitly checked by the verifier. Since both fields are only read, a fractional permission will suffice, corresponding to the fractional points-to assertion we saw in the rule in section 2.1.1.

5.2.4. Atomic Locations and Accesses

Before explaining the translation of the statements dealing with atomic accesses, let us look at the encoding of the RSL predicates we will need. Like Uninit, these are expressions generated in helper functions, so we can use them in our translation. As we already mentioned in the beginning, the initialization of an atomic location is represented by access to the `init` field and not its value. To make the Init predicate duplicable however, it is not represented by full access but rather by a wildcard permission amount to the `init` field. A Rel($l, \mathcal{Q}$) is represented by a wildcard permission to the `rel` field of l and the invariant number of $\mathcal{Q}$ stored in the `rel` field. This makes release predicates duplicable like Init, but does not allow any release predicates to be given away with an invariant different from the one the location is allocated with. For an Acq($l, \mathcal{Q}$) we want to be able to split the invariant to allow different threads to gain different parts of it on a read. It is encoded as

```
acc(l.acq,wildcard) && l.acq == true
    && AcqConjunct(l, i) && valsRead(l, i) == Set[Int]()
```

where the last two conjuncts are added for every invariant number i that numbers one of the immediate conjuncts of $\mathcal{Q}$. The first two conjuncts come from the macro SomeAcq(l) and represent that this is an Acq for some invariant. The third conjunct is an instance of the `AcqConjunct` predicate. These predicates represent the conjuncts of the invariant in a flexible way. Since the SomeAcq part of the Acq is duplicable, it is possible to split an Acq, for example giving part of it away in a fork, by giving the duplicable part and some of the `AcqConjunct` predicates away. The last conjunct encodes previously read values of an Acq. For a simple Acq it asserts that the set of already read values is empty. If we want to represent an Acq where some values have been read, the set on the right hand side of the comparison contains these values. A RMWAcq(l, $\mathcal{Q}$) is very similar to an Acq. We do not need to track previously read values, so we do not need the `valsRead` conjunct. The value of `l.acq` is `false` to indicate that it is a RMWAcq. To make the whole RMWAcq duplicable, we do not need full access to the `AcqConjuncts` but we use again a wildcard permission instead. We end up with:

```
acc(l.acq,wildcard) && l.acq == false
    && acc(AcqConjunct(l, i),wildcard)
```

The last conjunct is repeated for all i numbering a conjunct of $\mathcal{Q}$ as before.
Translating an atomic allocation statement works exactly like in the non-atomic case, except that instead of Uninit we inhale Rel and Acq or RWMAcq for the location and invariant in the allocation statement. Note that we do not inhale any permission to the `init` field, corresponding to the fact that the location is not yet initialized.
The access mode (Rel/Acq or Rlx) of an atomic access is represented in the atomic read and write statements by a boolean flag **synchronizes** that is false for relaxed accesses. The details of the modalities needed for relaxed accesses will be explained in the next subsection, for now just assume that we have a way of putting a translated invariant under a modality using the helper functions **makeUp** and **makeDown** that both take and return a Viper expression.
To encode an atomic write $[l]_{wm} := e$, we first generate an assertion ensuring we have read access to `l.rel`. We then want to exhale the invariant associated with l. For this we generate for each number i in our invariant numbering an if statement that checks whether the value of `l.rel` is i and in that case exhales the translation of the invariant numbered with i. In the translated invariant, the special variable `v` is replaced by the translation of e. If the write is a relaxed one, the invariant is put under the up modality after this replacement. Afterwards, we add a statement inhaling Init(l).
The translation of an atomic read statement $x := [l]_{rm}$ begins by introducing an auxiliary variable (called `tmpSet` in the following) of type `Set[Int]`. We make sure to pick a name that is not used in the input program, but subsequent atomic reads will pick the same name, so we add a call to `havocedIntSet` to havoc it as the first statement of the translation. The next statement asserts Init(l) and SomeAcq(l) to make sure that the location is initialized and an acquire predicate is available. Then x is havoced using a call to `havocedInt`, expressing that an arbitrary integer might be read from l. We then want to inhale whatever we can gain from reading the value and record that we have read the value in `valsRead`. For this we generate the following code snippet for every i in our invariant numbering, where replacedinv(i) is the translation of the invariant numbered with i with `v` replaced by x. In case the read is relaxed, it is also placed under the down

modality.

```
if (perm(AcqConjunct(l,i)) == 1 && !(x in valsRead(l,i))){
    inhale replacedinv(i)
    tmpSet := valsRead(l,i)
    exhale AcqConjunct(l,i)
    inhale AcqConjunct(l,i) && valsRead(l,i) == tmpSet union Set(x)
}
```

