Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2015,3
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Deductive Verification of Concurrent Programs

Daniel Bruns

2015

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Fakultät für **Informatik**

Please note:
This Report has been published on the Internet under the following Creative Commons License:
http://creativecommons.org/licenses/by-nc-nd/3.0/de.

Abstract

Verification of concurrent programs still poses one of the major challenges in computer science. Several techniques to tackle this problem have been proposed. However, they often do not scale. We present an adaptation of the rely/guarantee methodology in dynamic logic. Rely/guarantee uses functional specification to symbolically describe the behavior of concurrently running threads: while each thread *guarantees* adherence to a specified property at any point in time, all other threads can *rely* on this property being established. This allows to regard threads largely in isolation—only w.r.t. an environment constrained by these specifications. While rely/guarantee based approaches often suffer from a considerable specification overhead, we complement functional thread specifications with frame conditions.

We will explain our approach using a simple, but concurrent programing language. Besides the usual constructs for sequential programs, it caters for dynamic thread creation. We define semantics of concurrent programs w.r.t. an underspecified deterministic scheduling function.

To formally reason about programs of this language, we introduce a novel multi-modal logic, *Concurrent Dynamic Trace Logic* (CDTL). It combines the strengthes of dynamic logic with those of linear temporal logic and allows to express temporal properties about symbolic program traces. We first develop a sound and complete sequent calculus for the logic subset that uses the sequential part of the language, based on symbolic execution. In a second step, we extend this to a calculus for the complete logic by adding symbolic execution rules for concurrent interleavings and dynamic thread creation based on the rely/guarantee methodology. Again, this calculus is proven sound and complete.

Acknowledgement. The author acknowledges financial support by the German National Science Foundation (DFG) under project "Program-level Specification and Deductive Verification of Security Properties (DeduSec)"[1] within priority programme 1496 "Reliable Secure Software Systems (RS^3)."[2]

[1] `http://www.key-project.org/DeduSec/`
[2] `http://www.spp-rs3.de/`

Contents

Page

1 Concurrent Programs **1**
- 1.1 Sequential and Concurrent Programs . . . 2
- 1.2 Approach Overview . . . 3
- 1.3 Scheduler Assumptions . . . 4
- 1.4 Target Programing Language . . . 6
- 1.5 Representing Memory and Threads . . . 8
- 1.6 Trace Semantics for Sequential Programs . . . 10
- 1.7 Semantics of Concurrent Programs . . . 13

2 Concurrent Dynamic Trace Logic **17**
- 2.1 Syntax of Concurrent Dynamic Trace Logic . . . 18
- 2.2 Semantics of Concurrent DTL . . . 20
- 2.3 Discussion . . . 24

3 Deductive Verification of Concurrent Programs **27**
- 3.1 Concurrent Verification . . . 27
- 3.2 Rely/Guarantee Reasoning . . . 29
  - 3.2.1 Relevant Interleavings . . . 30
- 3.3 Proof Obligations . . . 31
- 3.4 A Calculus for Concurrent DTL . . . 36
  - 3.4.1 Reasoning About Environment Steps . . . 36
  - 3.4.2 Reasoning About Thread Creation . . . 37
  - 3.4.3 Soundness . . . 39
  - 3.4.4 Completeness . . . 40

4 Related Work **41**
- 4.1 Temporal Behavior of Java Programs . . . 41
- 4.2 Deductive Reasoning About Concurrent Programs . . . 42
- 4.3 Rely/Guarantee . . . 43

Bibliography **45**

Chapter 1

Concurrent Programs

Today, in the 21st century, software systems are behind almost any process of everyday life—business or private. With the ever growing amount of sensitive data they handle, and the connectedness of the world, we are in need of precise and enforcible security mechanisms. The recent discoveries of security threats induced by badly designed software, such as the Heartbleed bug, have demonstrated the demand for a rigorous security assessment in software development. This does not only concern domains with traditionally high assurance demands, such as banking or aeronautics, but also private communication.

At the same time, software systems have become more and more complex. Established concepts for modularization or information hiding, such as object orientation, facilitate the designers'/implementors' work; but raise the complexity for analyzing such a system. Parallelization has become a major paradigm in computer development. And many software faults are linked to an inferior understanding of concurrent program semantics.

In this chapter, we introduce our target concurrent language. Besides the usual constructs for sequential execution, it features a `fork` statement to spawn a fresh thread. We will develop denotational semantics for sequential programs and subsequently for concurrent programs. Concurrent programs consist of a set of threads, that each executes a sequential program that is interleaved by the environment. Concurrent changes to the shared memory are modeled explicitly in the program code through explicit release points and an explicit—yet underspecified—scheduling function.

The semantics defined in this chapter is agnostic concerning analysis techniques. This will serve as the foundation for the dynamic logic to be introduced in Chap. 2 and for Chap. 3 on modular reasoning about concurrent programs using rely/guarantee. We start by reviewing the established concepts of concurrency and by explaining the foundations of our approach.

1.1 Sequential and Concurrent Programs

Concurrent computer systems have existed for a long time. But it is only recently (i.e., since the late 2010s) that many end-user systems are concurrent. Precise semantics-based analysis is justified. We have powerful state of the art techniques for specification and verification of sequential programs. These are mature enough to be lifted to the concurrent setting.

Sequential programs run on single processors. Concurrent programs in general can be modelled as a collection of processes—that may each be described like sequential programs—some (implicit or explicit) channels. The central advantage of concurrent programs is that they can be actually executed in parallel on physically separate processor. This concurrency model is usually found in distributed systems, where each processor maintains its own memory and cannot interfere directly with the other processes. Instead, they communicate asynchronously through message passing using explicit channels.

Time share parallelism, on the other hand side, runs on one (or more) shared processors, with the next to-be-dispatched process to be determined by a scheduler. Processes may be *preempted* in order for other processes to be executed in between. In most systems this interleaving can occur at any point in time during the execution. In some systems the program states in which interleaving is possible are restricted. An example is the coöperative scheduling paradigm using explicit release points [Dovland et al., 2005] in the Abstract Behavioral Specification (ABS) language [Johnsen et al., 2010].

Most time share systems also share memory. This means that the functional behavior of one process may be influenced by another one (and the scheduler itself): one process writes a location that another one reads. These kind of processes cannot be assessed independently. While modern desktop computers have multiple processor cores, these are not (purely) distributed, but form a time share system that particularly shares the main memory.[3] Processes that share processors and memory are also called threads, designating this kind of concurrency as multi-threading. This thesis is dedicated to the multi-threading paradigm as it is used in the Java language, amongst others.

All kinds of programs can be described by their observable behavior. For sequential programs it suffices to describe the relation between initial and terminal system states (or between a multitude of possible states). Even for sequential programs, the exact definitions of observable behavior widely diverge, e.g., regarding termination, exceptions, heap structures, or side effects, etc. Modules of sequential programs are (public) procedures. The techniques for modular reasoning about sequential programs are design by contract, behavioral subtyping, etc.

[3]Each processor may have private memory in caches, etc.

Concurrent programs may allow more observations. "The key to formulating compositional proof methods for concurrent processes is the realisation that one has to specify not only their initial-final state behaviour, but also their interaction at intermediate points." [de Roever et al., 2001].

Processes in shared memory systems tend to interfere with each other. While some interactions are certainly benign—otherwise there would be little benefit in concurrency—their scope cannot be restricted in general. Thus, the goal is to harness concurrent modifications. Specifications help to describe functional behavior. But we also need to ensure that "nothing else changes." In sequential programs, this *frame problem* [Borgida et al., 1993] is well known.

Computing all possible interleavings for concurrent systems is far from being feasible. This is known as the *global method* [de Roever et al., 2001]. Its complexity is clearly exponential in the number of concurrent processes (with the number of local descision points in each process being the base of that power). The technique by Owicki and Gries [1976] was the first to be symbolic and has only linear complexity in the number of processes. It is based on traditional local correctness proofs in Hoare logic plus additional noninterference proofs. The rely/guarantee technique [Jones, 1983; Xu et al., 1997] completely relies on modular specification and is compositional. *Assumption/commitment* [Misra and Chandy, 1981] is a similar technique for distributed systems with message passing.

1.2 Approach Overview

In this thesis we consider a simple concurrent imperative language, that we call *deterministic While-Release-Fork* (dWRF). It extends the sequential language presented by Beckert and Bruns [2013] with interleavings and dynamic thread creation. It is 'Java-like' in the sense that it uses both local and global variables (aka. fields) and that an arbitrary number of sequential program fragments[4] can be executed concurrently. dWRF distinguishes between local variables with atomic assignments and global variables with assignments inducing (local) state transitions. The rationale behind this is that, in a concurrent setting, only global memory can be observed by the environment. Expressions do not have side effects. New threads can be spawned in a simple `fork` statement, that names the thread to create, but does not have parameters. Synchronization is not considered at the moment and will be left to future work. We introduce the syntax of dWRF in Sect. 1.4.

[4]Throughout this thesis, we will use the term 'program' for sequential program fragments (or, 'blocks' in Java). This is the usual notion of programs in the context of dynamic logics. We will sometimes use 'system of (sequential) programs' to denote entities that are considered 'programs' in other contexts. Since we do not introduce method calls in our simple language, this distinction is not essential.

Other Java features such as objects, arrays, types, or exceptions are not of relevance to our discourse. These are largely orthogonal (cf., e.g., [Stärk et al., 2001]) and could be added without invalidating the central results. All such features can be added in principle, but we keep the programing language simple for the presentation in this chapter.

We conjecture that write actions are immediately visible to the environment.[5] On the other hand side, concurrent changes induced by the environment only appear in the semantics when they actually may have an effect, namely upon read actions or termination. In order to extend the language defined by Beckert and Bruns [2013] in a *conservative* manner, we do not alter the semantics of read actions. Instead, we introduce explicit release points [Dovland et al., 2005]. Release points denote that a thread voluntarily releases control and the scheduler may select another thread. We represent this through explicit `release` statements, which semantics is defined through the local semantics of the environment threads. All other program statements are not affected by the environment. While in reality, interleavings may occur at any point in time, this setup is sufficient to model such systems, while it greatly reduces the number of program states in which we must expect interleavings.

For this chapter, we just assume that `release` statements may appear in the code. Later, in Sect. 3.2.1, we explain how purely sequential programs can be explicitly instrumented with `release` statements at the relevant interleaving points to model the behavior of actual concurrent programs.

The semantics of dWRF is meant to extend the semantics of the sequential language by Beckert and Bruns [2013] in a conservative way. However, in contrast to Beckert and Bruns [2013], we model global memory using an explicit (ghost) program variable `heap`, as explained in Sect. 1.5. The semantics of `heap` is a mapping from global variable names to values. This modeling caters both for abstract anonymization (i.e., havocking) on (possibly underspecified) parts of the heap and for a convenient comparisons of the entire memory, that we need for the techniques presented in Chaps. 3f. Program semantics with explicit heap representations have been used in [Weiß, 2011], for instance. We extend Weiß' approach with a second variable `heap'` to denote the heap in the previous state, that we use to represent two-state invariants in the rely/guarantee approach in Chap. 3.

1.3 Scheduler Assumptions

Our approach is widely scheduler agnostic. Validity of program properties will be defined in Sect. 1.6 w.r.t. (almost) *any* scheduler; we only make the following fundamental assumptions. A formalization of these properties will

[5]Unfortunately, the Java memory model (JMM) does not guarantee this property.

appear in Defs. 1.10 and 1.15 on pages 14ff. A general framework to formalize scheduler policies is not part of this work.

1. The number of active threads is always finite.

2. In any state in which at least one thread is active, i.e., a thread that is not yet terminated,[6] the scheduler selects an active thread. Without loss of generality, we assume that there is always an active thread, at least a synthetic 'idle' thread that infinitely loops with ineffective global writes.

3. The scheduling itself, i.e., selecting an active thread, does not change the global heap state. This means that a schedule is just a function on states, and thus deterministic. This should not impose a loss of generality as any set of nondeterministic schedulers can be simulated through a set of deterministic schedulers.[7] There already are formalizations of concurrent program semantics using deterministic schedulers in the literature, e.g., by Beckert and Klebanov [2013]. In fact, indeterminism does not offer more expressiveness as we only consider properties that are valid for *any* deterministic scheduler and any indeterministic scheduler can be simulated by a set of deterministic schedulers.

4. The scheduler is fair. By 'fair' we understand the property that every thread will be chosen sufficiently often to terminate—or infinitely often. Given assumption 1, an equivalent phrasing is that any thread is selected at least once within finite time. This does not seem to ban us from modeling real world schedulers. As mentioned by Beckert and Klebanov, Java schedulers are "statistically fair," which means effectively fair in almost any practical situation. From a theoretical point, this assumption makes validity definitions simpler and more consistent. Taking the possibility into account that an interleaving may not return, would effectively introduce a kind of indeterminism.[8]

For information flow, it will be interesting whether the scheduler can work on high data. We assume an attacker model where the attacker is

[6] Thread suspension is not yet considered.

[7] There may be a special thread that does the actual calculations for a schedule and changes the state accordingly. In particular, we can assume that the exact same program state cannot be reached twice.

[8] Dropping fairness would require to relax the semantics of properties on programs, leading only to partial correctness. An approach would be to introduce a special program 'state' that is unreachable and in which any formula is vacuously true. Such a definition would be very disturbing to our logic as there cannot be a regular program state with this properties. It would have to be treated explicitly in every definition. The logic of Beckert and Bruns [2013] is particularly well-behaved because of the (one) modality being dual to itself (i.e., it is invariant under negation). Such a property would be lost.

in control of threads, but not the scheduler. This means that an attacker cannot distinguish why/in which state its threads are scheduled or not—even in case the scheduler schedules using confidential information.

1.4 Target Programing Language

In this section, we introduce our target programing language *deterministic While-Release-Fork* (dWRF).[9] The sequential language constructs are assignments, conditional branching, and conditional loop statements. Additional constructs for concurrency are forking and release statements. Programs are sequences of statements. The (mathematical) integers and boolean are the only data type for program variables. Expressions can be of types integer or boolean; they do not have side effects. Integer operators are unary minus, addition, multiplication, division and modulo. The program language does not contain features such as functions and arrays; and there are no object-oriented features. The only special feature is the distinction between local variables (written in lowercase letters) and global variables (written in uppercase). We assume that local variable names are unique; in particular, there are no name clashes between threads.

Program expressions are typed. We use pairwise disjoint types $\mathbb{Z}$ (integers) and $\mathbb{B}$ (boolean). We assume disjoint sets *LVar* of local program variables and *GVar* of global program variables to be given.

Definition 1.1 (Program expressions)**.** Program expressions of type $\mathbb{Z}$ are constructed as usual over integer literals, local and global variables, and the operators `+`, `-`, `*`, `/`, and `%`. Program expressions of type $\mathbb{B}$ are constructed using the relations `==`, `>`, and `<` on integer expressions, the boolean literals `true` and `false`, and the logical operators `&&`, `||`, and `!`. A program expression is *simple* if it does not contain global variables.

As will be explained in Sect. 1.6, we consider assignments to global variables to be the only program statements that lead to a new observable state. To ensure that there cannot be a program that gets stuck in an infinite loop without ever progressing to a new observable state, we demand that every loop contains an assignment to a global variable.[10] Expressions on the right hand side of global assignments and conditions for `if` or `while` statements must be simple. The right hand side of local assignments may refer to at most one global variable.

We extend the core language introduced by Beckert and Bruns [2013] with two additional statements `release` and `fork` that represent explicit thread release and thread creation, respectively. By instrumenting a sequential

[9] It is pronounced [dwɔːɹf].

[10] This technical restriction can easily be fulfilled by adding ineffective assignments.

