Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2014,5
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Towards Specification and Verification of Information Flow in Concurrent Java-like Programs

Daniel Bruns

2014

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.

Towards Specification and Verification of Information Flow in Concurrent Java-like Programs*

Daniel Bruns

March 18, 2014

Abstract

Today, nearly all personal computer systems are multiprocessor systems, allowing multiple programs to be executed in parallel while accessing a shared memory. At the same time, computers are increasingly handling more and more data. Ensuring that no confidential data is leaked to untrusted sinks remains a major quest in software engineering. Formal verification, based on theorem proving, is a powerful technique to prove both functional correctness and security properties such as absence of information flow. Present verification systems are sound and complete, but often lack efficiency and are only applicable to sequential programs. Concurrent programs are notorious for all possible environment actions have to be taken into account.

In this paper, we point out steps towards a formal verification methodology for information flow in concurrent programs. We consider passive attackers who are able to observe public parts of the memory at any time during execution and to compare program traces between different executions. Our approach consists of two main elements: 1. formalizing properties on whether two executions are indistinguishable to such an attacker in a decidable logic, and 2. defining a technique to reason about executions of concurrent programs by regarding a single thread in isolation. The latter is based on the rely/guarantee approach.

*This work has been supported by Deutsche Forschungsgemeinschaft (DFG) under project "Program-level Specification and Deductive Verification of Security Properties (DeduSec)" within SPP 1496 "Reliably Secure Software Systems (RS^3)".

Contents

1 Introduction **5**

2 Target Programing Language **7**
2.1 Scheduler Assumptions 8
2.2 Outlook: Adapting to actual Java 8

3 Logical Foundations **9**
3.1 Trace based Program Semantics 9
3.2 Dynamic Trace Logic 10
3.3 Explicit Heaps 12
3.4 Base Calculus for DTL 12
3.5 A Theory of Partial Maps 14

4 Security Properties **14**
4.1 Stutter Tolerant Noninterference 15

5 Reasoning about Information Flows in DTL **15**
5.1 Extending DTL with Information Flow Operators 16
5.2 Formalizing Noninterference 17
5.3 Local Reasoning about Flows 18

6 Rely/Guarantee Reasoning **18**
6.1 Definitions and Notations 20
6.2 Guarantees 21
6.3 Rely Component: New Symbolic Execution Rules 22
6.4 Concurrent Thread Specification in JML 23
6.5 Example 24

7 Related Work **24**
7.1 Deductive Reasoning About Concurrent Programs 26
7.2 Temporal Behavior of Java Programs 26
7.3 Information Flow Analysis Based on Theorem Proving 26

A A Theory of Partial Mappings **28**

1 Introduction

Confidentiality is the subarea of (information) security that concerned with restricting unqualified access to confidential information. More abstractly, the goal is to ensure that no data must flow from confidential (i.e., high security level) sources to public (i.e., low security level) sinks. These security levels can be connected in arbitrary complex lattices, but for theoretical considerations it suffices to regard just two levels 'low' and 'high.' Language based information flow security [Sabelfeld and Myers, 2003] covers the scenario where information is handled by software which is known to attackers. It is assumed that an attacker knows all vulnerabilities and how to exploit them. The analysis therefore focuses on the program source (the 'language') alone.

Recently, theorem proving approaches to language based information flow analysis [Joshi and Leino, 2000, Amtoft and Banerjee, 2004, Darvas et al., 2005] have gained prominence. These are based on a semantical notion of information flow and therefore bear the advantage of semantical precision over established static techniques like type checking. Some program logics such as dynamic logic are—unlike Hoare logic [Hoare, 1972]—readily able to express relational properties like information flow.[1] And at the same time, formal verification of functional properties about software has made great progress in the past years. In particular, the KeY prover [Beckert et al., 2007], codeveloped by the author, for first order dynamic logic is able to formally verify information flow properties about *sequential* Java programs [Scheben and Schmitt, 2012]. One aim of this paper is to point out a first step on how these techniques can be lifted to reasoning about *concurrent* programs. In Sect. 4 our security property is defined and in Sect. 5 we show how it can be formalized and how to reason about it.

The setting which we investigate is where a sequential program (i.e., thread) π in a deterministic language[2] runs on shared memory with a possibly hostile environment. The control and information flow of thread π can be influenced by the environment. We use the rely/guarantee approach to restrict environment changes. In Sect. 6, we describe how guarantee conditions can be verified in the KeY prover.[3] This allows us to regard π, instrumented with havoc statements, in isolation. The attacker is able to observe low locations *at any time* and to observe the order of changes. This is different from the purely sequential setting where the attacker can only observe initial and final values. She is not able to mount timing attacks (w.r.t. wall clock time). Thread local information and control flows are expected to be instantaneous and are not observable.

We allow confidential information to be declassified. Since theorem proving approaches are founded semantically, precise subject declassification (i.e., which information is released) already comes for free. In this paper, we additionally

[1] Some authors refer to relational properties as "hyper-properties" [Clarkson and Schneider, 2010].