This ensures that for every acquire conjunct available at the point of the read, if we haven't read the same value before we inhale the part of the invariant this conjunct represents. We also add the value that was read to the values `valsRead` returns for this location and invariant number from here on, by essentially telling the verifier to assume that the function `valsRead` returns the set it returned before unified with the value of x.
To encode a **WaitOnAtomicRead** loop

$$\texttt{\textbf{while}}\,(\texttt{target} := \texttt{[loc]}_{\text{Acq/Rlx}} == e)\,\{\texttt{\textbf{skip}}\}$$

we observe that all iterations except the very first and the very last one cannot gain any ownership. If the first iteration has read the value e, afterwards the acquire predicate will be modified to not allow any further ownership to be gained from the same value. Therefore we do not actually need a loop to model the behaviour of this loop. Instead, it is encoded as one atomic read to model the first iteration, followed by:

```
if (target == e) {
        atomicRead(loc,target) //another atomic read of the location
        assume target != e
}
```

This models that if we ever entered the loop (the value read was e in the first iteration), we will at some point break out of it by reading some value different from e. Only this last read will affect the state of the program, so we skip all intermediate reads of value e.

5.2.5. Fences

The difficult part about encoding fences and relaxed accesses is the representation of the up and down modalities. This is done by simulating three separate heaps (using a Viper domain and axiomatised functions between the heaps, for details see [3]). We can then conceptually talk about the reference of location l under the up modality by talking about the reference of l in the up heap. The reference in the up heap corresponding to a reference `r` in the normal heap is accessed by a call of the domain function `up(r)`. The modalities distribute over logical connectives and implications, so an invariant under a modality can be expressed by replacing all the references in the original invariant by their counterpart in the respective other heap. This replacement is what the helper methods **makeUp** and **makeDown** do in our translations.
A **FenceRel** node contains the invariant that it is supposed to place under the modality. For translating it, we simply exhale the translation of this invariant and inhale the same

invariant after transforming it with **makeUp**.
The **FenceAcq** is more complicated, since it needs to find what is under the down modality in the current state and remove the modality from all of it. This makes sense, since there is no use for an invariant under a down modality without removing it, whereas in the release case we don't want to place additional information under the modality that should remain free of it. To do this, we use an auxiliary variable of type `Set[Ref]`. The translation of **FenceAcq** chooses a name not used by the input program for it, we will simply refer to it as `refSet` here. The `refSet` is havoced by a call to `havocedRefSet`. Through inhales of quantified permissions we make sure that exactly the references `r` for which we have some permission to `down(r).val` are in the `refSet`. Then we inhale the same amount of permission to `r.val`, equate the values in both heaps and exhale the permission we held in the down heap.

```
refSet := havocedRefSet()
inhale (forall r: Ref :: (r in refSet)
     ==> heap(r) == 0 && (!is_ghost(r) && perm(down(r).val) > none))
inhale (forall r: Ref :: perm(down(r).val) > none && !is_ghost(r)
     ==> (r in refSet))
inhale (forall r: Ref :: (r in refSet)
     ==> acc(r.val, perm(down(r).val)))
inhale (forall r: Ref :: (r in refSet) ==> r.val == down(r).val)
exhale (forall r: Ref :: (r in refSet)
     ==> acc(down(r).val, perm(down(r).val)))
```

This is repeated for every field and (without the equation of values) also for every `AcqConjunct`. After all this is done we have effectively moved the invariant from the down heap to the normal heap.

5.2.6. CAS

The idea of using multiple heaps introduced in the previous section is reused in the encoding of CAS to be able to find the overlapping part T of the gained and given up location invariants. For this another heap temp is introduced and the location invariant gained from the read is inhaled to the temp heap at first. The exhale corresponding to the write then tries to exhale everything it can from the temp heap and only the parts of the location invariant not available in the temp heap (corresponding to P) from the normal heap. Afterwards, everything still in the temp heap corresponds to the actually gained part of the invariant A and we move the temp heap to the normal heap (or the down heap if the read-mode was relaxed) just like we moved the down heap to the normal heap in the FenceAcq statement. Inhaling to the temp heap is done by replacing all references in the translated invariant that is inhaled by their counterpart in the temp heap. For this we use a helper function **makeTemp** that works just like **makeUp** and **makeDown**. The most complicated part is the exhale that in a sense dynamically chooses where to exhale from. For this we actually use a different function to translate the input language assertion instead of **translateAssert** that directly incorporates the encoding of this as well as the replacement of `v`.