Table 1.1: Syntax of sequential dWRF programs. Local and global program variables are represented by rules `v` and `G`, respectively.

$$
\begin{array}{lcl}
z & ::= & z\texttt{+}z \mid z\texttt{-}z \mid z\texttt{*}z \mid z\texttt{/}z \mid z\texttt{\%}z \mid \texttt{v} \mid \texttt{G} \mid \texttt{0} \mid \texttt{1} \mid \texttt{...} \\
b & ::= & \texttt{true} \mid \texttt{false} \mid b\ \texttt{\&\&}\ b \mid b\ \texttt{||}\ b \mid \texttt{!}b \mid z\ \texttt{==}\ z \mid z\ \texttt{>}\ z \mid \\
 & & z\ \texttt{<}\ z \mid \texttt{v} \mid \texttt{G} \\
x & ::= & z \mid b \\
\pi & ::= & \texttt{G = } x \mid \texttt{v = } x \mid \pi\texttt{;}\pi \mid \texttt{if (}b\texttt{) \{}\pi\texttt{\} else \{}\pi\texttt{\}} \mid \\
 & & \texttt{while (}b\texttt{) \{}\pi\texttt{\}} \mid \texttt{release} \mid \texttt{fork}\ f
\end{array}
$$

program with the `release` statement, we simulate interleavings in a concurrent program, as explained in Sect. 3.2.1.

Definition 1.2 (dWRF syntax)**.** A *statement* is one of the following:

- ***local assignment***: `v =` x`;` where `v` is a local variable and x is an expression of the same type not containing reference to more than one global variable
- ***global assignment***: `F =` x`;` where `F` is a global variable and x is a simple expression of the same type
- ***conditional***: `if (`b`) {`π_0`} else {`π_1`}` where b is a simple boolean expression and π_0, π_1 are programs
- ***loop***: `while (`b`) {`π`}` where b is a simple boolean expression and π is a program containing at least one global assignment
- ***thread release***: `release;`
- ***thread creation***: `fork` f`;` where f is a new thread

Definition 1.3. A *sequential program*, or just 'program' for short, is a finite sequence of statements. The set of sequential programs is denoted by *Prg*. Programs not containing `release` or `fork` are called *noninterleaved*. A *concurrent program* Π is a finite set of sequential programs, $\Pi \in 2^{Prg}_{\text{fin}}$.

Table 1.1 displays the syntax of sequential programs. The language of Beckert and Bruns [2013] is not strictly included in this, as it permits nonsimple expressions to appear as guards or the right hand side of global assignments. Nevertheless, any Beckert and Bruns [2013] program can be transformed into an equivalent program in the intersection by adding local assignments; see Lemma 1.9.

Example 1.4. The following line shows a small (noninterleaved) program, that reads two integers from global variables `A` and `B` and writes the minimum to a third global variable `C`.

```
x = A; y = B; if (x < y) { C = x; } else { C = y; }
```

The following, shorter line is not a valid dWRF program since the statements in the conditional and on the right hand side of the global assignments to `C` are not simple. Yet, both are equivalent in the language presented by Beckert and Bruns [2013].

```
if (A < B) { C = A; } else { C = y; }
```

1.5 Representing Memory and Threads

We now lay the foundations for defining a semantics for dWRF. There are essentially two possibilities of representing computer memory in semantics and logic:[11] 1. to represent each memory location by a function symbol[12] or 2. to use a dedicated theory of storage and update a special variable representing the current memory state.

While Hoare logic and classical dynamic logic [Harel, 1979] pursue the former approach and use function symbols for each memory location,[13] the concept of having just one mathematical object to represent the whole memory of a computer system has been proven to be more convenient in may regards. It does allow to specify dynamically allocated memory—in particular recursively defined data structures—in a modular way; and it allows to specify information flow properties [Scheben and Schmitt, 2012]. Instead of enumerating all the locations that are unchanged, we can just quantify over them. In Chap. 3, we show that with an explicit heap we can express two-state invariants conveniently. Such explicit heap modeling appears in [Poetzsch-Heffter and Müller, 1999; Stenzel, 2005; Barnett et al., 2005; Smans et al., 2008; Leino, 2010; Leino and Rümmer, 2010; Weiß, 2011].

To represent the heap on the semantical level, we introduce special program variables **heap** and **heap'**, that must not appear in programs.[14] Their semantics is a partial function from global variables to values. Upon every state change, induced by a write action, the values of **heap** an **heap'** are updated.

[11] The reader may excuse that this section anticipates logic to some extend, that is meant to appear in Chap. 2. On the other hand side, the logic representation is strongly related to the modeling issues that are discussed here.

[12] Although it may sound confusing, program variables are considered (nonrigid) constants, i.e., 0-ary functions, in this context.

[13] This approach was also taken in earlier versions of the KeY system, see [Beckert, 2001; Beckert et al., 2007b].

[14] They can be described as ghost variables, thus.

Reasoning about heaps is provided through the explicit heap theory of Weiß [2011], that is already implemented in the KeY verification system from version 2.0 onwards. It compromises of the three data types Field (representing the syntactical entity of the same name in program code, denoted by $\mathbb{F}$), LocSet (representing finite sets of locations, denoted by $\mathbb{L}$), and Heap (representing a mapping from the set of all locations to values, denoted by $\mathbb{H}$). For most of this dissertation, we identify the terms 'location' and 'field' with each other, since we do not have the notion of objects.[15]

The field data type contains only a finite number of constants. The signature of the location set data type is the same as for standard (finite) sets; it includes a constructors empty set $\dot\varnothing$ and singleton $\{\ell\}$ (with a location ℓ), the binary set operators $\dot\cup$ (union), $\dot\cap$ (intersection), and $\setminus$ (set minus); as well as the unary set operator $\cdot^{\complement}$ representing the complement in the set of all locations.[16] The predicate $\dot\in$ indicates whether a field is in a location set. For convenience, we write $\{x_0, x_1, \ldots, x_n\}$ as shorthand for $\{x_0\} \mathbin{\dot\cup} \{x_1\} \mathbin{\dot\cup} \ldots \mathbin{\dot\cup} \{x_n\}$.

The heap data type is a coalgebraic data type[17] with a single observer (or, 'destructor') *select* and two elementary mutator functions *store* and *anon* whose semantics are given in terms of *select*s on them, which is based on the theory of arrays by McCarthy [1962]:

(i) $\mathit{select}(h, \ell)$ of type $\top$, where h is term of type $\mathbb{H}$ and ℓ ('location') of type $\mathbb{F}$, representing value retrieval from a location;

(ii) $\mathit{store}(h, \ell, v)$ of type $\mathbb{H}$, where v is a term of any value type (e.g., integer) and h, ℓ as above, representing a state change; and

(iii) $\mathit{anon}(h, L)$ of type $\mathbb{H}$, represents a heap that is havocked on all location in the location set L, but agrees on h otherwise.

We do not give formal semantics for the LocSet and Heap theories here as they should be intuitively clear; the interested reader is pointed to [Weiß, 2011, Chap. 5]. By abuse of notation, we write logic symbols and their semantical counterpart functions alike.

Threads are also represented by semantical objects. The state of currently active threads is recorded in a special variable `threads`. Like `heap`, it must not appear in programs. It is updated whenever a fresh thread is forked. It is of type $\mathbb{T}$, that is to be understood as finite, nonempty sets of threads.

[15] Weiß [2011] defines a location as a pair of a receiver object and a field (which is just an identifier).

[16] The set of all locations is a welldefined finite set in this setting since there are only finitely many field constants. In general, the set of locations may not be finite.

[17] Confer the introduction to coalgebraic data types by Jacobs and Rutten [1997], for instance.

We assume set theoretical operators present, equivalent to the ones for $\mathbb{L}$ introduced above, that we denote with the same symbols. Like for the above theories, we refrain from overloading this section with formal semantics.

1.6 Trace Semantics for Sequential Programs

In this section, we give semantics for noninterleaved sequential dWRF programs. Instead of defining semantics as a relation between initial and final states of an abstract execution of the program (like by Beckert [2001], for instance), we use complete traces of intermediate program states. This will be extended to concurrent program in Sect. 1.7.

Expressions and formulae are evaluated over traces of states (that give meaning to program variables) and variable assignments (that give meaning to logical variables). The *domain*, denoted by $\mathcal{D}$, contains all semantical values to which an expression can evaluate. It does not depend on the program state (*constant domain*). The domain can be partitioned into $\mathcal{D}_T$ for a type T.[18] All theories have the usual semantics. In particular the domain of integer expressions is $\mathbb{Z}$ and the domain of location set expressions consists of sets of locations: $\mathcal{D}_{\mathbb{L}} \subseteq 2^{\mathcal{D}_{\mathbb{F}}}$.

In addition to the sets *LVar* and *GVar*, we introduce the disjoint set of 'special variables' $SVar := \{\mathtt{heap}, \mathtt{heap'}, \mathtt{threads}\}$ that do not appear in programs, but only in semantics.

Definition 1.5 (Program state)**.** A *program state*—or simply *state* for short—is a function $s : LVar \cup SVar \to \mathcal{D}$ assigning values to program variables. It assigns integer or boolean values to all proper local variables of the appropriate type (i.e., $s|_{LVar} : LVar \to \mathcal{D}_{\mathbb{Z}} \cup \mathcal{D}_{\mathbb{B}}$), a heap function to the special variables `heap` and `heap'` (i.e., $s|_{\{\mathtt{heap},\mathtt{heap'}\}} : SVar \to \mathcal{D}_{\mathbb{H}}$), and a set of threads to the special variable `threads` (i.e., $s|_{\{\mathtt{threads}\}} : SVar \to \mathcal{D}_{\mathbb{T}}$).

Instead of the usual mathematical notation $s(x)$ for function application, we will frequently use the notation x^s, that is common in logic texts. We use the notation $s\{x \mapsto d\}$ to denote the state that is identical to s except that the variable x is assigned the value $d \in \mathcal{D}$, formally $s\{x \mapsto d\} = \{x \mapsto d\} \cup \{y \mapsto s(y) \mid y \in LVar \cup SVar \setminus x\}$. Likewise, we write $\tau\{x \mapsto d\}$ (where τ is a trace, see below) with the obvious semantics. For global program variables, the special variable `heap` is updated to a new function using the (higher order) function *store*, see Sect. 1.5.

Definition 1.6 (Traces)**.** A *computation trace*, or just *trace* for short, τ is a non-empty, finite or infinite sequence of (not necessarily different) states. The set of traces is denoted by $\mathcal{S}^*$.

[18] Remember that we do not have subtypes in dWRF.

We use the following notations related to traces:

- $|\tau| \in \mathbb{N} \cup \{\infty\}$ is the *length* of a trace τ. If $\tau = \langle s_0, \dots s_k \rangle$, then $|\tau| = k+1$.
- $\tau_1 \cdot \tau_2$ is the *concatenation* of traces:
  - If $|\tau_1| = \infty$, then $\tau_1 \cdot \tau_2 = \tau_1$.
  - If $\tau_1 = \langle s_0, \dots, s_k \rangle$ (finite) and $\tau_2 = \langle t_0, \dots \rangle$ (possibly infinite), then $\tau_1 \cdot \tau_2 = \langle s_0, \dots, s_k, t_0, \dots \rangle$.
- $\tau[i, j)$ for $i, j \in \mathbb{N} \cup \{\infty\}$ is the *subtrace* beginning in the i-th state (inclusive) and ending before the j-th state:
  - If $i \geq |\tau|$ or $i \geq j$, then $\tau[i, j) = \tau$
  - If $i < |\tau| < j$, then $\tau[i, j) = \tau[i, |\tau|)$
  - If $\tau = \langle s_0, \dots, s_i, s_{i+1}, \dots, s_{j-1}, s_j, \dots \rangle$, then $\tau[i, j) = \langle s_i, s_{i+1}, \dots, s_{j-1} \rangle$ for $j < \infty$ and $\tau[i, \infty) = \langle s_i, s_{i+1}, \dots \rangle$.
- $\tau[i]$ for $i \in \mathbb{N}$ is the state at position i in τ (with $\tau[i] := \tau[0]$ for $i \geq |\tau|$). For convenience, we identify singleton traces with their sole element.

Computation traces of programs are defined through small step denotational semantics on observable states. As mentioned above, we consider assignments to global variables to be the only statements that lead to a new observable state. By specifying which variables are local and which are global, the user can thus determine which states are 'interesting' and are to be included in a trace. For the feasibility of proving properties about dWRF programs, it is important that not too many irrelevant intermediate states are included in a trace.

Definition 1.7 (Trace of a noninterleaved program)**.** Given an initial state s, the trace of a noninterleaved program π, denoted $trc_\Sigma(s, \pi)$, is defined by (the greatest fixpoint of):

$$
\begin{aligned}
trc_\Sigma(s, \epsilon) &:= \langle s \rangle \\
trc_\Sigma(s, \texttt{x=a;}\ \omega) &:= trc_\Sigma(s\{x \mapsto a^s\}, \omega) \\
trc_\Sigma(s, \texttt{X=a;}\ \omega) &:= \langle s \rangle \cdot trc_\Sigma\left(s\left\{\begin{array}{l}\texttt{heap'} \mapsto \texttt{heap}^s, \\ \texttt{heap} \mapsto \texttt{heap}^s\{\texttt{X} \mapsto a^s\}\end{array}\right\}, \omega\right) \\
trc_\Sigma\left(s, \begin{array}{l}\texttt{if (a) \{}\pi_1\texttt{\}} \\ \texttt{else \{}\pi_2\texttt{\}}\ \omega\end{array}\right) &:= \begin{cases} trc_\Sigma(s, \pi_1\ \omega) & \text{if } s \models a \\ trc_\Sigma(s, \pi_2\ \omega) & \text{if } s \not\models a \end{cases} \\
trc_\Sigma(s, \texttt{while (a) \{}\pi\texttt{\}}\ \omega) &:= \begin{cases} trc_\Sigma(s, \pi\ \texttt{while (a) \{}\pi\texttt{\}}\ \omega) & \text{if } s \models a \\ trc_\Sigma(s, \omega) & \text{if } s \not\models a \end{cases}
\end{aligned}
$$

where ϵ is the empty program and ω is a program.

```
                    heap ↦ {A ↦ 5, B ↦ 7, C ↦ −1}                              (0)
x = A;
                    heap ↦ {A ↦ 5, B ↦ 7, C ↦ −1}, x ↦ 5                       (1)
y = B;
                    heap ↦ {A ↦ 5, B ↦ 7, C ↦ −1}, x ↦ 5, y ↦ 7                (2)
if (x < y) {
                    heap ↦ {A ↦ 5, B ↦ 7, C ↦ −1}, x ↦ 5, y ↦ 7                (3)
  C = x;
                    heap ↦ {A ↦ 5, B ↦ 7, C ↦ 5}, x ↦ 5, y ↦ 7                 (4)
} else {
  C = y; }
                    heap ↦ {A ↦ 5, B ↦ 7, C ↦ 5}, x ↦ 5, y ↦ 7                 (5)
```

Table 1.2: The intermediate states of a program execution are shown on the right. The states shown in red are included in the program trace.

The scheduling function Σ (see Sect. 1.7 below) does not have an effect on this definition. We will omit Σ whenever it is not relevant to the context.

Remark. Typically, program semantics are defined inductively in terms of (sets of) reachable terminal states (i.e., big step semantics), cf. Beckert et al. [2007b, Sect. 3.3]. Opposed to this, our definition is *coinductive*. In it is based on traces of all reachable states (i.e., small step semantics), which may be infinite. This is why the semantics are defined through the *greatest* fixpoint of trc_Σ instead of the least fixpoint.

Example 1.8. We look again on the program from Ex. 1.4, that reads integers from two global variables and writes the minimum to another global variable:

```
x = A; y = B; if (x < y) { C = x; } else { C = y; }
```

Let s be a state with $\mathtt{heap}^s = \{\mathtt{A} \mapsto 5, \mathtt{B} \mapsto 7, \mathtt{C} \mapsto -1\}$. Table 1.2 shows the concrete intermediate states that the execution passes through when started in s. Not all these states are included in the trace of the program, but only the two states before the global assignment (3) and the terminal state (5). All other states are equivalent to one of them regarding the value of `heap`.

The following lemma states that the syntactical restrictions regarding global variables imposed on programs (see Def. 1.2) do not lessen expressivity.