An **acc**(a) is translated by this function to

```
let p == ((perm(temp(a).init) < write ? perm(temp(a).init) : write))
in acc(temp(a).init, p) && acc(up(a).init, write - p) &&
((p > none && write - p > none ==> temp(a).init == up(a).init) ==>
(old[CAS_0](perm(temp(a).init)) > none ? temp(a).init : up(a).init)))
&&(let p0 ==((perm(temp(a).val) < write ? perm(temp(a).val) : write))
in acc(temp(a).val, p0) && acc(up(a).val, write - p0) &&
((p0 > none && write - p0 > none ==> temp(a).val == up(a).val)
==> //subsequent conjuncts of the invariant
```

where fresh names for `p` and `p0` are picked every time. The **old**`[CAS_0]` refers to a label that is inserted after the inhale to the temp heap and given to the translation function as an argument. If the write was not a relaxed write, everywhere `up(a)` is used here, `a` would be used. To determine this, the translation function takes the **synchronizes** flag from the write as an argument. This translation is actually a little more complicated than it would need to be, that is in preparation for supporting fractional access notations in the input language. As it is, our **acc**(a) denotes full permission, denoted by the **write** in the code snippet. The usual translation of **acc**(a) is

```
acc(l.val) && acc(l.init) && l.init
```

as explained in section 5.4. We now want to take as much of the access permissions from the temp heap as possible. For the `init` field, we assign the minimum of the permission amount we need (**write** here, but might be something different if we supported fractional permissions in the input language) and the permission amount available in the temp heap to `p`. This amount we then take from the temp heap and the rest of what we need (**write** `- p` here) from the normal/up heap in the second line. If we actually took permission from both heaps, we want to make sure that the values match, otherwise we would be in an odd state. This is checked in the implication in the third line. If the values matched (or we only took permission from one heap so the implication became trivially true) we can then just take the value in the temp heap if we had any permission to it or the value in the normal/up heap otherwise. With all this, we have replaced the **acc**`(l.init) && l.init` part of the usual encoding seen above.
The same pattern is repeated for the `val` field using `p0` instead of `p` afterwards. Here however, we do not access the value like we did for `init`. The actual access to the `val` field will happen in one of the following conjuncts of the overall invariant. We therefore need to put the rest of the invariant under the implication that the values of the `val` field are consistent.
The biggest issue we faced in implementing this translation was getting everything that appeared in conjuncts to the right of an **acc** into the implication. To achieve this, the translation function takes an argument **rightOfAnd** of type **Option[Assert]** and the translation of **acc** will recursively call translate on **rightOfAnd** (if it is defined) and put the result in the RHS of the implication. The translation of And and SeparatingConjunction consequently recursively calls the translation function on the left child with the right child (conjuncted with **rightOfAnd** if it was defined) given as **rightOfAnd**. The following nodes are first translated and then wrapped in an And with **rightOfAnd** if **rightOfAnd** is defined.

Other boolean and arithmetic operations as well as comparisons are directly translated to the corresponding Viper expression using recursive calls to translate the children. `v` is simply replaced by an expression that the translate function takes as an argument. $^*a$ is translated, just like the access to `a.init` we saw above, as

```
(old[CAS_0](perm(temp(a).val)) > none ? temp(a).val : up(a).val)
```

where again `up(a)` is replaced by just `a` if the write was not relaxed. This conceptually means that if we had any permission to the value in the temp heap after inhaling, we take the value from the temp heap. Otherwise we take the value from the normal/up heap depending on the write mode.
To model the **WaitOnCAS** loop, we use the fact that in the failing case no ownership is transferred. Since the loop is simply waiting for a CAS to succeed, it is enough to model it with one succeeding CAS. The previous iterations of the loop would not affect the ownership we hold, as they would all be failing CAS operations.

5.3. Expressions

All simple expressions map directly to a Viper expression, so **translateExp** pattern matches on the kind of expression it receives, calls itself recursively on the subexpressions and creates the respective Viper expression with the results of the recursive calls. The only case in which something more happens is the case of a **LocalVar** node. Here there is additionally a check to make sure that the local variable has been declared. If the name of the variable is not in **currentLocals** or **currentFormals**, a problem is reported.

5.4. Assertions

Just like for expressions, a lot of the **AssertExp** nodes map directly to a Viper expression. The encoding of the RSL predicates was explained in section 5.2 as they were needed in the translation of statements. In the **translateAssert** method the helper functions mentioned above are again used to generate their encoding. The locations mentioned in them are also added to **currentNeededVars** to make sure they are available in the current method. To translate the modalities, we translate the assertion they contain and use **makeUp** and **makeDown** respectively to replace the references inside by their counterparts in the respective heap. The only interesting assertions to translate that still remain are the separating conjunction, **acc**(l) and $^*l$. Due to the way Viper's conjunction works, we can simply translate separating conjunctions into standard Viper conjunctions, which is very convenient. As we have seen in chapter 3, a points-to assertion $l \overset{1}{\mapsto} e$ would be expressed as **acc**(l) && ($^*l \; == \; e$) in our input language. Conceptually, our **acc**(l) is the same as $l \overset{1}{\mapsto} \_$ in RSL. We therefore encode it as

```
acc(l.val) && acc(l.init) && l.init
```

which gives us full access to the location's `val` and `init` fields and ensures that the location is initialized. Our $^*l$ can then simply be translated to `l.val`, allowing us to use it inside of other assert expression like the equality comparison above.

6. Evaluation

To evaluate the tool, we examined its behaviour on a number of examples. Some of these are examples from the papers on RSL and FSL, others were handcrafted to exercise different parts of the encoding. Some of these fail checks at different stages of the checking and translation process, some (expectedly) don't verify. For those that do verify, we include a second version with some slightly tweaked postcondition that is expected to fail. This ensures that the verification does not just reach some inconsistent state and would accept any postcondition.

6.1. Examples

We will give a short description of the main features exercised for each example. The full code of the examples in the input language is given in the appendix section A.2 on page 34.

Access without Owning This is a very simple example in which a non-atomic heap location is allocated in one method and accessed in a second method. The second method has no ownership of the location and consequently verification fails.

CASExample This is an example handcrafted to exercise different versions of the CAS statement. It contains CAS with and without relaxed access modes, where the relaxed case makes use of fences. It also includes both WaitOnCAS loops and a CAS that just occurs inside an If. It also contains a slightly more complicated location invariant than most of the other examples. The example verifies and there is a second version with a wrong postcondition that does not verify.

Fork Join Result This is another handcrafted example. It exercises forking and joining a method that takes multiple arguments and uses the `Result` variable to return a value. It verifies and the version with a wrong postcondition does not verify.

FSL Figure 2 This example is our rendition of the program presented in Figure 2 of [2]. It uses two atomic locations to signal different threads that access two non-atomic locations. The accesses to the atomic locations are relaxed and fences are used for synchronization. The program verifies and a tweaked version with a wrong postcondition does not.

FSL Figure 2 Variant This example closely resembles the previous one. Instead of two atomic locations only one atomic location is used. Its location invariant is split and the parts are given to the different threads. This exercises our encoding of acquire conjuncts. It verifies and the version with a wrong postcondition does not verify.

Nonatomic Sequential Simple Example This example only consists of one relevant method. It exercises non-atomic allocation and writes as well as local variable assignment and conditional expressions. It verifies and a variant with a wrong postcondition does not.

Release Write on Non-Atomic In this example, a non-atomic location `al` is allocated and subsequently used in a release write operation. The checker correctly identifies the problem and rejects the program. The problem description the checker gives is `Problem with Heap Location: Tried to use nonatomic heap location al in atomic write operation`.

RSL Figure 7 This example is a slightly modified version of the example presented in Figure 7 of [1]. It consists of multiple methods implementing a lock. In the paper a predicate J is used to model some arbitrary resource that is guarded by the lock. Since our input language doesn't give us any way to specify such a J, we instead use a non-atomic heap location `z` allocated in an auxiliary method as this resource. The lock in our example guards **`acc`**`(z)`. This example verifies and a version where we add **`False`** to the postcondition of the lock method does not verify.

RSL Figure 7 non-relaxed This is a variant of the previous example where the access mode of the CAS used inside the lock method is AcqRel instead of AcqRlx. This version also verifies and the corresponding version with the wrong postcondition does not verify.

RSL Figure 8 This example is our version of Figure 8 from [1]. It is a simple message passing idiom using release/acquire atomics. It also exercises the WaitOnAtomicRead loop. The example verifies and a version with a wrong postcondition does not verify.

Signal two threads with one location This is very similar to the FSL Figure 2 Variant. The difference is that it uses release/acquire atomics instead of relaxed accesses and fences. It therefore exercises the splitting of the Acq predicate in a slightly different scenario. The example verifies and another version with a wrong postcondition does not.

Undeclared Local In this example a method tries to assign to a local variable `readvar` without declaring it. The Translator correctly detects this and rejects the program. The problem description given to the user is `Tried to assign to Local Variable readvar without declaring it.`

6.2. Timing

We compare the total time spent in the front-end (including parsing, checking and translation) with the time spent in the verifier. Table 6.2 gives the timings measured in one example run. We can observe that the time spent in the frontend is typically much smaller than the time needed for verification. We also see that the failing variants of the examples do not systematically differ from the originals.