Lemma 1.9. *Let π be a noninterleaved 'program' with the restrictions on global variables waved, i.e., a program of Beckert and Bruns [2013]. There is a proper dWRF program π' that is equivalent to π, i.e., $trc_\Sigma(s, \pi) = trc_\Sigma(s, \pi')$ for all $s \in \mathcal{S}$.*

Proof. We show the lemma by structural induction over π. The base case is the empty program. For the step case, assume that for any proper subprogram π_i of π, there is an equivalent dWRF program π_i'. We have to distinguish between the different kinds of statements:

- $\pi = \texttt{v = } x\texttt{; } \pi_2$: Let x be representable as a function $f_x(\texttt{G}_1, \ldots, \texttt{G}_n)$ where $\texttt{G}_j$ are the global variables in x for some $n \in \mathbb{N}$. Let $\tilde{\pi}_x :=$ $\texttt{v}_1\texttt{ = G}_1\texttt{; } \ldots \texttt{ v}_n\texttt{ = G}_n\texttt{;}$ where the $\texttt{v}_j \in \mathit{LVar}$ are fresh and $\pi' := \tilde{\pi}_x \texttt{ v = } f_x(\texttt{v}_1, \ldots, \texttt{v}_n)\texttt{; } \pi_2'$. Let $\tilde{s} := s\{\texttt{v}_j \mapsto \texttt{G}_j^s \mid 0 < j \leq n\}$. Since the $\texttt{v}_j$ are fresh, for all expressions y that do not contain any $\texttt{v}_j$, it is $y^s = y^{\tilde{s}}$. It is obvious to see that it follows from the definition of trc_Σ for local assignments that the traces of π and π' are the same.
- $\pi = \texttt{F = } x\texttt{; } \pi_2$: as above.
- $\pi = \texttt{if (}b\texttt{) \{}\pi_0\texttt{\} else \{}\pi_1\texttt{\} } \pi_2$: Let b be representable as a function $f_b(\texttt{G}_1, \ldots, \texttt{G}_n)$. Then $\pi' := \tilde{\pi}_b \texttt{ if(}f_b(\texttt{v}_1, \ldots, \texttt{v}_n)\texttt{)\{ }\pi_0'\texttt{\} else \{}\pi_1'\texttt{\} }\pi_2'$. Again, the trace equality is obvious.
- $\pi = \texttt{while (}b\texttt{) \{}\pi_0\texttt{\} } \pi_2$: Let everything be as above. Then $\pi' :=$ $\tilde{\pi}_b \texttt{ while(}f_b(\texttt{v}_1, \ldots, \texttt{v}_n)\texttt{)\{ }\pi_0' \ \tilde{\pi}_b\texttt{\} }\pi_2'$. We prove the trace equality; it is $\mathit{trc}_\Sigma(s, \pi') = \mathit{trc}_\Sigma(\tilde{s}, \texttt{while(}f_b(\ldots)\texttt{) } \ldots)$ and $b^s = (f_b(\vec{\texttt{v}}))^{\tilde{s}}$. If $s \models b$, then $\mathit{trc}_\Sigma(s, \pi') = \mathit{trc}_\Sigma(\tilde{s}, \pi_2')$ and we are done. If $s \not\models b$, then $\mathit{trc}_\Sigma(s, \pi') = \mathit{trc}_\Sigma(\tilde{s}, \pi_0'\tilde{\pi}_b \texttt{ while } \ldots)$. Assume $\mathit{trc}_\Sigma(\tilde{s}, \pi_0') = \mathit{trc}_\Sigma(s, \pi_0')$ is finite with final state $\bar{s}$. Then $\mathit{trc}_\Sigma(s, \pi') = \mathit{trc}_\Sigma(\bar{s}, \tilde{\pi}_b \texttt{ while } \ldots) =$ $\mathit{trc}_\Sigma(\tilde{\bar{s}}, \texttt{while } \ldots)$ and the fixpoint theorem closes the proof. ◁

1.7 Semantics of Concurrent Programs

Above in Def. 1.7, we have given a semantics for the purely sequential part of dWRF, i.e., for noninterleaved programs. In this section, we define semantics for interleaved programs and in turn for concurrent dWRF programs. Remember that interleavings are made explicit in the program code.

At runtime, a concurrent program Π is identified with a set $\mathcal{T}$ of *threads* that can be created and a (fair) *scheduling function* Σ. Every thread $t \in \mathcal{T}$ has an associated sequential program π_t, which syntactically is one of the members of the concurrent program Π, modulo renaming of local variables. Without loss of generality, we assume that local variables are unique to one thread. This means that no two threads have the same program. It is reasonable that $\mathcal{T}$ contains an infinite number of isomorphic copies of each sequential programs as a reservoir. We can think of the syntactical appearance of sequential programs as templates for these copies. Keeping a reservoir from which fresh objects can be selected, instead of actually creating new ones, is a common modeling technique in the context of dynamic logic (cf. [Beckert et al., 2007b, Sect. 3.6.6]). We refer to the pair $(\mathcal{T}, \Sigma)$ as a *concurrent system*.

Besides the memory state as introduced above, concurrent programs also have a *thread state*. We will refer to the set $T \subseteq \mathcal{T} \cap \mathcal{D}_\mathbb{T}$ of currently alive threads as the *thread pool*. Through dynamic thread creation, the thread pool may change throughout program execution. In any reachable program

state, a thread pool is finite and nonempty. Syntactically, we represent the thread pool by the special variable $\texttt{threads} \in SVar$.

Following our assumptions in Sect. 1.3, the scheduler Σ is a mathematical function—deterministic and without side effects. Depending on the state, it chooses a thread from the thread pool. Without loss of generality, we assume that any run of a concurrent program never reaches the exact same state s twice, except for the special case that all threads in $\texttt{threads}^s$ have terminated.[19] The axiom of choice [Zermelo, 1904] guarantees that schedulers do actually exist.

Definition 1.10. A *scheduler* is a function $\Sigma : \mathcal{S} \to \mathcal{T}$ such that $\Sigma(s) \in \texttt{threads}^s$ for any state s.

To define the big step semantics of concurrent programs, we need to define a total state transition function σ, that also takes into account dynamic thread creation. The thread-local transition function σ_t is equivalent to the computation trace of the noninterleaved program π_t. We assume one definite trace here since a sequential program is deterministic. We first extend our definition of program traces for noninterleaved programs in Def. 1.7 to sequential programs that contain `fork` statements, but not `release`.

Definition 1.11 (Computation trace of a forking program; extends Def. 1.7)**.** Let everything be as in Def. 1.7.

$$trc_\Sigma(s, \texttt{fork}\ t;\ \omega) := trc_\Sigma(s\{\texttt{threads} \mapsto \texttt{threads}^s \cup t\}, \omega)$$

We will complete the definition of program traces for interleaved programs in Def. 1.16, where we add the definition of trc_Σ applied to a `release` statement, after having developed a semantics for interleavings.

Definition 1.12. Let π_t be an interleaved sequential program. Let π' be the program obtained by removing all `release` statements. Let the trace fo π' be given as $\langle s_0, s_1, \dots \rangle$. The *thread-local state transition* function $\sigma_t : \mathcal{S} \to \mathcal{S}$ maps any nonterminal state s_i to its successor s_{i+1} and a terminal state to itself in a single step.

We now take the environment into consideration; we assume that it is also deterministic (i.e., it contains other deterministic noninterleaved programs that are executed according to a deterministic scheduler).

Definition 1.13.

[19] Typically, there is a thread that manages the schedule and performs the necessary computations. Even though all other threads are not advancing to a different global state, we can assume that this thread will.

1. For a concurrent system $(\mathcal{T}, \Sigma)$ the *system state transition* function $\sigma_\Sigma : \mathcal{S} \to \mathcal{S}$ denotes a single step of the concurrent program, with $\sigma_\Sigma(s) := \sigma_t(s)$ where $t = \Sigma(s)$.
2. The iterated extension of the transition function σ_Σ with $n \in \mathbb{N}$ is defined inductively as $\sigma^0_\Sigma(s) := s$ and $\sigma^{n+1}_\Sigma(s) := \sigma_\Sigma\left(\sigma^n_\Sigma(s)\right)$.
3. The semantical predicate $\Omega_t(s)$ indicates that s is a terminal state for thread t, i.e., s is a fixpoint of σ_t.[20] $\Omega_T(s)$ means s is terminal for all threads in a thread pool T.

The following definitions are only welldefined for fair schedulers. We define fairness as the property that for every point in time, every alive and not yet terminated thread is called within finite time. For nonterminating threads this means being called infinitely often. In case all threads have terminated, any thread may be chosen ad infinitum.

Definition 1.14 (Fairness)**.** A scheduler Σ is *fair* if for every thread pool T and every thread $t \in T$ there is exists an $n \in \mathbb{N}$ such that $\Sigma\left(\sigma^n_\Sigma(s)\right) = t$, for any nonterminal state $s \in \{s' \in \mathcal{S} \mid \neg\Omega_t(s')\}$.

Definition 1.15 (Macro step)**.** Let Σ be a fair scheduler. The state transition function $\sigma^*_\Sigma : \mathcal{S} \times \mathcal{T} \to \mathcal{S}$ describes the *macro step* between two states in which a thread $t \in \mathtt{threads}^s$ is active or all threads have terminated: $\sigma^*_\Sigma(s,t) := \sigma^n_\Sigma(s)$ where n is the smallest natural number such that $\Sigma\left(\sigma^n_\Sigma(s)\right) = t$, or $\Omega_T(\sigma^n_\Sigma(s))$ with $T = \mathtt{threads}^{\sigma^n_\Sigma(s)}$.

The state transition function $\sigma^*_\Sigma(s,t)$ describes the state change that occurs in between atomic steps of a thread t under investigation. Within the macro step σ^*_Σ, there is no t-transition. But in the final state of $\sigma^*_\Sigma(s,t)$, the scheduler selects t again, as displayed in the example in Fig. 1.3 on the following page. The fairness assumption guarantees that this minimum actually exists. In the special case that all threads have terminated, the scheduler may select any thread, but the transition is defined as the identity function in any case. Note that we do not need to specify program pointers/ active statements in the other threads; this is already encoded in the program state s.

Following these definitions, the thread pool T has an influence on the computation trace of a program, and thus on the definition of validity. We extend Defs. 1.7 and 1.11 with the `release` statement, effectively providing a semantics for interleaved sequential programs.

Definition 1.16 (Computation trace of an interleaved program)**.** Let everything be as in Def. 1.11, but let additionally Σ be a fair scheduler. We additionally define

$$\mathit{trc}_\Sigma(s, \mathtt{release;}\ \omega) := \mathit{trc}_\Sigma(\sigma^*_\Sigma(s, \Sigma(s)), \omega) \qquad .$$

[20] This is sufficient for termination since we assume no state to be repeated.

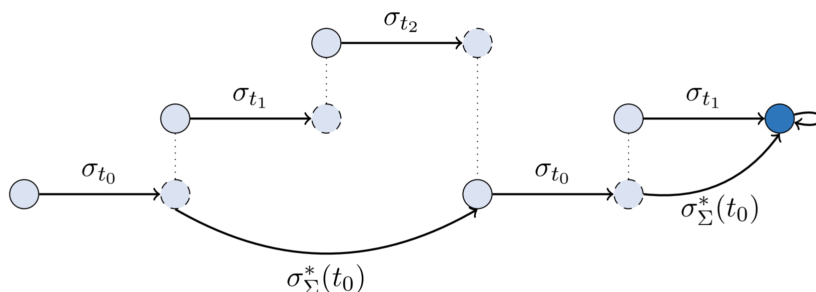

Figure 1.3: A system trace for three threads t_0, t_1, t_2. Two atomic steps σ_{t_1} and σ_{t_2} are combined into a macro step $\sigma^*_\Sigma(t_0)$. Another macro step points to the terminal state represented by the dark node on the right.

This definition of a trace is similar to what Xu et al. [1997] call a "computation," that distinguishes between component and environment state transitions, on the one hand. On the other hand, they model concurrency as indeterminism, allowing any enabled transition to be taken, while our definition is based on deterministic program semantics.

The above definition of program traces establishes that the special variable `heap'` refers to the previous state heap in a non-trivial trace produced by a valid program:[21]

Lemma 1.17. *Let π be a sequential program with a trace $\tau = trc_\Sigma(s, \pi)$ of length $|\tau| \geq 2$. Then for all $i \in (0, |\tau|)$, it is* $\texttt{heap'}^{\tau[i]} = \texttt{heap}^{\tau[i-1]}$.

[21] There are traces that are not induced by a program for which this statement is not valid.

Chapter 2

Concurrent Dynamic Trace Logic

Dynamic logic is an established instrument for program verification and for reasoning about the semantics of programs and programing languages. Most dynamic logics, however, consider only sequential programs. In previous work, [Beckert and Bruns, 2013], we have defined Dynamic Trace Logic (DTL), that combines the expressiveness of program logics such as first order dynamic logic with that of temporal logic.

In contrast to standard dynamic logic, which is entirely state-based, we use a notion of program semantics based on traces of program states. In the previous chapter, we have introduced dWRF, a simple programing language with basic support for multithreading. We have defined a trace-based semantics for dWRF, including an interleaving semantics w.r.t. a deterministic scheduler. In this chapter, we define a dynamic logic for dWRF, extending DTL to Concurrent Dynamic Trace Logic (CDTL). A base calculus for pure DTL has been presented by Beckert and Bruns [2012, 2013]. It will be extended to full CDTL in Chap. 3.

The KeY verification system (co-developed by the author) is built on a calculus for JavaDL, a dynamic logic for sequential Java [Beckert, 2001; Beckert et al., 2007b]. As these features are mostly orthogonal to those discussed in this chapter, the JavaDL calculus has been used as a basis to extend CDTL to Java and implement the CDTL calculus (a prototypical implementation exists). Additional rules needed to handle full (sequential) Java can be derived from the KeY rules for the $[\cdot]$ modality by analogy. Since a language like Java incorporates a lot of features, in particular object-orientation, and various syntactic sugars, the rule set is quite voluminous (c. 1600 rules) in comparison to simple while languages. These special cases can, however, be reduced to a smaller set of base cases. For instance, the

assignment `x=y++` containing a postincrement operator is transformed into two consecutive assignments `x=y` and `y=y+1` during symbolic execution.[22]

2.1 Syntax of Concurrent Dynamic Trace Logic

In this section, we define the syntax of formulae in our target logic, Concurrent Dynamic Trace Logic (CDTL). It is a typed first order dynamic logic with dedicated theories that extends Dynamic Trace Logic (DTL) [Beckert and Bruns, 2012, 2013]. Programs of the *deterministic While-Release-Fork* (dWRF) language, that we introduced in Chap. 1, give rise to modalities in CDTL.

Signatures and Expressions

In addition to program variables (cf. Sect. 1.4), there is a separate set V of logical variables. Logical variables are *rigid*, i.e., they cannot be changed by programs and—in contrast to program variables—are assigned the same value in all states of a program trace.[23] Logical variables must not occur in programs. Quantifiers can only range over logical variables and not over program variables.

Expressions are typed. We use pairwise disjoint types $\mathbb{Z}$ (integers), $\mathbb{B}$ (boolean), $\mathbb{H}$ (heaps), $\mathbb{L}$ (location sets), $\mathbb{F}$ (fields), $\mathbb{T}$ (thread pools), and $\mathbb{S}$ (sequences); cf. Sect. 1.5. There is a common supertype $\top$. Quantified formulae have the shape $\forall x{:}U.\ \varphi$ where U is one of the above types. If U is the supertype $\top$, it is ommited. For an expression x, its type is denoted by $\mathit{type}(x)$.[24] Both local and global program variables always have types $\mathbb{Z}$ or $\mathbb{B}$.

Functions have signatures $A_1 \times \cdots \times A_n \to B$ where all A_i and B are types. A 0-ary function is called a *constant*. *Predicates* have signatures $A_1 \times \cdots \times A_n$, where $n = 0$ is allowed. Both functions and predicates are rigid. The sets of functions and predicates are denoted by $\mathcal{F}$ and $\mathcal{P}$, respectively. The set $\mathfrak{S} = \mathit{LVar} \cup \mathit{GVar} \cup \mathit{SVar} \cup V \cup \mathcal{F} \cup \mathcal{P}$ is called the *signature* of the logic.