Example	Frontend time [s]		Verifier time [s]	
Version	correct	wrong	correct	wrong
Access without owning	-	1.772	-	10.41
CASExample	0.797	0.363	56.295	53.559
Fork Join Result	0.054	0.017	11.117	11.111
FSL Figure 2	0.521	0.271	14.849	17.071
FSL Figure 2 Variant	0.351	0.372	19.749	13.664
Nonatomic Seq. Simple Ex.	0.038	0.05	10.581	9.837
Rel Write on NA	-	0.006	-	-
RSL Figure 7	0.138	0.197	12.625	11.741
RSL Figure 7 non-relaxed	0.131	0.089	11.766	10.296
RSL Figure 8	0.089	0.075	11.407	10.229
Signal 2 threads 1 location	0.154	0.212	11.783	11.13
Undeclared Local	-	0.047	-	-

Table 6.1.: Example Runtimes

7. Conclusion and Future Work

We presented a prototype verifier for weak memory programs based on an encoding of relaxed separation logics into Viper. We have seen a number of interesting examples that can be succesfully verified using this newly implemented tool. Nonetheless, there are still some examples that can't be expressed in our input language. Extensions of the front-end and input language to cover additional concepts are therefore an interesting topic for future work.
One example of such an extension would be adding support for *ghost locations*, as introduced in [5] and necessary for the running example of the paper. In this particular case, a lot of groundwork has already been laid in the front-end. The main part missing is a way for the user to specify ghost locations in the input language. The translation already uses the encoding of the parallelHeaps domain with ghost locations presented in the appendix of [3]. Since there is no way to allocate ghost locations, we do not yet need to track the ghost status of locations and consequently simply assume at the beginning of each method that all locations given to it are real locations.
The assertions of the current input language include only the $\mathbf{acc}(x)$ assertion denoting full access to the location x. It would be possible to add another assertion to denote fractional access permissions to locations. As we have seen in section 5.2.6, for the CAS we already have an encoding that is more general than necessary for the current input language in order to be easy to reuse for fractional accesses. Therefore, actually adding the assertion would not require a lot of conceptually new things.
A third possible addition to the input language would be the `rewrite` statement presented in the appendix of [3]. This statement allows the programmer to indicate entailment between different invariants in Acq predicates. It can be used to split Acq predicates in more flexible ways than the encoding using conjuncts supports on its own. Adding it should be straight-forward, as it is just another statement that needs to be handled and does not interact with the rest of the translation.

A. Appendix

A.1. Parser Specification

```
readmode ::= Acq | Rlx | SC | NA
writemode ::= Rel | Rlx | SC | NA
allocmode ::= RMW | AR
Ident ::= a letter, followed by zero or more
              letters or numbers
Aexp ::= Ident | Integer
| (Aexp + Aexp) | (Aexp * Aexp) | (Aexp - Aexp)
| (Bexp ? Aexp : Aexp)
| Ident  := [Ident]_readmode
| [Ident]_writemode := Aexp
| Ident := CAS(Ident, Aexp, Aexp)_readmodewritemode
Bexp ::= not Bexp | (Bexp or Bexp) | (Bexp and Bexp)
| (Aexp == Aexp) | (Aexp <= Aexp) | (Aexp != Aexp)
| (Aexp < Aexp) | (Aexp >= Aexp) | (Aexp > Aexp)
Stm ::= Ident := Aexp | var Ident | skip | [Stm;]+
| if Bexp then {Stm} else {Stm}
| while Bexp invariant Assert {Stm}
| while Bexp {Stm}
| Ident := alloc()
| Ident := alloc(Assert)_allocmode
| Ident := fork(Ident [, Aexp]*)
| Ident :=  join(Ident)
| Ident := [Ident]_readmode
| [Ident]_writemode := Aexp
| Ident  := CAS(Ident, Aexp, Aexp)_readmodewritemode
| FenceAcq | FenceRel(Assert)
Method ::= method Ident ([Ident][,Ident]*)
           pre Assert post Assert {Stm}
Program ::= [Method]+
AssertExp ::= *Ident | Integer | True | False | Ident
| (AssertExp + AssertExp) | (AssertExp * AssertExp)
| (AssertExp - AssertExp) | not AssertExp
| (AssertExp or AssertExp) | (AssertExp and AssertExp)
| (AssertExp == AssertExp) | (AssertExp != AssertExp)
| (AssertExp <= AssertExp) | (AssertExp >= AssertExp)
| (AssertExp < AssertExp) | (AssertExp > AssertExp)
Assert ::= acc(Ident) | AssertExp | (Assert && Assert)
| (AssertExp => Assert)
| (AssertExp ? Assert : Assert)
| Rel(Ident, Assert)
| Acq(Ident, Assert)
| Acq(Ident, Assert[Integer :=emp [, Integer := emp]*]
| RMWAcq(Ident, Assert)
| Init(Ident) | Uninit(Ident)
| Up(Assert) | Down(Assert)
```