In this chapter, the sets of function and predicate symbols are fixed. They contain the usual integer and boolean operators with their standard semantics and the theories of heaps (see Sect. 1.5) and final sequences.

Final sequences (i.e., tuples of arbitrary size) are represented by the algebraic data type $\mathbb{S}$. The constructors are $\langle\rangle$ (empty), $\langle\cdot\rangle$ (singleton) and

[22] This is what actually happens inside the Java Virtual Machine, cf. [Lindholm et al., 2014, Sect. 3.11].

[23] Rigid variables are essential to the expressiveness of the logic. Without them it would be impossible to compare values in different states. E.g., expressing ‘`X` has increased by 1’ requires to introduce a rigid variable u which in every state evaluates to the prestate value of `X`.

[24] Later, we overload the function *type* to map semantical objects to their type.

$\oplus$ (concatenation). We use the two observer functions $|\cdot|$ (length) and $\cdot[i]$ (random access at position i, where i is an expression of type $\mathbb{Z}$; postfix operator). For longer sequnces, we write $\langle x_0, x_1, \ldots, x_n\rangle$ as shorthand for $\langle x_0\rangle \oplus \langle x_1\rangle \oplus \cdots \oplus \langle x_n\rangle$.

Definition 2.1 (Logic expressions)**.** Logic expressions of type $\mathbb{Z}$ are constructed as usual over integer literals, program variables, logical variables, and the operators $+$, $-$, $*$, $/$, $\%$. Expressions of type $\mathbb{B}$ are constructed using the relations $\doteq$, $>$, $<$ on integer expressions, the boolean literals *true* and *false*, and the logical operators $\wedge$, $\vee$, $\neg$.

Expressions of type $\mathbb{S}$ are constructed using the operators $\langle\cdot\rangle$, $\oplus$, $|\cdot|$, and $\cdot[\cdot]$. Expressions of types $\mathbb{F}$, $\mathbb{L}$, $\mathbb{H}$, and $\mathbb{T}$ are constructed using the special variables `heap` and `heap'`; and the operators $\dot{\varnothing}$, $\{\cdot\}$, $\dot{\in}$, $\dot{\cap}$, $\dot{\cup}$, $\setminus$, $\cdot^{\mathtt{C}}$, *select*, *store*, and *anon* as described above in Sect. 1.5.

Integer and boolean logic expressions are constructed similar to their program expression counterparts (cf. Def. 1.1). They may additionally contain logical variables and 'special' variables. They must contain not global program variables. Instead, they may refer to the special program variable `threads`. For a concise representation, we pairwise identify the literals and operators of program and logic expressions, e.g., the symbols `&&` and $\wedge$ denote the same operator. We display them in program style (using typewriter font) when they appear inside of programs and in math style when appearing outside.

Definition 2.2 (State updates)**.** For $i \in [0, n]$, let x_i be a local program variable, and let a_i be an expression. Then, $\{x_0 := a_0 \,||\, \ldots \,||\, x_n := a_n\}$ is a *parallel update*. Let $\mathcal{U}_0, \ldots, \mathcal{U}_m$ be parallel updates, then $\mathcal{U}_0 \ldots \mathcal{U}_m$ is a *sequential update*. *Update* means parallel or sequential update.

For instance, $\{x := 4\}$ and $\{x := x+1\}$ are elementary updates. Applying these updates sequentially (after each other, from right to left) to the formula $x \doteq 5$ yields $4 + 1 \doteq 5$. The parallel update $\{x := 4 \,||\, x := x + 1\}$ behaves differently. When there are conflicting assignments to variable x, the last item 'wins' in our semantics. Applied to $x \doteq 5$, the resulting formula is $x + 1 \doteq 5$. Although the calculus is complete with just sequential updates, parallel updates allow ad hoc simplifications where a modality is still present. We will not go into much more detail; a complete calculus for parallel updates can be found in [Rümmer, 2006].

Formulae

CDTL formulae have the general appearance $\mathcal{U}[\![\pi]\!]\varphi$ where $\mathcal{U}$ is an update, π is a sequential dWRF program, and φ is a formula (that may or may not contain temporal operators and further sub-formulae of the same form).

Intuitively, $\mathcal{U}[\![\pi]\!]\varphi$ expresses that φ holds when evaluated over all traces τ such that the initial state of τ is (partially) described by $\mathcal{U}$ and the further states of τ are constructed by running the program π.

Definition 2.3 (Formula)**.** *State formulae* and *trace formulae* are inductively defined as follows:

0. All boolean expressions (Def. 2.1) are state formulae.
1. All state formulae are also trace formulae.
2. If φ and ψ are (state or trace) formulae, then the following are trace formulae: $\Box\varphi$ (always), $\bullet\varphi$ (weak next), $\varphi \, \mathsf{U} \, \psi$ (until).
3. If $\mathcal{U}$ is an update and φ a state formula, then $\mathcal{U}\varphi$ is a state formula.
4. If π is a sequential program (Def. 1.2) and φ a trace formula, then $[\![\pi]\!]\varphi$ is a state formulae.
5. The sets of state and trace formulae are closed under the logical operators $\neg, \wedge, \forall$.

In addition, we use the following abbreviations:

$$\begin{array}{rclllrcll} \Diamond\varphi &:=& \neg\Box\neg\varphi & \text{(eventually)}, & & \circ\,\varphi &:=& \neg\bullet\neg\varphi & \text{(strong next)}, \\ \varphi\,\mathsf{W}\,\psi &:=& \varphi\,\mathsf{U}\,\psi \vee \Box\varphi & \text{(weak until)}, & & \varphi\,\mathsf{R}\,\psi &:=& \neg(\neg\varphi\,\mathsf{U}\,\neg\psi) & \text{(release)}, \\ \varphi \vee \psi &:=& \neg(\neg\varphi \wedge \neg\psi), & & & \varphi \rightarrow \psi &:=& \neg\varphi \vee \psi, & \\ \exists x.\varphi &:=& \neg\forall x.\neg\varphi. & & & & & & \end{array}$$

A formula is called *non-temporal* if it neither contains a temporal operator nor a program modality $[\![\pi]\!]$. A formula is a *strict* DTL formula if it contains only program modalities with noninterleaved programs.

A complete syntactical schema of the logic used in this work can be found in Tab. 2.1, while the program syntax appears in Tab. 1.1 on page 7.

2.2 Semantics of Concurrent DTL

We extend [Beckert and Bruns, 2013] to concurrent dWRF programs. We use modalities of the shape $[\![\pi_t]\!]$ where π_t the program associated with a thread $t \in T$ under investigation, where T is the current thread pool state. Below in Lemma 2.9, we will prove that these extensions are indeed conservative.

We consider assignments to global variables to be the only statements that lead to a new observable state on the trace. All other statements are atomic in this sense. For the feasibility of proving CDTL formulae, it is important that not too many irrelevant intermediate states are included in a trace. For instance, if a formula such as $[\![\pi]\!]\Box\varphi$ is to be proven valid, intermediate states require sub-proofs showing that φ holds in each of them.

Table 2.1: Syntax of Concurrent Dynamic Trace Logic. The program syntax (rule π) can be found in Tab. 1.1 on page 7.

$$
\begin{array}{lcl}
\varphi & ::= & \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \forall x{:}T.\varphi \mid \exists x{:}T.\varphi \mid \mathcal{U}\varphi \mid [\![\pi]\!]\psi \mid b \\
\psi & ::= & \bullet\psi \mid \circ\psi \mid \Box\psi \mid \Diamond\psi \mid \psi \,\mathsf{U}\, \psi \mid \psi \,\mathsf{W}\, \psi \mid \psi \,\mathsf{R}\, \psi \mid \varphi \\
\mathcal{U} & ::= & \{\mathtt{v} := e\} \\
T & ::= & \mathbb{B} \mid \mathbb{Z} \mid \mathbb{H} \mid \mathbb{F} \mid \mathbb{L} \mid \mathbb{T} \mid \mathbb{S} \\
b & ::= & \mathit{true} \mid \mathit{false} \mid \ e \doteq e \ \mid \ z < z \ \mid \ \ldots \\
e & ::= & x \mid \mathit{select}(h, \mathtt{G}) \mid \mathrm{ifthenelse}(b, e, e) \mid \bot \mid m(e) \mid S[z] \mid z \mid h \mid L \mid S \\
z & ::= & z + z \ \mid \ z * z \ \mid \ -z \mid |S| \mid 0 \mid 1 \mid 2 \mid \ \ldots \\
h & ::= & \mathtt{heap} \mid \mathtt{heap'} \mid \mathtt{threads} \mid \mathit{store}(h, \mathtt{G}, e) \mid \mathit{anon}(h, L) \\
L & ::= & \dot{\varnothing} \mid \{\mathtt{G}\} \mid L \mathbin{\dot\cup} L \mid L \mathbin{\dot\cap} L \mid L \setminus L \mid L^{-1} \\
S & ::= & \langle e \rangle \mid S \oplus S
\end{array}
$$

Expression Semantics

Definition 2.4 (Variable assignments)**.** A *variable assignment* β is a function assigning integer values to all logical variables, i.e., $\beta : V \to \mathcal{D}$. Similar to the notion for states, we write $\beta\{x \mapsto d\}$ for the updated variable assignment.

Definition 2.5 (Interpretation)**.** An *interpretation* I is a mapping of function symbols $f \in \mathcal{F}$ with signature $A_1 \times \cdots \times A_n \to B$ to a semantical function $I(f) : \mathcal{D}_{A_1} \times \cdots \times \mathcal{D}_{A_n} \to \mathcal{D}_B$ and of predicate symbols $p \in \mathcal{P}$ with signature $A_1 \times \cdots \times A_n$ to a relation $I(p) \subseteq \mathcal{D}_{A_1} \times \cdots \times \mathcal{D}_{A_n}$ with the definitions in [Weiß, 2011, Def. 5.4] (for dynamic logic with an explicit heap) and [Beckert et al., 2013, Appendix A] (for the theory of finite sequences).

Semantical functions are always total. Possible gaps, such as division or modulo by zero, are underspecified [Gries and Schneider, 1995; Hähnle, 2005]. The concurrent dynamic logic of Beckert and Klebanov [2013] also defines semantics through underspecification.

Remark. The tuple $(\mathcal{D}, I, s)$ consisting of the domain, an interpretation, and a state forms a *first-order structure.* In most works on classical first-order logic (and also mostly in general modal logics), the program variable assignment s forms part of the interpretation, with program variables being 0-ary functions. In the dynamic logic literature (cf. [Weiß, 2011]), however, it has recently become customary to separate those in order to distinguish between nonrigid (i.e., functions and predicates) and rigid (i.e., program variables) entities.

Definition 2.6 (Semantics of expressions)**.** Given a state s and a variable assignment β, the value $a^{I,s,\beta}$ of an expression a of type A in a state s is the value $d \in \mathcal{D}_A$ resulting from interpreting program variables x by x^s, logical variables u by u^β, and using the interpretation I for all functions and relations.

Table 2.2: Defining axioms for location set, heap, and sequence theories

$$select(store(h, X, a), Y) \doteq \text{ifthenelse}(X \doteq Y, a, select(h, Y)) \tag{2.1}$$

$$select(anon(h, L), X) \doteq \text{ifthenelse}(X \mathrel{\dot\in} L, sk, select(h, X)) \quad \text{where } sk \text{ is a fresh symbol} \tag{2.2}$$

$$x \mathrel{\dot\notin} \dot\varnothing \tag{2.3}$$

$$x \mathrel{\dot\in} \{x\} \tag{2.4}$$

$$x \mathrel{\dot\in} L_0 \mathbin{\dot\cap} L_1 \leftrightarrow x \mathrel{\dot\in} L_0 \wedge x \mathrel{\dot\in} L_1 \tag{2.5}$$

$$x \mathrel{\dot\in} L_0 \mathbin{\dot\cup} L_1 \leftrightarrow x \mathrel{\dot\in} L_0 \vee x \mathrel{\dot\in} L_1 \tag{2.6}$$

$$x \mathrel{\dot\in} L_0 \setminus L_1 \leftrightarrow x \mathrel{\dot\in} L_0 \wedge x \mathrel{\dot\notin} L_1 \tag{2.7}$$

$$x \mathrel{\dot\in} L^{\complement} \leftrightarrow x \mathrel{\dot\notin} L \tag{2.8}$$

$$|\langle\rangle| \doteq 0 \tag{2.9}$$

$$|s_0 \oplus s_1| \doteq |s_0| + |s_1| \tag{2.10}$$

$$0 \leq i < |s_0| \rightarrow (s_0 \oplus s_1)[i] \doteq s_0[i] \tag{2.11}$$

$$|s_0| \leq i < |s_0| + |s_1| \rightarrow (s_0 \oplus s_1)[i] \doteq s_1[i] \tag{2.12}$$

Since the interpretation I is assumed to be fixed in a structure, with the standard interpretations for usual function symbols, we usually omit I. Program expressions that do not contain logical variables are independent of β, and we write a^s instead of $a^{I,s,\beta}$. If a is a boolean expression, we write $I, s, \beta \models a$ resp. $s \models a$ to denote that $a^{I,s,\beta}$ resp. a^s is true.

Since the heap and sequence theories are built on (co)algebraic data types, it is trivial to give standard interpretations, yet not very instructive here. For reference, we give some of the defining axioms in Tab. 2.2. Complete accounts are provided by Weiß [2011] (heap) or Beckert et al. [2013, Appendix A] (sequences), respectively.

CDTL Formula Semantics

In Sects. 1.6f., we have defined semantics for dWRF programs based on traces of program states. States (Def. 1.5) are functions mapping program variables to values. Local variables are directly mapped to values of their respective type, the ‘special’ variables `heap` and `heap'` are mapped to functions themselves, mapping global variables to values. The function trc_Σ (Defs. 1.7, 1.11 and 1.16) assigns a trace to an initial state and a sequential program, w.r.t. a deterministic fair scheduler Σ (cf. Def. 1.14).

Remark. The tuple $(\mathcal{D}, I, \mathcal{S}, \rho)$ with a transition relation $\rho = \{(s, \pi, s') \in \mathcal{S} \times Prg \times \mathcal{S} \mid |trc_\Sigma(s, \pi)| > 1 \wedge trc_\Sigma(s, \pi)[1] = s'\}$ forms a standard Kripke structure [Kripke, 1963]. We do not use this notation here since it only

relates initial and final states of an execution, but we are interested in all intermediate states.

We have now everything at hand needed to define the semantics of CDTL formulae in a straightforward way. The valuation of a state formula is given w.r.t. a state s and a variable assignment β; and the valuation of a trace formula is given w.r.t. a trace τ and a variable assignment β. This is expressed by the validity relation, denoted by $\vDash$. For the sake of uniformity, we do not distinguish between state and trace formulae here.