A.2. Input Examples

Access without Owning

```
method foo()
pre True
post True
{
var lo;
lo := 5;
al := alloc();
[al]_NA := lo;
lo := ((1 == 1) ? 6 : 7);
lo := ((1 != 1) ? 8 : 9);
}
method bar()
pre True
post True
{
var readvar;
readvar := [al]_NA;
}
```

CASExample

```
method outerscope()
pre True
post ((acc(a) && ((*a == 7) or (*a == 8))) && (Result >= 7))
{
a := alloc();
lock := alloc(((v == 0) ?
            (acc(a) && ((*a == 7) or (*a == 8))) : True))_RMW;
[a]_NA := 7;
FenceRel((acc(a) && (*a == 7)));
[lock]_Rlx := 0;
t1 := fork(thread1);
t2 := fork(thread2);
var tmp;
tmp := join(t1);
Result := join(t2);
while(tmp := CAS(lock,0,1)_RlxRlx != 0){skip};
FenceAcq;
}

method thread1()
pre ((RMWAcq(lock,
        ((v == 0) ? (acc(a) && ((*a == 7) or (*a == 8))) : True))
    && Init(lock))
    && Rel(lock,
        ((v == 0) ? (acc(a) && ((*a == 7) or (*a == 8))) : True)))
post True
{
var tmp;
if(tmp := CAS(lock,0,1)_AcqRel == 0)then{
    [a]_NA := 8;
    [lock]_Rel := 0;
} else {skip};
}

method thread2()
pre ((RMWAcq(lock,
        ((v == 0) ? (acc(a) && ((*a == 7) or (*a == 8))) : True))
    && Init(lock))
    && Rel(lock,
        ((v == 0) ? (acc(a) && ((*a == 7) or (*a == 8))) : True)))
post (Result >= 7)
{
var tmp;
while(tmp := CAS(lock,0,1)_AcqRel != 0){skip};
Result := [a]_NA;
[a]_NA := 7;
[lock]_Rel := 0;
}
```

Fork Join Result

```
method foo()
pre True
post (acc(al) && (*al == 5))
{
var lo;
var tmp;
lo := 2;
al := alloc();
[al]_NA := 3;
token := fork(bar,lo,tmp := [al]_NA);
lo := join(token);
[al]_NA := lo;
}

method bar(a,b) pre (a > 0) post ((Result == 5) && (a > 0)){
Result := 5;
}
```

FSL Figure 2

```
method outerscope()
pre True
post ((acc(a) && (*a == 43)) && (acc(b) && (*b == 8)))
{
a := alloc();
b := alloc();
x := alloc(((v==0) ? True : (acc(a) && (*a == 42))))_AR;
y := alloc(((v==0) ? True : (acc(b) && (*b == 7))))_AR;
[x]_Rlx := 0;
[y]_Rlx := 0;
t1 := fork(left);
t2 := fork(right);
t3 := fork(middle);
var tmp;
tmp := join(t1);
tmp := join(t2);
tmp := join(t3);
}

method left()
pre (Acq(x,((v==0) ? True : (acc(a) && (*a == 42)))) && Init(x))
post (acc(a) && (*a == 43))
{
var tmp;
while(tmp := [x]_Rlx == 0){skip};
FenceAcq;
[a]_NA := (tmp := [a]_NA + 1);
}

method right()
pre (Acq(y,((v==0) ? True : (acc(b) && (*b == 7)))) && Init(y))
post (acc(b) && (*b == 8))
{
var tmp;
while(tmp := [y]_Rlx == 0){skip};
FenceAcq;
[b]_NA := (tmp := [b]_NA + 1);
}

method middle()
pre ((Uninit(a) && Uninit(b))
    && (Rel(x,((v==0) ? True : (acc(a) && (*a == 42))))
    && Rel(y,((v==0) ? True : (acc(b) && (*b == 7))))))
post ((Init(x) && Init(y))
    && (Rel(x,((v==0) ? True : (acc(a) && (*a == 42))))
    && Rel(y,((v==0) ? True : (acc(b) && (*b == 7))))))
{
[a]_NA := 42;
[b]_NA := 7;
FenceRel(((acc(a) && (*a == 42)) && (acc(b) && (*b == 7))));
[x]_Rlx := 1;
[y]_Rlx := 1;
}
```