Definition 2.7 (Validity in CDTL)**.** Given a computation trace τ, variable assignment β, and fair scheduler Σ; the *validity* relation $\vDash$ is the smallest relation satisfying the following.

$$
\begin{array}{lll}
\tau,\beta,\Sigma \vDash a & \text{iff} & a^{\tau[0],\beta} = \mathit{true} \\
\tau,\beta,\Sigma \vDash \neg\varphi & \text{iff} & \tau,\beta,\Sigma \nvDash \varphi \\
\tau,\beta,\Sigma \vDash \varphi \wedge \psi & \text{iff} & \tau,\beta,\Sigma \vDash \varphi \text{ and } \tau,\beta,\Sigma \vDash \psi \\
\tau,\beta,\Sigma \vDash \forall u{:}U.\varphi & \text{iff} & \text{for every } d \in \mathcal{D}_U\text{: } \tau,\beta\{u \mapsto d\},\Sigma \vDash \varphi \\
\tau,\beta,\Sigma \vDash \Box\varphi & \text{iff} & \tau[i,\infty),\beta,\Sigma \vDash \varphi \text{ for every } i \in [0,|\tau|) \\
\tau,\beta,\Sigma \vDash \varphi \,\mathsf{U}\, \psi & \text{iff} & \tau[j,i),\beta,\Sigma \vDash \varphi \text{ and } \tau[i,\infty),\beta,\Sigma \vDash \psi \\
 & & \quad \text{for some } i \in [0,|\tau|) \text{ and all } j \in [0,i) \\
\tau,\beta,\Sigma \vDash \bullet\,\varphi & \text{iff} & \tau[1,\infty),\beta,\Sigma \vDash \varphi \text{ or } |\tau| = 1 \\
\tau,\beta,\Sigma \vDash \{x_1 := a_1 \,||\, \ldots & & \\
\qquad\qquad || \, x_n := a_n\}\varphi & \text{iff} & \tau\{x_1 \mapsto a_1^{\tau[0]}\}\ldots\{x_n \mapsto a_n^{\tau[0]}\},\beta,\Sigma \vDash \varphi \\
\tau,\beta,\Sigma \vDash [\![\pi]\!]\varphi & \text{iff} & \mathit{trc}_\Sigma(\tau[0],\pi),\beta,\Sigma \vDash \varphi
\end{array}
$$

A formula φ is *valid*, written $\vDash \varphi$, if $\tau,\beta,\Sigma \vDash \varphi$ for all τ, β, and Σ.

In this definition, the scheduler Σ forms part of the validity relation. It entails an implicit universal quantification over all (fair) schedulers on the semantical level. A result of that is that our logic still uses only one kind of modality, that speaks about *the* deterministic trace. An alternative definition would be to introduce two modalities, that universally or existentially range over schedulers,[25] respectively.

Example 2.8. Assume that the scheduler is *not* part of the validity relation, but the semantics of $[\![\cdot]\!]$ is defined w.r.t. all schedulers. Let us consider a dual modality $\langle\!\langle\cdot\rangle\!\rangle$ that is defined w.r.t. *some* scheduler. Consider a concurrent program with thread pool $T = \{t_0, t_1\}$ and $\pi_{t_0} =$ `X=0;` and $\pi_{t_1} =$ `X=1;`. Depending on the concrete scheduler, the final value of `X` can be either 0 or 1. Thus the formulae $\langle\!\langle\pi_{t_0}\rangle\!\rangle\circ\mathtt{X} \doteq 0$ and $\langle\!\langle\pi_{t_0}\rangle\!\rangle\circ\mathtt{X} \doteq 1$ are both valid, while neither $[\![\pi_{t_0}]\!]\circ\mathtt{X} \doteq 0$ nor $[\![\pi_{t_0}]\!]\circ\mathtt{X} \doteq 1$ is valid.

[25] or traces, equivalently

On the one hand, this entails the obvious disadvantage that the semantics of a formula is defined in terms of *concrete* parallel programs. As a result, this definition is not modular. However, as we will see in Sect. 3.4.1, the rely/guarantee approach allows us to reason about interleavings modularly w.r.t. any environment.

On the other hand, our single modality is dual to itself and therefore exhibits some good properties. For instance, the formulae $\neg[\![\pi]\!]\varphi \leftrightarrow [\![\pi]\!]\neg\varphi$ or $[\![\pi]\!](\varphi_1 \vee \varphi_2) \leftrightarrow ([\![\pi]\!]\varphi_1 \vee [\![\pi]\!]\varphi_2)$ are tautologies.[26] This allows to give a smaller and more efficient calculus as compared to a calculus that would have to deal with two kinds of modalities,[27] as the results of Jeannin and Platzer [2014] suggest, for instance.

Remark. In this definition, formulae are always of the shape $[\![\pi]\!]\varphi$. This formula is valid if and only if φ is valid on any trace of π under any *deterministic* scheduler Σ. This is equivalent to φ being valid on any trace under any *indeterministic* scheduler. The reason is that any indeterministic scheduler can be simulated by a set of deterministic schedulers.

The logic CDTL presented here is a *semantical conservative extension* of the base DTL logic presented by Beckert and Bruns [2013]. This allows us to adapt the base DTL calculus to a sound calculus for the sequential part of CDTL. It follows from Lemma 1.9 that the replacement of program modalities is welldefined.

Lemma 2.9 (Semantical conservative extension)**.** *Let φ be a valid formula according to [Beckert and Bruns, 2013, Def. 9]. Let φ' be the CDTL formula obtained from replacing all quantifiers by their $\top$-typed equivalent and all program modalities by a CDTL equivalent (i.e., restricting to simple expressions). Then φ' is a valid CDTL formula.*

2.3 Discussion

In this chapter, we have defined Dynamic Trace Logic (DTL) and its extension to concurrent programs, CDTL, that stem from a novel combination of dynamic logic and first order temporal logic. A complete calculus for DTL that is proven sound and complete can be found in [Beckert and Bruns, 2013] (with the proofs in [Beckert and Bruns, 2012]. In contrast to previous work by Beckert and Schlager [2001]; Platzer [2007], there is no restriction on the shape of trace formulae. Through this, we have got an expressive logic allowing to describe complex temporal properties of programs. We present a proof of a nontrivial DTL formula below; other (smaller) proofs can be found in [Wagner, 2013].

[26] Another example of a modality with this property is the update operator of dynamic logic, if viewed as a modality.

[27] As already discussed in Sect. 1.3, the possibility of an environment macro step of infinite length would require such a change.

Implementation

The sequent calculus $\mathcal{C}_{\mathrm{DTL}}$ for the sequential subset of the language has been prototypically implemented on top of the release version 2.2 of the interactive KeY prover.[28] Instead of the simple toy language introduced in this paper, the implemented calculus works on actual Java programs. The implementation benefits from the fact that most complex statement in Java can be transformed into a sequence of simple statements. This is a key element of the symbolic execution in the Javadynamic logic (DL) calculus of the KeY system [Beckert, 2001; Beckert, Klebanov, and Schlager, 2007b]. Most calculus rules dealing with these kind of program normalization can be adapted straight away from the present rules for the $[\cdot]$ modality in the JavaDL calculus.

[28] This version is available on request.

Chapter 3

Deductive Verification of Concurrent Programs

In this chapter, we extend the calculus for the sequential fragment of dWRF, as presented by Beckert and Bruns [2013], to the full multithreaded language, including interleavings and thread creation. Our goal is to allow modular reasoning in *open programs*, that can be extended with further threads. The rely/guarantee approach allows to reason modularly about the behavior of shared memory concurrent programs. We regard one thread (executing a sequential interleaved program—as defined in Sect. 1.4) in isolation, with possible spontaneous state transitions induced by the environment. This means that we still have a deterministic program semantics, with underspecified (i.e., havocked) heap states. Rely/guarantee uses functional specification to restrict the effect of these transitions. In this sense, reasoning about interleavings is similar to reasoning about sequential method invocations through contracts. We present an implementation in dynamic logic with 'contracts' for each heap read access. To reduce the specification overhead, we complement functional rely/guarantee specifications with framing, that restricts havoc to defined partitions of the heap.

3.1 Concurrent Verification

There have been several approaches to formally reason about shared memory concurrent programs since the mid-1970s, such as [Ashcroft and Manna, 1971; Ashcroft, 1975; Keller, 1976; Hoare, 1978; Lamport, 1980]. Widely known is the one by Owicki and Gries [1976], that is considered the first practical approach to concurrency verification. They define a Hoare logic for programs with parallel composition. The major issue with this approach is that the rule for parallel composition requires isolated threads. This means that, in addition to prove local correctness of programs, one needs to prove noninterference of parallel executions. This technique has some limitations:

1. the number of concurrent threads is fixed; 2. noninterference proofs tend to be complicated;[29] 3. it is not compositional in the sense that parts of the concurrent program could be verified in isolation, but only the (closed) composed system as a whole; and 4. most importantly, programs that actually *do* interfere can only be verified against very weak specifications. This is not practical since modularity is a key to feasible verification of sophisticated software systems.

The rely/guarantee approach [Jones, 1983; Stølen, 1991] (sometimes also called "assume/guarantee") attempts to overcome these issues. It abstracts away from concrete interferences to only consider the *effects* of possible interleavings. The main idea is similar to design by contract, though not on the level of a public interface but of atomic program steps. This makes it possible to modularly reason about one single thread in isolation, while there may be an unbounded number of others in an only partially specified environment. It is, in particular, not of any interest which sequential program other threads execute or in which (thread local) state they are. Sequential programs (i.e., single threads) are evaluated over traces of states. The original paper by Jones sketches this fundamental idea. A comprehensive account on the rely/guarantee approach can be found in the article by Xu, de Roever, and He [1997], that includes a Hoare style calculus and proofs of soundness and completeness for a fixed number of threads. A completeness proof for a system that is parametric in the number of threads can be found in [Prensa Nieto, 2002]. Xu et al. further "observe that the rely-guarantee method is [. . .] a reformulation of the classical non-compositional Owicki & Gries method."

Overview

In this chapter, we describe how rely/guarantee can be integrated into our logical framework. Section 3.2 gives some fundamental definitions regarding the rely/guarantee approach and the kind of programs that we consider. In Sect. 3.3, we develop correctness conditions (partly thread-local, partly on the system level) that—if they hold—assure soundness of a calculus rule to deal with environment actions, to be introduced in Sect. 3.4.1. We instrument noninterleaved programs with the special statement `release`; representing an environment macro-step. We give a proof of soundness for this rule and a rigorous argument on why a calculus using it is complete w.r.t. the proposed target programing language.

[29] The proofs grows exponentially with the number of threads.

3.2 Rely/Guarantee Reasoning

The central idea of rely/guarantee is to describe the transition functions from Sect. 1.7, σ_t (of the thread t under investigation) and $\sigma^*_\Sigma(t)$ (of the environment), in specifications using formulas *rely* and *guar*. Those are *two-state invariants*, i.e., they are preserved throughout the execution and are evaluated over two succeeding states. The formula *rely* describes $\sigma^*_\Sigma(t)$, i.e., it defines on which properties the execution of t may rely upon. The formula *guar* describes σ_t, i.e., it defines which properties the execution of t has to guarantee.

Obviously, there always are strongest formulae satisfying these conditions (if the environment is perfectly known): The strongest rely relation is the reflexive/transitive closure of the union of guarantee relations. In practice, this strongest condition will not be necessary. It is sufficient that *rely* is strong enough to imply the postcondition in a final state and that *guar* is strong enough—in disjunction with the *guar* specification of other threads—to imply the *rely* conditions of a third party thread. Since they describe the behavior of zero or more atomic environment transitions, rely conditions have to be always reflexive and transitive. Jones additionally requires guarantees to be reflexive and transitive. But this restricts the possible specifications and requires an additional proof obligation, while it does not provide any advantages since the union of transitive relations is not necessarily transitive again. Transitive closure of guarantees is not expressible in first order logic anyway. Xu et al. do not require rely conditions to be transitive. Instead, they require that pre- and postconditions are stable under environment transitions. This means that rely conditions are partial equivalence relations. While this is more liberal on the shape of rely conditions, it severely restricts pre- and postconditions in practice.

In typical cases, the memory partitions to which different threads write to are strongly separated and only a few locations are actually shared. Therefore we combine the well known two state invariant specification of threads with *framing*, to specify what locations a thread writes to at most (and what locations it can rely on not to be changed). Frame specifications alone can be very expressive, in the dynamic frames approach [Kassios, 2011; Weiß, 2011], location sets describing frames can depend on the program state and can be constructed through comprehensions. Through framing, we take the burden of specifying the 'nonbehavior' of threads in addition to its behavior.

We also borrow the concept of *preconditions* from Design by Contract (DbC), to restrict the states in which fresh threads can be created. Like framing, this does not increase the expressiveness of the approach, but it is very effective in reducing the specification overhead. Following the approach by Weiß, we do not include implicit class invariants in this framework, but leave it to the specifier to refer to invariants explicitly in postconditions (or guarantees).

3.2.1 Relevant Interleavings

Through instrumentation with the explicit `release` statement, we construct sequential programs that simulate the *observable* runtime behavior of concurrent programs. Although in a real concurrent program the environment may be active at *any* point in time, with this definition, we restrict it to fewer states. The rationale behind this is to already keep to model simple enough to efficiently reason about. This is justified—on the meta level—by the observation that only some interleavings are actually observable to the thread under investigation.

An interleaving is only observable if has an effect on 1. the control flow or 2. a property stated about the program. Since control statements in our language have simple conditions, the control flow can only be influenced in assignments from the global state. that appear immediately before a read or termination action. Thus it is sufficient to consider the interleaving before the read action. Note that a write action is never influenced by environment actions. In the actual concurrent behavior, the order of writes may be different, but the observable effect of a write action is always the same. Item 2 is more intricate. 'Properties' does not only include the trace properties that appear after program modalities in formulae, but also any property stated on subtraces. Fortunately, our calculus does not allow arbitrary trace decompositions, since program rules do (usually) focus on active statements. This is in contrast to other program logics, in particular Hoare logics, that have sequential decomposition rules.[30] The only exception are the invariant rules, that state properties about the subtrace induced by the loop body.

To simulate the concurrent behavior, we instrument noninterleaved programs with environment action statements and amend the invariant rules. A noninterleaved program is instrumented such that environment actions appear before every heap read and the termination action.

Definition 3.1 (Instrumented program)**.** Let $\pi = \langle stm_1, \ldots, stm_n \rangle$ be a noninterleaved program. The corresponding *instrumented program* $\underline{\pi}$ is constructed following: For each statement stm_i with $i \in (0, n]$,

- if stm_i is a local assignment with a nonsimple expression on the right hand side, the statement `release;` is inserted before stm_i,
- if $stm_i =$ `while` (b) $\{\pi'\}$, it is replaced by `while` (b) $\{\underline{\pi'}\}$
- if $stm_i =$ `if` (b) $\{\pi_1\}$ `else` $\{\pi_2\}$, it is replaced by `if` (b) $\{\underline{\pi_1}\}$ `else` $\{\underline{\pi_2}\}$,
- and the statement `release;` is inserted after stm_n.

[30] Confer [Beckert et al., 2007b, p. 115] for a discussion on this.

Please note that this instrumentation is purely syntactic and independent of a thread pool. Later, we expect that programs under investigation are already instrumented.

Like the control flow, invariants can depend on shared locations. This means that the original invariant rules presented by Beckert and Bruns [2013] are not sound for concurrent programs. A concurrently executed thread may influence whether the invariant γ holds. For this reason, we slightly adapt the invariant rules. They only differ from the original ones by additional instrumentation. In all rules, in the first premiss the invariant γ is replaced by $[\![__]\!]\gamma$ (i.e., a program modality with the instrumentation of the empty program). This adds exactly one interleaving point at the end of the empty program. In the second and the last premiss of all rules, the loop body π is instrumented, which effectively adds an interleaving point at the end of π. Further instrumentations (i.e., on the trailing program ω or the complete loop) are not necessary since the programs are either already properly instrumented or the same invariant rules apply.

3.3 Proof Obligations

A *thread specification* is a tuple $(pre_t, rely_t, guar_t, R_t, M_t)$ where pre_t is a state formula, $rely_t$ and $guar_t$ are two-state formulae, and R_t and M_t are terms of type LocSet. The intuitive understanding is that the active thread can rely on the relation $rely_t$ to hold between state transitions induced by the environment while the locations in R_t never change, and at the same time, it guarantees only to write to the locations in M_t and to maintain the relation $guar_t$ between all atomic steps. The precondition pre_t restricts the states in which fresh threads may be created.

A formal definition of a thread specification being valid is given in Def. 3.10 on page 35. The formulas *rely* and *guar* are still state formulae in the sense that they must not include temporal operators, but they are expected to refer to the builtin heap variables `heap` and `heap'`, for that we justify it as 'two-state.' The location set expressions can be nontrivial, e.g., depending on the state or including if-then-else operators. This makes location set expressions as expressive the logic itself.

Thread specifications relate only to one thread, not to a complete concurrent program. This is important in order to have modular specifications for reasoning about open systems. The choice of a set R_t of locations that must *not change*, instead of the set of locations that may change, may seem counterintuitive at first sight. But in a strictly modular setting, we (consequently) cannot name the locations that are allowed to change.

Definition 3.2. Let s and s' be two states and $\mathcal{L}$ a location set. The states s and s' are *$\mathcal{L}$-equivalent*, written as $s \approx_{\mathcal{L}} s'$, if $\mathtt{heap}^s(F) = \mathtt{heap}^{s'}(F)$ for all global variables $F \in \mathcal{L}$. A binary relation $A \in \mathcal{S}^2$ is *$\mathcal{L}$-invariant* if $A \subseteq \approx_{\mathcal{L}}$.

Lemma 3.3. $\approx_{\mathcal{L}}$ *is an equivalence relation.*

Guarantees

In order to establish that a thread t of a concurrent system $(\mathcal{T}, \Sigma)$ satisfies a thread specification $(pre_t, rely_t, guar_t, R_t, M_t)$, need to prove that—under rely condition $rely_t$—it only writes to locations specified in M_t and that it fulfills the two state invariant $guar_t$. The precondition pre_t will be used to relax the 'guarantee' proof obligation in a way such that it only needs to hold for states resulting from the creation of new threads; see below in Sect. 3.4.2. This relation needs to be proven for any two succeeding states in the trace. The property that only locations in M_t may be changed throughout the program execution is also known as *strict modifies clause*. This is in contrast to *weak modifies* properties as imposed in standard JavaDL [Beckert et al., 2007a; Weiß, 2011], that still allow locations outside M_t to be changed temporarily. A proof obligation can be formulated using the trace modality as

$$\{h^{\text{pre}} := \texttt{heap}\}[\![\underline{\pi_t}]\!]\bullet\Box(frame_t \wedge guar_t) \tag{3.1}$$

where $frame_t$ stands for the following formula:

$$\forall F{:}\mathbb{F}.\ \Big(F \mathrel{\dot\in} \{\texttt{heap} := h^{\text{pre}}\}(R_t \mathbin{\dot\cap} M_t^{\complement}) \to select(\texttt{heap}, F) \doteq select(\texttt{heap'}, F)\Big) \tag{3.2}$$

The formula $frame_t$ is similar to the one used in [Weiß, 2011, Sect. 6.4.1] to formalize weak modifies properties. The values of the location set expressions are state dependent, but evaluate in the initial state of the trace. This is assured through the updates, that store the initial heap h^{pre}. A difference is that not only the very first and the final state are in relation, but every pair of consecutive states.[31] Another difference is that we allow the values of locations in both M_t and $R_t^{\complement}$ to change, i.e., all locations in $\mathcal{L}_t := \Big(R_t \mathbin{\dot\cap} M_t^{\complement}\Big)^{s_0}$ must evaluate to the same value, due to possible concurrent changes. The second part of the formula entails the two state invariant property. Note that the proof obligation of (3.1) could be written as two separate ones, since the formula $[\![\pi]\!]\bullet\Box(\varphi_1 \wedge \varphi_2)$ is equivalent to $[\![\pi]\!]\bullet\Box\varphi_1 \wedge [\![\pi]\!]\bullet\Box\varphi_2$.

Lemma 3.4. *Let s_0 be a state; let t be a thread and thread specification $(pre_t, rely_t, guar_t, R_t, M_t)$. Let $\tau = trc_\Sigma(s_0\{h^{\text{pre}} \mapsto \texttt{heap}^s\}, \pi_t)$. If $\tau \vDash \bullet\Box(frame_t \wedge guar_t)$ (as in (3.1)), then*

1. *$guar_t$ describes a relation binary $\gamma_t \supseteq \sigma_t$.*
2. *γ_t and σ_t are $\mathcal{L}_t$-invariant.*

[31] Note that we do not have to use a temporal construct to refer to the previous state (there are no past operators in our logic, anyways), but through the variable `heap'` since we do not need the complete state, but just the heap state.

Proof. Ad 1: follows from Lemma 1.17.
Ad 2: The two-state formula $frame_t$ formalizes $s' \approx_{(R_t \dot{\cap} M_t^{\complement})^{s'''}} s''$ for all states with $\texttt{heap'}^{s'} = \texttt{heap}^{s''}$ and $(h^{\text{pre}})^{s'} = \texttt{heap}^{s'''}$. From the assumption, $frame_t$ is valid for $s''' = s_0$ and $s' = s_i$ for any $i \in [1, |\tau|)$. Lemma 1.17 then gives $s'' = s_{i-1}$. Thus it is $s_{i-1} \approx_{\mathcal{L}_t} s_i$ for all $i \in [1, |\tau|)$. By reflexivity and transitivity of $\approx$ (Lemma 3.3), it follows $s_i \approx_{\mathcal{L}_t} s_j$. ◁

Remark. A slight alteration (i.e., removing 'next') of Formula 3.1 to

$$\{\texttt{heap'} := \texttt{heap} \,||\, h^{\text{pre}} := \texttt{heap}\}[\![\underline{\pi_t}]\!]\Box(frame_t \wedge guar_t)$$

requires $guar_t$ to describe a reflexive relation. (The formula $guar_t$ is true in the initial state of τ. The only restriction on τ is that $\texttt{heap'}^{\tau[0]} = \texttt{heap}^{\tau[0]}$, thus the result is universally valid.)

To restrict the possible initial states in formula (3.1), we relax this formula using a precondition. The following formula is valid in all states in which (3.1) is valid or the formula pre_t is not valid.

$$\begin{aligned} pre_t \to \{h^{\text{pre}} := \texttt{heap}\}[\![\underline{\pi_t}]\!]\bullet\Box(\forall F : Field.\ (F \mathrel{\dot\in} \{\texttt{heap} := h^{\text{pre}}\}(R_t \mathrel{\dot\cap} M_t^{\complement}) \\ \to select(\texttt{heap}, F) \doteq select(\texttt{heap'}, F)) \wedge guar_t) \end{aligned} \tag{3.3}$$

Lemma 3.5. *Let everything be as in Lemma 3.4. Let the relation $\tilde\sigma_t$ be defined as $\sigma_t \cup \{(s, s') \mid s \nvDash pre_t\}$. Formula* (3.3) *is valid if and only if $\tilde\sigma_t \subseteq \gamma_t$ and $\tilde\sigma_t$ is $\mathcal{L}_t$-invariant.*

Rely Conditions

The idea of rely conditions is that they describe an upper bound on environment actions. This entails two items: 1. Since a `release` denotes an environment macro step (i.e., zero or more atomic steps), rely conditions must describe reflexive and transitive relations. 2. They must not be stronger than the combined guarantees that the environment prescribes. These are formalized in (3.4) and (3.5) below. While (3.4) is a 'local' property—i.e., it is a property of one rely condition alone—property (3.5) is concerned with the relation of rely conditions to guarantee conditions in the combined system.

$$\begin{aligned} \forall h_1, h_2, h_3{:}\mathbb{H}.\ (\quad & \{\texttt{heap'} := h_1 \,||\, \texttt{heap} := h_1\}rely_t \\ \wedge \quad (\quad & \{\texttt{heap'} := h_1 \,||\, \texttt{heap} := h_2\}rely_t \\ \wedge & \{\texttt{heap'} := h_2 \,||\, \texttt{heap} := h_3\}rely_t \quad) \\ \to \quad & \{\texttt{heap'} := h_1 \,||\, \texttt{heap} := h_3\}rely_t \quad) \end{aligned} \tag{3.4}$$

Lemma 3.6 (Reflexivity/transitivity of *rely*)**.** *Let φ be the formula in* (3.4)*. It is a valid formula if and only if $rely_t$ describes a binary relation $\rho_t \subseteq \mathcal{S}^2$ that is reflexive and transitive.*

System Properties

As already mentioned above, rely conditions must not be stronger than the combined guaratees of the system. This is formalized in the following:

$$\{\texttt{heap'} := \texttt{heap} \,||\, \texttt{heap} := \mathit{anon}(\texttt{heap}, R_t^{\complement})\} \left(\left(\bigvee_{t' \in T \setminus t} \mathit{guar}_{t'} \right) \rightarrow \mathit{rely}_t \right) \tag{3.5}$$

Note that, in general, guarantees are allowed to describe relations that are neither reflexive nor transitive—while this is required by Jones. The essential point is that rely conditions are reflexive/transitive, and since the union of transitive relations is not necessarily transitive again, we still need (3.4) as a proof obligation anyway. There is always a strongest rely condition to fulfil obligations (3.4) and (3.5), that is the reflexive/transitive closure of the union of guarantee conditions. Since, however, transitive closure cannot be expressed in first order logic, see, e.g., [Ebbinghaus and Flum, 1995], we require these conditions here explicitly.

Similar to the functional rely condition, also the frame conditions for threads of a system need to be aligned.

$$\left(\dot{\bigcup_{t' \in T \setminus t}} M_{t'} \right) \dot{\cap} R_t \subseteq \dot{\varnothing} \tag{3.6}$$

Lemma 3.7. *Let $\gamma_t, \rho_t \subseteq \mathcal{S}^2$ be the relations introduced in Lemmas 3.4 and 3.6 for some $t \in \mathcal{T}$. If both formulae (3.5) and (3.6) are valid, then $\bigcup_{t' \in T \setminus t} \gamma_{t'} \subseteq \rho_t$.*

To prove Lemma 3.7, we first need the following definition and lemma.

Definition 3.8. Let φ be a formula and $\mathcal{L}$ a location set. We say φ is *$\mathcal{L}$-invariant* if for all states s with $s \vDash \varphi$, it holds $s' \vDash \varphi$ for all states s' with $s \approx_{\mathcal{L}} s'$.

Lemma 3.9.

1. *Let s be state and $\mathcal{L}$ a location set. The states s and $s\{\texttt{heap}^s \mapsto \mathit{anon}(\texttt{heap}^s, \mathcal{L})\}$ are $\mathcal{L}^{\complement}$-equivalent.*

2. *Let $\mathcal{L}$ be a location set and φ an $\mathcal{L}$-invariant formula. Let s be a state with $s \vDash \varphi$ and L a location set expression with $L^s = \mathcal{L}$. Then it holds that $s \vDash \{\texttt{heap} := \mathit{anon}(\texttt{heap}, L)\}\varphi$.*

3. *Let $\mathcal{L} \supseteq \mathcal{L}'$ be location sets. If a formula φ is $\mathcal{L}$-invariant, then it is $\mathcal{L}'$-invariant.*

Proof. Ad 1: This follows from the semantics of *anon* (cf. Sect. 1.5). Ad 2: This immediately follows from item 1. Ad 3: This follows from the $\mathcal{L}$-monotonicity of the $\mathcal{L}$-equivalence relation. ◁

Proof of Lemma 3.7. Lemma 3.4 provides that $guar_{t'}$ is $\mathcal{L}_{t'}$-invariant. From Lemma 3.9 it follows that $guar_{t'}$ is also R^s_t-invariant. ◁

Valid Thread Specifications

For the correctness of the rely/guarantee method, we need to establish the notion of valid thread specifications w.r.t. a particular thread pool. The rule for reasoning about interleavings can then be defined in a thread pool agnostic way.

Definition 3.10 (Valid thread specification)**.** Let $t \in \mathcal{T}$ be a thread. A thread specification $(pre_t, rely_t, guar_t, R_t, M_t)$ is *valid* for a thread pool $T \in 2^{\mathcal{T}}_{\text{fin}}$ if the formulae (3.3), (3.4), (3.5), and (3.6) are valid. A thread specification is only *locally valid* if only formulae (3.3) and (3.4) are valid, i.e., t fulfils its guarantees and the rely condition is reflexive and transitive. We call the set $\{(pre_t, rely_t, guar_t, R_t, M_t) \mid t \in T\}$ a valid thread specification for T if all $(pre_t, rely_t, guar_t, R_t, M_t)$ are valid thread specifications w.r.t. T.

Note that the thread pool T is independent of the state of evaluation of the formulae.

In another point, the properties denoted by formulas (3.3) and (3.4) are thread-local, i.e., if they are valid for a particular environment then they are also valid for any environment. The properties denoted by formulae (3.5) and (3.6) refer to the concrete, complete system instead. But they do not contain any program, only first order formulas over the theories of location sets and heaps. In contrast to the proof obligations by Jones [1983]; Stirling [1988]; Xu et al. [1997], we do not include a postcondition here. The reason is to decouple the proof of well-behaved concurrency from proofs for other properties, like functional correctness or information flow security.

Theorem 3.11. *Let $(\mathcal{T}, \Sigma)$ be a concurrent system and $t \in \mathcal{T}$ some thread. If there is a valid thread specification for $\mathcal{T}$, then it is $\sigma^*_\Sigma(t) \subseteq \rho_t$, where ρ_t is the semantical relation represented by $rely_t$.*

Proof. $\sigma^*_\Sigma(t)$ consists of atomic environment transitions $\sigma^*_\Sigma(t) = \sigma_{t_{i_1}} \circ \cdots \circ \sigma_{t_{i_k}}$ with $t_{i_j} \in \mathcal{T} \setminus \{t\}$ and $k \in \mathbb{N}$ as determined by Σ. According to Lemma 3.5, it is $\sigma_{t_{i_j}} \subseteq \tilde{\sigma}_{t_{i_j}} \subseteq \gamma_{t_{i_j}}$ and thus $\sigma^*_\Sigma(t) \subseteq \gamma_{t_{i_1}} \circ \cdots \circ \gamma_{t_{i_k}}$. We can still weaken the relation by replacing all concrete $\gamma_{t_{i_j}}$ by the union $\bigcup_{t' \neq t} \gamma_{t'}$, i.e., $\sigma^*_\Sigma(t) \subseteq \left(\bigcup_{t' \neq t} \gamma_{t'}\right)^k$. Since $\gamma_{t'}$ is R_t-invariant (Lemma 3.7), we obtain $\sigma^*_\Sigma(t) \subseteq \rho^k_t$ through Lemma 3.7. Since ρ_t is reflexive and transitive (Lemma 3.6), i.e., $\rho_t = \rho^k_t$, we conclude $\sigma^*_\Sigma(t) \subseteq \rho_t$. ◁

3.4 A Calculus for Concurrent DTL

The calulus $\mathcal{C}_{\text{CDTL}}$ for CDTL consists of the rules of $\mathcal{C}_{\text{DTL}}$ as defined by Beckert and Bruns [2013], plus Rules 37 and 38 for release and thread creation, to be introduced below.

3.4.1 Reasoning About Environment Steps

By assigning a calculus rule to the synthetic `release` statement, we can establish that it sufficient for a sequential program π to be correct w.r.t. a given specification in a concurrent setting if the instrumented program $\underline{\pi}$ is. Rule R37 below can be applied on a program modality where `release` is the active statement.

$$\frac{\Gamma, \mathcal{U}\mathcal{V} \mathit{rely}_t \Longrightarrow \mathcal{U}\mathcal{V}[\![\omega_t]\!]\varphi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\texttt{release;}\ \ \omega_t]\!]\varphi, \Delta} \; \mathsf{R37}$$

where $\mathcal{V} := \{\texttt{heap'} := \texttt{heap} \,||\, \texttt{heap} := \mathit{anon}(\texttt{heap}, R_t^{\complement}) \,||\, \texttt{threads} := T'\}$, T' is a fresh symbol of type $\mathbb{T}$, and t is the thread which program appears in the modality. The rely_t formula is inserted under the update $\mathcal{U}$, to specify environment changes from the state partially specified by $\mathcal{U}$. The heap variable is anonymized on the complement of R_t.

We follow the usual approach in software verification that specifications live as background theories and therefore are not part of formulae or sequents. This has been pursued by, e.g., Beckert et al. [2007a]; Weiß [2011] for method contracts in sequential programs. For concurrent programs, there is a similar situation with rely conditions. This means that we cannot assess the soundness of rule R37 on grounds of the rule itself, but only w.r.t. the specification framework of thread specifications.