FSL Figure 2 Variant

```
method outerscope()
pre True
post ((acc(a) && (*a == 43)) && (acc(b) && (*b == 8)))
{
a := alloc();
b := alloc();
x := alloc((((v==0) ? True : (acc(a) && (*a == 42)))
        && ((v==0) ? True : (acc(b) && (*b == 7)))))_AR;
[x]_Rlx := 0;
t1 := fork(left);
t2 := fork(right);
t3 := fork(middle);
var tmp;
tmp := join(t1);
tmp := join(t2);
tmp := join(t3);
}

method left()
pre (Acq(x,((v==0) ? True : (acc(a) && (*a == 42)))) && Init(x))
post (acc(a) && (*a == 43))
{
var tmp;
while(tmp := [x]_Rlx == 0){skip};
FenceAcq;
[a]_NA := (tmp := [a]_NA + 1);
}

method right()
pre (Acq(x,((v==0) ? True : (acc(b) && (*b == 7)))) && Init(x))
post (acc(b) && (*b == 8))
{
var tmp;
while(tmp := [x]_Rlx == 0){skip};
FenceAcq;
[b]_NA := (tmp := [b]_NA + 1);
}

method middle()
pre ((Uninit(a) && Uninit(b))
    && Rel(x,(((v==0) ? True : (acc(a) && (*a == 42)))
        && ((v==0) ? True : (acc(b) && (*b == 7))))))
post (Init(x)
    && Rel(x,(((v==0) ? True : (acc(a) && (*a == 42)))
        && ((v==0) ? True : (acc(b) && (*b == 7))))))
{
[a]_NA := 42;
[b]_NA := 7;
FenceRel(((acc(a) && (*a == 42)) && (acc(b) && (*b == 7))));
[x]_Rlx := 1;
}
```

Nonatomic Sequential Simple Example

```
method foo()
pre True
post ((acc(al) && (*al == 5)) && (Result == 9))
{
var lo;
lo := 5;
al := alloc();
[al]_NA := lo;
lo := ((1 == 1) ? 6 : 7);
lo := ((1 != 1) ? 8 : 9);
Result := lo;
}
method bar()
pre True
post True
{skip}
```

Release Write on Non-Atomic

```
method foo() pre True post True
{
var lo;
lo := 5;
al := alloc();
[al]_Rel := lo;
lo := ((1 == 1) ? 6 : 7);
lo := ((1 != 1) ? 8 : 9);
}
method bar()
pre True
post True
{skip}
```

RSL Figure 7

```
method new_lock()
pre acc(z)
post ((Rel(x,((v == 0) ? True : ((v == 1) ? acc(z) : False)))
    && RMWAcq(x,((v == 0) ? True : ((v == 1) ? acc(z) : False))))
    && Init(x))
{
x := alloc(((v == 0) ? True : ((v == 1) ? acc(z) : False)))_RMW;
[x]_Rel := 1;
}

method unlock()
pre (acc(z)
    && ((Rel(x,((v == 0) ? True : ((v == 1) ? acc(z) : False)))
    && RMWAcq(x,((v == 0) ? True : ((v == 1) ? acc(z) : False))))
    && Init(x)))
post ((Rel(x,((v == 0) ? True : ((v == 1) ? acc(z) : False)))
    && RMWAcq(x,((v == 0) ? True : ((v == 1) ? acc(z) : False))))
    && Init(x))
{
[x]_Rel := 1;
}

method lock()
pre ((Rel(x,((v == 0) ? True : ((v == 1) ? acc(z) : False)))
    && RMWAcq(x,((v == 0) ? True : ((v == 1) ? acc(z) : False))))
    && Init(x))
post (acc(z)
    && ((Rel(x,((v == 0) ? True : ((v == 1) ? acc(z) : False)))
    && RMWAcq(x,((v == 0) ? True : ((v == 1) ? acc(z) : False))))
    && Init(x)))
{
var tmp;
while(tmp := CAS(x,1,0)_AcqRlx != 1){skip};
}

method allocationOfNonAtomic()
pre True
post acc(z)
{
z := alloc();
[z]_NA := 42;
}
```

RSL Figure 7 non-relaxed

```
method new_lock()
pre acc(z)
post ((Rel(x,((v == 0) ? True : ((v == 1) ? acc(z) : False)))
    && RMWAcq(x,((v == 0) ? True : ((v == 1) ? acc(z) : False))))
    && Init(x))
{
x := alloc(((v == 0) ? True : ((v == 1) ? acc(z) : False)))_RMW;
[x]_Rel := 1;
}

method unlock()
pre (acc(z)
    && ((Rel(x,((v == 0) ? True : ((v == 1) ? acc(z) : False)))
    && RMWAcq(x,((v == 0) ? True : ((v == 1) ? acc(z) : False))))
    && Init(x)))
post ((Rel(x,((v == 0) ? True : ((v == 1) ? acc(z) : False)))
    && RMWAcq(x,((v == 0) ? True : ((v == 1) ? acc(z) : False))))
    && Init(x))
{
[x]_Rel := 1;
}

method lock()
pre ((Rel(x,((v == 0) ? True : ((v == 1) ? acc(z) : False)))
    && RMWAcq(x,((v == 0) ? True : ((v == 1) ? acc(z) : False))))
    && Init(x))
post (acc(z)
    && ((Rel(x,((v == 0) ? True : ((v == 1) ? acc(z) : False)))
    && RMWAcq(x,((v == 0) ? True : ((v == 1) ? acc(z) : False))))
    && Init(x)))
{
var tmp;
while(tmp := CAS(x,1,0)_AcqRel != 1){skip};
}

method allocationOfNonAtomic()
pre True
post acc(z)
{
z := alloc();
[z]_NA := 42;
}
```