Lemma 3.12 (Conditional soundness of R37)**.** *Assume the premiss of rule R37 to be valid. Let $s \in \mathcal{S}$ be some state; let s' be the unique state that coincides with s except for the effects of the update $\mathcal{U}$. If there exists a valid thread specification for* $\texttt{threads}^{s'}$*, then the conclusion is valid.*

Proof. Following from Thm. 3.11. ◁

Note that soundness of Rule R37 is independent of the kind of modality and the formula φ. This means that a derived rule using the $[\cdot]$ ('box') or the $\langle\cdot\rangle$ ('diamond') modality from standard dynamic logic is also sound as those modalities can be expressed using trace formulae.

The program instrumentation with the synthetic `release` statement and this corresponding calculus rule allow to extend the present calculus for purely sequential DTL conservatively. It seems desirable to overcome this instrumentation.

3.4.2 Reasoning About Thread Creation

The following theorem states which conditions are necessary to extend a thread pool while preserving validity of thread specifications. This frees us from unhandy proof obligations that are stated in terms of 'for all threads.' We make use of preconditions in this section.

Theorem 3.13 (Thread pool expansion)**.** *Let T be a thread pool and let $S_T = \{(pre_t, rely_t, guar_t, R_t, M_t) \mid t \in T\}$ be a valid thread specification for T. Let $t'' \notin T$ be another thread with locally valid specification $S_{t''} = (pre_{t''}, rely_{t''}, guar_{t''}, R_{t''}, M_{t''})$. If the following formulae are valid for some $t \in T$, then $S_T \cup \{S_{t''}\}$ is a valid thread specification for $T \cup \{t''\}$.*

(a) $\{\texttt{heap'} := \texttt{heap} \,||\, \texttt{heap} := anon(\texttt{heap}, R_t^{\complement} \mathbin{\dot\cup} M_t)\}((rely_t \vee guar_t) \rightarrow rely_{t''})$

(b) $\{\texttt{heap'} := \texttt{heap} \,||\, \texttt{heap} := anon(\texttt{heap}, R_t^{\complement})\}(guar_{t''} \rightarrow rely_t)$

(c) $(R_t^{\complement} \mathbin{\dot\cup} M_t) \mathbin{\dot\cap} R_{t''} \subseteq \dot\varnothing$

(d) $M_{t''} \mathbin{\dot\cap} (M_t^{\complement} \mathbin{\dot\cup} R_t) \subseteq \dot\varnothing$

Following this theorem, we can expand a purely symbolic thread specification system. The original thread pool T does not appear in the formulae to be proven, but only one single thread t.

Proof. It remains to show formulae (3.5) and (3.6) valid. We start with the latter.

Ad (3.6). From t having a valid thread specification, we get $\models \dot\bigcup_{t' \in T \setminus t} M_{t'} \subseteq R_t^{\complement}$ and $\models \dot\bigcup_{t' \in T \setminus t} R_{t'} \subseteq M_t^{\complement}$. Replacing $R_t^{\complement}$ and $M_t^{\complement}$ in formulae (c) and (d), respectively, gives us $\models \dot\bigcup_{t' \in T} M_{t'} \mathbin{\dot\cap} R_{t''} \subseteq \emptyset$ and $\models M_{t''} \mathbin{\dot\cap} \dot\bigcup_{t' \in T} R_{t'} \subseteq \emptyset$, that are equivalent to (3.6) for t or t'', respectively.

Ad (3.5). For t, this immediately follows from (b). For t'', the formula (a) can be weakened according to Lemma 3.9(3). We use the shorthand notation ${}^{L}\varphi$ for $\{\texttt{heap'} := \texttt{heap} \,||\, \texttt{heap} := anon(\texttt{heap}, L)\}\varphi$. We obtain $\models ({}^{R_t^{\complement}} rely_t \vee {}^{R_t^{\complement} \dot\cup M_t} guar_t) \rightarrow {}^{R_t^{\complement} \dot\cup M_t} rely_{t''}$ by update distributivity and Lemma 3.9(3) applied on $rely_t$. From the valid thread specification for t w.r.t. T, we obtain $\models {}^{R_t^{\complement}}(guar_{t'} \rightarrow rely_t)$ for some $t' \in T$. From (c) in combination with Lemma **??**, it follows that this is equivalent to $\models {}^{R_t^{\complement} \dot\cup M_t} guar_{t'} \rightarrow {}^{R_t^{\complement}} rely_t$. We then replace ${}^{R_t^{\complement}} rely_t$ by ${}^{R_t^{\complement} \dot\cup M_t} guar_{t'}$ in the above formula to obtain $\models ({}^{R_t^{\complement} \dot\cup M_t} guar_{t'} \vee {}^{R_t^{\complement} \dot\cup M_t} guar_t) \rightarrow {}^{R_t^{\complement} \dot\cup M_t} rely_{t''}$. Pulling out the update and applying Lemma 3.9(3) with formula (c) finally leads us to $\models {}^{R_{t''}^{\complement}}((guar_{t'} \vee guar_t) \rightarrow rely_{t''}$, that is what we needed to prove. ◁

We use this result to cast this into a symbolic execution rule for `fork`. The following calculus rule deals with the creation of new threads in the program modality.

$$\frac{\begin{array}{ll} \text{(a) } \Gamma \Longrightarrow \mathcal{U}\mathcal{W}[\![\omega_t]\!]\varphi, \Delta & \text{(b) } \Gamma \Longrightarrow \mathcal{U} pre_{t'}, \Delta \\ \text{(c) } \Longrightarrow \mathcal{V}_1(guar_{t'} \to rely_t) & \text{(d) } \Longrightarrow M_{t'} \dot{\cap} (R_t \dot{\cup} M_t^{\complement}) \subseteq \dot{\varnothing} \\ \text{(e) } \Longrightarrow \mathcal{V}_0((rely_t \vee guar_t) \to rely_{t'}) & \text{(f) } \Longrightarrow (R_t^{\complement} \dot{\cup} M_t) \dot{\cap} R_{t'} \subseteq \dot{\varnothing} \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\![\texttt{fork } t';\ \omega_t]\!]\varphi, \Delta} \quad \text{R38}$$

where t is the current thread, $\mathcal{V}_0$ stands for the update $\{\texttt{heap'} := \texttt{heap} \,||\, \texttt{heap} := anon(\texttt{heap}, R_t^{\complement} \dot{\cup} M_t)\}$, $\mathcal{V}_1$ stands for the update $\{\texttt{heap'} := \texttt{heap} \,||\, \texttt{heap} := anon(\texttt{heap}, R_t^{\complement})\}$, and $\mathcal{W}$ stands for the update $\{\texttt{threads} := \texttt{threads} \dot{\cup} \{t'\}\}$; and $(pre_t, rely_t, guar_t, R_t, M_t)$ is a (not necessarily valid) thread specification for t (same for t').

Lemma 3.14. *Rule R38 is sound.*

Proof. Let premiss (a) be valid, then the conclusion follows from Def. 1.11. ◁

Since soundness of Rule R38 does not depend on premisses (b)–(f), we could devise a simpler sound rule with only premiss (a). But the interesting property is the propagation of thread specification validity—otherwise we could not close a proof except for trivial postconditions. that is guaranteed by premisses (b)–(f). Since a proof includes proving validity of the precondition $pre_{t'}$, the 'guarantee' proof obligation for thread t can be relaxed to states in which $pre_{t'}$ holds.

Lemma 3.15. *Let S be a valid thread specification for T.*

If the formulas in premisses (c)–(f) in Rule R38, as well as formulae (3.3) and (3.4) are all valid, then $S \cup (pre_{t'}, rely_{t'}, guar_{t'}, R_{t'}, M_{t'})$ is a valid thread pool for $T \cup t'$.

Proof. Since the precondition $pre_{t'}$ holds in premiss (b) holds, the thread specification is valid in this state. Then, the conjecture follows from Thm. 3.13. ◁

3.4.3 Soundness

We first observe that most rules for the purely sequential DTL Beckert and Bruns [2013] are also sound for the concurrent setting.

Lemma 3.16. *The $\mathcal{C}_{\text{DTL}}$ rules* R1–R36 *are sound w.r.t. CDTL.*

Proof. According to Lemma 2.9, CDTL is a semantical conservative extension of DTL. This means that the soundness result of Beckert and Bruns [2013] also applies to CDTL. ◁

Together with the rules R37 and R38, the aforementioned rules constitute the calculus $\mathcal{C}_{\text{CDTL}}$. We have already observed that R37 cannot be considered sound in general, but only w.r.t. a valid thread specification. The overall (relative) soundness of the calculus depends on the interplay of rules R38 (i.e., showing that the thread specification can be expanded) and R37 (i.e., requiring a valid thread specification to be sound). Thus, proving overall soundness requires a structural analysis over the complete proof tree.

Theorem 3.17 (Relative soundness of $\mathcal{C}_{\text{CDTL}}$)**.** *Let $\Gamma \Longrightarrow \Delta$ be a sequent that is derivable through applications $A_1, \ldots, A_n$ of rule R37. Let for each rule application A_i the thread pool be T_i. If there is a valid thread specification for each T_i, then $\Gamma \Longrightarrow \Delta$ is valid.*

Note that we make no assumption about the location of the modalities within the sequent; they may appear on both sides of the sequent and may be nested.

Proof. The theorem depends on all rules appearing in the proof tree for $\Gamma \Longrightarrow \Delta$ being valid. All rules except R37 are sound without further prerequisites. This follows from Lemmas 3.14 and 3.16. Following Lemma 3.12, rule R37 is only sound if there is a valid thread specifiation for the thread pool appearing in the conclusion. We show by structural induction over the proof tree that all thread pool appearing in a sequent have a valid specification. For most rules, this is trivial since they do not touch the thread pool. (I.e., if thread pool appears in one of the premisses, it is also present in the conclusion.) Rule R38 is the only rule to introduce additional threads to the pool in a premiss. Since the proof tree is closed, it follows from Lemma 3.15 that the new thread pool has a valid specification. ◁

3.4.4 Completeness

We believe that our calculus is (relatively) complete, although we do not provide a formal proof here. An argument in favor is the proof by Prensa Nieto [2002] that the original calculus by Jones [1983] is complete w.r.t. the concurrent program semantics by Owicki and Gries [1976]. We admit that this is more a theoretical argument, while *practical completeness* strongly relies on the precision quality of the provided specifications. A formal proof of completeness will be part of future work.

Chapter 4

Related Work

Temporal reasoning as well as verification of concurrent programs has traditionally been the domain of model checkers. Verification based on model checking is never complete, and sometimes not even sound. For concurrent Java, the tools Java PathFinder [Havelund and Pressburger, 2000] or Bogor [Robby et al., 2006] are available. There are several others for C and derived languages; see [D'Silva et al., 2008] for an overview. Other common techniques to make the behavior of concurrent programs more expectable are permission systems and ownership annotations, that are checked at runtime. Notable examples are built into the Spec# [Barnett et al., 2005] and Dafny [Leino and Müller, 2009] languages.

4.1 Temporal Behavior of Java Programs

Bandera [Corbett et al., 2000] was one of the first projects to aim at software model checking. It is of particular interest that it employs an implementation-aware temporal specification language called Bandera specification language (BSL). The major goal of BSL was to avoid formalisms such as Linear Temporal Logic (LTL), which are deemed to be not comprehesible to software developers. Therefore, a set of particular specification patterns [Dwyer et al., 1999] was selected to form the essential syntactical entities.

Inspired by the Bandera specification language, Trentelman and Huisman [2002] define an extension to Java Modeling Language (JML) with events and temporal properties. The set of permitted expressions is reduced compared to Bandera; particularily, scopes can only be triggered by events. Statements not expressible include, for instance, 'if φ holds, then eventually `m` is called', or $\varphi \to \Diamond call_{\mathtt{m}}$ in LTL, where φ is a state property. State properties are regular JML expressions enriched by the 'enabled' statement providing whether a method invoked in that state would terminate normally. Events in this context are calls to methods and returns from calls (either normal or exceptional). The semantics for an event to 'hold' in a state s_i of a

sequence $\bar{s}$ is that it represents a transition from s_{i-1} to s_i. It is however not clarified in the paper, what is exactly meant by a 'state,' in particular, whether they only consider visible or observed states. There is a runtime checker implementation for this language called temporaljmlc [Hussain and Leavens, 2010]. Wagner [2013] provides a translation from temporalJML style specifications to DTL.

Programs of concrete programing languages like Java are usually reasoned about in a state based manner. There are a few runtime checking approaches that check for trace properties using LTL-like specification [Bartetzko et al., 2001; Stolz and Bodden, 2006]. Hussain and Leavens also check assertions at runtime, but in addition, they use temporalJML as an extension to the JML specification language, that allows to write high level temporal properties, but is not as expressive as LTL.

4.2 Deductive Reasoning About Concurrent Programs

Abrahamson [1979] presents one of the first works on the issues of dynamic logic, combining program analysis with temporal properties, and concurrency. Here an unstructured programming language with parallel composition and explicit labels gives rise to a branching time temporal structure. Trace formulae are implicitly evaluated over all possible traces. They resemble LTL formulae, but modalities may contain path conditions (typically sequences over labels). The paper does not contain formal semantics or a calculus.

Peleg [1987] introduces *Concurrent Dynamic Logic (CDL)*— based on Harel's original notion—where program modalities contain a parallel composition operator $\cap$. The programs here are linear programs; there is no shared memory. As Peleg himself acknowledges "processes of CDL are totally independent and mutually ignorant." For this reason, the formula $\langle \pi_1 \cap \pi_2 \rangle \varphi$ with π_1 and π_2 executed in parallel is just equivalent to $\langle \pi_1 \rangle \varphi \wedge \langle \pi_2 \rangle \varphi$.

The book by de Roever et al. [2001] provides a good overview over early (both compositional and noncompositional) approaches to verification of shared memory concurrent programs.

A closely related work is [Schellhorn et al., 2011] that Interval Temporal Logic (ITL) [Cau et al., 2002] with interleaved programs and higher order logic. They present a calculus based on symbolic execution and rely/guarantee, that is implemented in the Karlsruhe Interactive Verifier (KIV) theorem prover.

Another approach using a dynamic logic—named *multi-threaded object-oriented dynamic logic (MODL)*—is taken by Klebanov [2009]; Beckert and Klebanov [2013], that uses a realistic Java-like programing language and explicitly constructs interleaved programs. Concurrent programs are composed sequentially into a single program with multiple program pointers. During

symbolic execution of threads, these pointers are moved in the (unmodified) program code. This is different to our dynamic logic where we consider sequential programs executed by some thread; and program statements are deleted and the one program pointer is implicitly at the beginning of the remainder. The (deterministic) scheduler is explicitly axiomatized in MODL. They use a Java-like language, but impose the rather strong requirement that all loops are atomic. It also includes atomic blocks that are symbolicly executed in another kind of DL modality. Like our calculus, theirs is implemented in the KeY system, too. Through the vast possibilities of interleaved executions—it can be seen as an instance of what de Roever et al. [2001] call the *global method* for concurrency verification—this approach suffers from a high complexity (that however can be reduced in parts by making stronger assumptions about the scheduling process).

4.3 Rely/Guarantee

Prensa Nieto [2002] presents the first thorough formalization of rely/guarantee that can be machine-checked (in Isabelle/higher order logic (HOL) [Paulson, 1994]). "Surprisingly, it appears that there has been no work on embedding Hoare logics for shared-variable parallelism in any theorem prover."

Ahrendt and Dylla [2009, 2012] describe a verification system for concurrent programs written in the Creol language. Creol [Johnsen et al., 2006] is an experimental object-oriented language that features different kinds of concurrency. On an outer layer, it features *distributed objects* (i.e., distributed components that are class instances), which code execute truly in parallel. Distributed objects communicate through asynchronous message passing. But intra-object execution is multi-threaded with a shared memory. Ahrendt and Dylla apply a rely/guarantee approach to reason about this kind of concurrency. An important difference to our work is that Creol follows a *coöperative scheduling* philosopy [Dovland et al., 2005], in which sequential executions are not arbitrarily interleaved, but threads actively release control at explicit *release points* programmatically. Their semantics are based on the technique by Zwiers [1989] to construct histories of interactions by nondeterministically 'guessing' environment actions. These interactions include 'yield' and 'resume' events that capture the memory state upon a release, thus bearing a similarity to our state traces.