RSL Figure 8

```
method outerscope()
pre True
post (acc(a) && (*a == 8))
{
a := alloc();
c := alloc(((v == 0)? True : (acc(a) && (*a == 7))))_AR;
[c]_Rlx := 0;
t1 := fork(thread1);
t2 := fork(thread2);
var tmp;
tmp := join(t1);
tmp := join(t2);
}

method thread1()
pre (Uninit(a) && Rel(c,((v == 0)? True : (acc(a) && (*a == 7)))))
post Rel(c,((v == 0)? True : (acc(a) && (*a == 7))))
{
[a]_NA := 7;
[c]_Rel := 1;
}

method thread2()
pre (Acq(c,((v == 0)? True : (acc(a) && (*a == 7)))) && Init(c))
post (acc(a) && (*a == 8))
{
var tmp;
while(tmp := [c]_Acq == 0){skip};
var b;
b := [a]_NA;
[a]_NA := (b + 1);
}
```

Signal two threads with one location

```
method outerscope()
pre True
post ((acc(a) && (*a == 43)) && (acc(b) && (*b == 8)))
{
a := alloc();
b := alloc();
x := alloc((((v==0) ? True : (acc(a) && (*a == 42)))
        && ((v==0) ? True : (acc(b) && (*b == 7)))))_AR;
[x]_Rel := 0;
t1 := fork(left);
t2 := fork(right);
t3 := fork(middle);
var tmp;
tmp := join(t1);
tmp := join(t2);
tmp := join(t3);
}

method left()
pre (Acq(x,((v==0) ? True : (acc(a) && (*a == 42)))) && Init(x))
post (acc(a) && (*a == 43))
{
var tmp;
while(tmp := [x]_Acq == 0){skip};
[a]_NA := (tmp := [a]_NA + 1);
}

method right()
pre (Acq(x,((v==0) ? True : (acc(b) && (*b == 7)))) && Init(x))
post (acc(b) && (*b == 8))
{
var tmp;
while(tmp := [x]_Acq == 0){skip};
[b]_NA := (tmp := [b]_NA + 1);
}

method middle()
pre ((Uninit(a) && Uninit(b))
    && Rel(x,(((v==0) ? True : (acc(a) && (*a == 42)))
        && ((v==0) ? True : (acc(b) && (*b == 7))))))
post (Init(x)
    &&  Rel(x,(((v==0) ? True : (acc(a) && (*a == 42)))
        && ((v==0) ? True : (acc(b) && (*b == 7))))))
{
[a]_NA := 42;
[b]_NA := 7;
[x]_Rel := 1;
}
```

Undeclared Local

```
method foo()
pre True
post True
{
var lo;
lo := 5;
al := alloc();
[al]_NA := lo;
lo := ((1 == 1) ? 6 : 7);
lo := ((1 != 1) ? 8 : 9);
}
method bar()
pre True
post True
{
readvar := [al]_NA
}
```

B. Bibliography

[1] Viktor Vafeiadis and Chinmay Narayan. Relaxed separation logic: A program logic for c11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 867–884, New York, NY, USA, 2013. ACM.

[2] Marko Doko and Viktor Vafeiadis. A program logic for C11 memory fences. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, VMCAI 2016, pages 413–430, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[3] Alexander J. Summers and Peter Müller. Automating deductive verification for weak-memory programs. Technical report, 2017. To appear on `arxiv.org`. Draft available from `http://people.inf.ethz.ch/summersa/wiki/doku.php?id=research`.

[4] Peter W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, April 2007.

[5] Marko Doko and Viktor Vafeiadis. Tackling real-life relaxed concurrency with FSL++. *European Symposium on Programming (ESOP) 2017.* To appear. Draft available from `http://plv.mpi-sws.org/fsl/`.

[6] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, VMCAI 2016, pages 41–62, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[7] `https://github.com/scala/scala-parser-combinators`.

[8] `http://www.codecommit.com/blog/scala/the-magic-behind-parser-combinators`.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

A Prototype Verifier for Weak Memory Reasoning

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Golz	Christiane

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Waldshut, 20.03.2017	C. Golz

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.