Ahrendt and Dylla present a symbolic execution calculus for a dynamic logic for Creol with an implementation in an experimental version of KeY. The calculus rule dealing with release uses a special kind of update that anonymizes the state, through a (deterministic) ε assignment [Hilbert and Bernays, 1939], such that the rely condition holds in this state.

The Abstract Behavioral Specification (ABS) language [Johnsen et al., 2010; Hähnle et al., 2011] borrows many concepts, in particular regarding

concurrency, from Creol. Din et al. [2012]; Din [2014] describe a verification system similar to the one of Ahrendt and Dylla [2009] and implementation in KeY. Threads in both Creol and ABS cannot be created dynamically. The scope of their shared memory is restricted to class boundaries.

Recently, rely/guarantee has been often considered in combination with separation logic [Vafeiadis and Parkinson, 2007] since it does not only provide functional specifications for threads, but also separates the memory on which threads work. The central idea is similar to our approach to frame possible write effects. The main difference is that we make our frame annotations explicit, while in separation logic it is enshrined in the 'star' operator. Dodds, Feng, Parkinson, and Vafeiadis [2009] discuss the inability of rely/guarantee to specify the behavior of programs that use forking and joining of threads, as opposed to parallel decomposition (on which rely/guarantee is traditionally defined). Forking and synchronization (of which joining is a special case) are the mechanisms used in real world programing languages. With those, the lifetime of a thread is controlled dynamically. To deal with programs of this shape, they propose an approach called "deny/guarantee," building on the work by Vafeiadis and Parkinson, in which nonbehavior of environments is specified using separation logic. These specifications are also dynamic, unlike the staticly defined invariants of rely/guarantee.

Haack and Hurlin [2008] also present a separation logic calculus for fork/join concurrency, modeling the multi-threading architecture of Java. It is based on fractional permissions [Boyland, 2003].

[Smans et al., 2014] extend concurrent separation logic by introducing "shared boxes," that encapsulate shared variables with a two-state invariant. This invariant must hold whenever a thread accesses the shared variable. This is to be checked in VeriFast.

Bibliography

Karl R. Abrahamson. Modal logic of concurrent nondeterministic programs. In Gilles Kahn, editor, *Semantics of Concurrent Computation*, volume 70 of *Lecture Notes in Computer Science*, pages 21–33. Springer, 1979. ISBN 3-540-09511-X. URL `http://dx.doi.org/10.1007/BFb0022461`.

Wolfgang Ahrendt and Maximilian Dylla. A verification system for distributed objects with asynchronous method calls. In Karin Breitman and Ana Cavalcanti, editors, *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*, volume 5885 of *Lecture Notes in Computer Science*, pages 387–406. Springer, 2009. ISBN 978-3-642-10372-8.

Wolfgang Ahrendt and Maximilian Dylla. A system for compositional verification of asynchronous objects. *Science of Computer Programing*, 77(12): 1289–1309, 2012.

Edward A. Ashcroft. Proving assertions about parallel programs. *J. Comp. Sys. Sci.*, 10:110–135, 1975.

Edward A. Ashcroft and Zohar Manna. Formalization of properties of parallel programms. *Machine Intelligence*, 6:17–41, 1971.

Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: an overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2005. URL `http://www.springerlink.com/openurl.asp?genre=article&issn=0302-9743&volume=3362&spage=151`.

Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass – Java with assertions. *Electr. Notes Theor. Comput. Sci*, 55(2):103–117, 2001. URL `http://dx.doi.org/10.1016/S1571-0661(04)00247-6`.

Bernhard Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, volume 2041 of *Lecture Notes in Computer Science*, pages 6–24. Springer, 2001.

Bernhard Beckert and Daniel Bruns. Dynamic trace logic: Definition and proofs. Technical Report 2012-10, Department of Informatics, Karlsruhe Institute of Technology, 2012. URL `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000028184`. A revised version replacing an unsound rule is available at `http://formal.iti.kit.edu/~bruns/papers/trace-tr.pdf`.

Bernhard Beckert and Daniel Bruns. Dynamic logic with trace semantics. In Maria Paola Bonacina, editor, *24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Computer Science*, pages 315–329. Springer-Verlag, 2013. ISBN 978-3-642-38573-5. doi: 10.1007/978-3-642-38574-2_22. URL `http://link.springer.com/chapter/10.1007/978-3-642-38574-2_22`.

Bernhard Beckert and Vladimir Klebanov. A dynamic logic for deductive verification of multi-threaded programs. *Formal Aspects of Computing*, 25 (3):405–437, 2013. ISSN 0934-5043.

Bernhard Beckert and Steffen Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In R. Goré, A. Leitsch, and T. Nipkow, editors, *Proceedings, International Joint Conference on Automated Reasoning, Siena, Italy*, volume 2083 of *Lecture Notes in Computer Science*, pages 626–641. Springer, 2001.

Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2007a.

Bernhard Beckert, Vladimir Klebanov, and Steffen Schlager. Dynamic logic. In Beckert et al. [2007a], chapter 3, pages 69–178.

Bernhard Beckert, Daniel Bruns, Vladimir Klebanov, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. Secure information flow for Java – a dynamic logic approach. Technical Report 2013-10, Department of Informatics, Karlsruhe Institute of Technology, 2013. URL `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000036786`.

Alexander Borgida, John Mylopoulos, and Raymond Reiter. "...and nothing else changes": The frame problem in procedure specifications. In Victor R. Basili, Richard A. DeMillo, and Takuya Katayama, editors, *Proceedings of the 15th International Conference on Software Engineering, Baltimore,*

Maryland, USA, May 17-21, 1993, pages 303–314. IEEE Computer Society / ACM Press, 1993. ISBN 0-89791-588-7. URL `http://dl.acm.org/citation.cfm?id=257572`.

John Boyland. Checking interference with fractional permissions. In Radhia Cousot, editor, *Static Analysis (SAS)*, volume 2694 of *Lecture Notes in Computer Science*, pages 55–72, Berlin, 2003. Springer-Verlag. URL `http://dx.doi.org/10.1007/3-540-44898-5_4`.

Antonio Cau, Ben Moszkowski, and Hussein Zedan. Interval temporal logic, September 23 2002. URL `http://www.cse.dmu.ac.uk/~cau/papers/itlhomepage.pdf`.

James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, New York, NY, June 2000. ACM Press.

Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

Crystal Chang Din. *Verification Of Asynchronously Communicating Objects*. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, March 2014.

Crystal Chang Din, Johan Dovland, Einar Broch Johnsen, and Olaf Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *Journal of Logic and Algebraic Programming*, 81(3): 227–256, 2012.

Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In G. Castagna, editor, *Programming Languages and Systems*, number 5502 in Lecture Notes in Computer Science, pages 363–377. Springer-Verlag, 2009.

Johan Dovland, Einar Broch Johnsen, and Olaf Owe. Verification of concurrent objects with asynchronous method calls. In *International Conference on Software – Science, Technology and Engineering. SwSTE '05*, pages 141–150. IEEE Computer Society, 2005. ISBN 0-7695-2335-8. URL `http://doi.ieeecomputersociety.org/10.1109/SWSTE.2005.24`.

Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on*

CAD of Integrated Circuits and Systems, 27(7):1165–1178, 2008. URL http://dx.doi.org/10.1109/TCAD.2008.923410.

Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 411–420. IEEE Computer Society Press / ACM Press, 1999. URL http://www.acm.org/pubs/articles/proceedings/soft/302405/p411-dwyer/p411-dwyer.pdf.

Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite model theory*, volume 2. Springer, 1995.

David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 366–373. Springer-Verlag, New York, NY, 1995.

Christian Haack and Clément Hurlin. Separation logic contracts for a Java-like language with fork/join. In José Meseguer and Grigore Roşu, editors, *Algebraic Methodology and Software Technology, 12th International Conference, AMAST 2008, Urbana, IL, USA, July 28-31, 2008, Proceedings*, volume 5140 of *Lecture Notes in Computer Science*, pages 199–215. Springer, 2008. ISBN 978-3-540-79979-5.

Reiner Hähnle. Many-valued logic, partiality, and abstraction in formal specification languages. *Logic Journal of the IPGL*, 13(4):415–433, July 2005.

Reiner Hähnle, Michiel Helvensteijn, Einar Broch Johnsen, Michael Lienhardt, Davide Sangiorgi, Ina Schaefer, and Peter Y. H. Wong. HATS Abstract Behavioral Specification: The architectural view. In Bernhard Beckert, Ferruccio Damiani, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects 2011*, volume 7542 of *Lecture Notes in Computer Science*, pages 109–132. Springer, 2011. ISBN 978-3-642-35886-9; 978-3-642-35887-6.

David Harel. *First-order dynamic logic*, volume 68 of *Lecture notes in computer science*. Springer-Verlag, New York, 1979.

Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA pathfinder. *STTT*, 2(4):366–381, 2000. URL http://dx.doi.org/10.1007/s100090050043.

David Hilbert and Paul Bernays. *Die Grundlagen der Mathematik II*. Springer-Verlag, Berlin, 1939.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1978.

Faraz Hussain and Gary T. Leavens. temporaljmlc: A JML runtime assertion checker extension for specification and checking of temporal properties. Technical Report CS-TR-10-08, UCF, Dept. of EECS, Orlando, Florida, July 2010.

Bart Jacobs and Jan Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of the European Association for Theoretical Computer Science*, 62:222–259, 1997.

Jean-Baptiste Jeannin and André Platzer. dTL^2: Differential temporal dynamic logic with nested modalities for hybrid systems. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Int. Joint Conference on Automated Reasoning 2014*, volume tba of *LNCS*. Springer, 2014.

Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1-2):23–66, 2006.

Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Formal Methods for Components and Objects 2010*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2010. ISBN 978-3-642-25270-9.

Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5 (4):596–619, 1983. URL `http://doi.acm.org/10.1145/69575.69577`.

Ioannis T. Kassios. The dynamic frames theory. *Formal Aspects Computing*, 23(3):267–288, 2011. URL `http://dx.doi.org/10.1007/s00165-010-0152-5`.

Robert M. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

Vladimir Klebanov. *Extending the Reach and Power of Deductive Program Verification*. PhD thesis, Universität Koblenz, 2009.

Saul Kripke. Semantical considerations on modal logic. *Acta philosphica fennica*, 16:83–94, 1963.

Leslie Lamport. The ‘Hoare logic’ of concurrent programs. *Acta Informatica*, 14:21–37, 1980.

K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning, 16th International Conference, LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer-Verlag, 2010.

K. Rustan M. Leino and Peter Müller. A basis for verifying multi-threaded programs. In Giuseppe Castagna, editor, *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*, pages 378–393, Berlin, March 2009. Springer-Verlag.

K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010*, volume 6015 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2010. ISBN 978-3-642-12001-5.

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. The Java Series. Addison-Wesley, Boston, Mass., May 2014.

John McCarthy. Towards a mathematical science of computation. In *Information Processing ’62*, pages 21–28, Amsterdam, 1962. North-Holland.

Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981.

Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

David Peleg. Concurrent dynamic logic. *Journal of the ACM*, 34(2):450–479, April 1987.

André Platzer. A temporal dynamic logic for verifying hybrid system invariants. In Sergei N. Artëmov and Anil Nerode, editors, *Logical Foundations of Computer Science, International Symposium, LFCS 2007, New York, NY, USA, June 4-7, 2007, Proceedings*, volume 4514 of *Lecture Notes in Computer Science*, pages 457–471. Springer, 2007. ISBN 978-3-540-72732-3. URL `http://dx.doi.org/10.1007/978-3-540-72734-7_32`.

Arndt Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *Programming Languages and Systems (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999.

Leonor Prensa Nieto. *Verification of parallel programs with the Owicki-Gries and rely-guarantee methods in Isabelle/HOL*. PhD thesis, Technische Universität München, 2002. URL `http://mediatum.ub.tum.de/doc/601717/601717.pdf`.

Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: A flexible framework for creating software model checkers. In Phil McMinn, editor, *Testing: Academia and Industry Conference; Practice And Research Techniques (TAIC PART), Windsor, United Kingdom*, pages 3–22. IEEE Computer Society, 2006.

Philipp Rümmer. Sequential, parallel, and quantified updates of first-order structures. In Miki Hermann and Andrei Voronkov, editors, *Proc. Logic for Programming, Artificial Intelligence and Reasoning, Phnom Penh, Cambodia*, volume 4246 of *LNCS*, pages 422–436. Springer-Verlag, 2006.

Christoph Scheben and Peter H. Schmitt. Verification of information flow properties of JAVA programs without approximations. In Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov, editors, *Formal Verification of Object-Oriented Software International Conference, FoVeOOS 2011, Revised Selected Papers*, volume 7421 of *Lecture Notes in Computer Science*, pages 232–249. Springer, 2012.

Gerhard Schellhorn, Bogdan Tofan, Gidon Ernst, and Wolfgang Reif. Interleaved programs and rely-guarantee reasoning with ITL. In Carlo Combi, Martin Leucker, and Frank Wolter, editors, *Eighteenth International Symposium on Temporal Representation and Reasoning, TIME 2011, Lübeck, Germany, September 12-14, 2011*, pages 99–106. IEEE, 2011. ISBN 978-1-4577-1242-5. URL `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6063703`.

Jan Smans, Bart Jacobs, Frank Piessens, and Wolfram Schulte. An automatic verifier for Java-like programs based on dynamic frames. In *Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*, pages 261–275, Berlin, April 2008. Springer-Verlag. doi: 10.1007/978-3-540-78743-3_19. URL `https://lirias.kuleuven.be/handle/123456789/178243`.

Jan Smans, Dries Vanoverberghe, Dominique Devriese, Bart Jacobs, and Frank Piessens. Shared boxes: Rely-guarantee reasoning in VeriFast. Technical Report CW 662, KU Leuven, Department of Computer Science, May

2014. URL http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW662.pdf.

Robert F. Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: definition, verification, validation*. Springer-Verlag, 2001. ISBN 3-540-42088-6.

Kurt Stenzel. *Verification of Java Card Programs*. PhD thesis, Fakultät für angewandte Informatik, University of Augsburg, 2005. URL http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/dissertations/2005_stenzel_diss/diss-stenzel.pdf.

Colin Stirling. A generalization of Owicki-Gries's Hoare logic for a concurrent while language. *Theoretical Computer Science*, 58(1–3):347–359, July 1988.

Ketil Stølen. A method for the development of totally correct shared-state parallel programs. In J. C. M. Baeten and J. F. Groote, editors, *CONCUR '91: 2nd International Conference on Concurrency Theory*, volume 527 of *Lecture Notes in Computer Science*, pages 510–525, Amsterdam, The Netherlands, August 1991. Springer-Verlag.

Volker Stolz and Eric Bodden. Temporal assertions using aspectJ. *Electr. Notes Theor. Comput. Sci*, 144(4):109–124, 2006. URL http://dx.doi.org/10.1016/j.entcs.2006.02.007.

Kerry Trentelman and Marieke Huisman. Extending JML specifications with temporal logic. In Hélène Kirchner and Christophe Ringeissen, editors, *Algebraic Methodology and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, September 9-13, 2002, Proceedings*, volume 2422 of *Lecture Notes in Computer Science*, pages 334–348. Springer, 2002. ISBN 3-540-44144-1. URL http://link.springer.de/link/service/series/0558/bibs/2422/24220334.htm.

Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *Concurrency Theory, 18th International Conference, CONCUR 2007*, volume 4703 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2007. ISBN 978-3-540-74406-1.

Andreas Wagner. Trace based reasoning with KeY and JML. Studienarbeit, Karlsruhe Institute of Technology, 2013. URL http://www.key-project.org/DeduSec/2013_Wagner_TraceSpecification.pdf.

Benjamin Weiß. *Deductive Verification of Object-oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, Karlsruhe Institute of Technology, January 2011. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/1600837.

Qiwen Xu, Willem-Paul de Roever, and Jifeng He. The rely–guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

Ernst Zermelo. Beweis, daß jede Menge wohlgeordnet werden kann. *Mathematische Annalen*, 59(4):514–516, 1904.

Job Zwiers. *Compositionality, Concurrency and Partial Correctness*, volume 321 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.