[2] Many models of concurrent program executions regard programs as indeterministic. We deliberately do not follow this paradigm. The reason is that real world programing languages are deterministic. Scheduling may depend on unknown parts of the system state. We prefer to model this through *underspecification* [Hähnle, 2005], which allows us to talk about exactly one program trace. This approach is also taken in [Beckert and Klebanov, 2013].

[3] In many situations, there may be even stronger guarantee conditions like perfect separation which could be checked with other methodologies, like type checking or runtime checking, which are less precise but more efficient.

consider *timing* of declassification. Just like subject declassification can be expressed as a relational property between states, temporal declassification can be expressed as a relation between traces. In our logic, we formalize this through temporal operators. Controlling the temporal dimension of declassification is essential in state based software systems. Consider, for instance, an electronic voting system, which has different declassification policies before and after the election has been closed: only afterwards the result (i.e., the sum of votes) may be published.

In our previous work [Beckert and Bruns, 2013], we have presented a logic called Dynamic Trace Logic (DTL), which combines typed first order dynamic logic with temporal logic. While standard dynamic logic [Harel, 1979] is able to express relationships between initial and final states of a program execution. The semantics of DTL is based on (finite or infinite) *traces* of program states. This allows us to state temporal properties about program executions, e.g., the formula $\phi_1 \rightarrow [\![\pi]\!]\Diamond\Box\phi_2$ intuitively means 'for a program π started in a state where ϕ_1 holds, it eventually reaches a state from which on ϕ_2 holds always'. DTL features a trace modality $[\![\cdot]\!]$ and well-known temporal operators $\Box$ (throughout), $\Diamond$ (eventually), $\bullet$ (weak next), $\circ$ (strong next), and $\mathbf{U}$ (until) similar to those of Linear Temporal Logic [Manna and Pnueli, 1995]. We have also given a sound and complete calculus for DTL which has been prototypically implemented in the KeY prover.

In this paper, we will use DTL (plus first order definable theories) to express information flow properties, in particular noninterference with declassification, for single threads of concurrent programs as explained above. We benefit from the strength of our logic being able to formalize relational properties about programs readily. A formalization of noninterference for sequential programs in dynamic logic has already been presented [Scheben and Schmitt, 2012]. One disadvantage of this approach is that proofs tend to get large and practically infeasible. For that reason—as a kind of complementary approach—we add dedicated information flow (temporal) operators to DTL and provide calculus rules which allow us to reason about information flow more modularly in particular cases.

As mentioned above, there is a prototype implementation in the KeY prover. Since KeY has been designed for the verification of (sequential) Java programs, there is relatively little overhead in lifting the techniques presented in this paper from the toy language to Java. In principle, an extension to the calculus presented in this paper should be doable without much effort. We expect, however, that several optimizations will be necessary in order to efficiently prove information flow properties about nontrivial programs.

Examples. Consider the one-liner programs in Listing 1, written in a simple imperative language. `L` is a low global variable, `H` etc. are high global variables, and `x` is a local variable. Let us so far assume that `L` is not written by concurrently running threads. The first four programs are both secure in purely sequential execution as well as in a concurrent environment since, in any state, the value of `L` does not depend on the initial value of `H`. However, to *prove* that these programs are secure can only be achieved with precise program semantics.

In addition, Program 5 is considered secure in the sequential setting, but not in the concurrent one: There may be an interleaving between the two assign-

```
1 { x = H; x = 23; L = x; }
2 { L = H * 0; }
3 { H = L; L = H; }
4 { if (H = 0) { L = L + H; } else {} }
5 { L = H; L = 23; }
6 { if (H0 > 0) { H1 = L; H2 = H1; } else {} }
7 { while (H > 0) { L = L; } }
8 { while (H > 0) { L = L +1; } }
```

Listing 1: Example programs

ments in which the confidential information (temporally) stored in `L` is leaked to other threads.

Programs 6–8 possibly have different run time on low equivalent runs. They are thus insecure if traces are compared componentwise. But, since the traces only differ in high component values, Programs 6 and 7 are secure if stuttering is tolerated. In Program 8, the value of `L` obviously depends on high values; even with stuttering, it is insecure. Finally, if we lift the restriction that `L` must not be modified concurrently, neither program can be proven secure. The reason is that other threads could write secret information at any time, in particular at the end of these programs.

2 Target Programing Language

For this paper we consider a simple imperative language which 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 in parallel. Other Java features such as objects, types, or exceptions are not of relevance to our discourse and could be added without invalidating the central results. Synchronization and dynamic thread creation are also not considered at the moment and will be left to future work.

Our core language consists of assignments, conditional branching, and conditional loop 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 and boolean; they do not have side effects. Integer operators are unary minus, addition, and multiplication—no division or 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 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.[5] Expressions are called *simple* if they do not contain global variables.

[4] Throughout this paper, we will use the term 'program' for sequential program fragments (or, 'blocks' in Java).

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

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 varible.

Core language programs can be instrumented with the special statement `env;` which represents environment actions. Programs not containing `env;` are called noninterleaved.

Definition 1 (Sequential program syntax)**.** *A* sequential program*, or just 'program' for short, is a sequence of statements. 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`}` *where* b *is a simple boolean expression and* π *is a program,*
- **loop***:* `while (`b`) {`π`}` *where* b *is a simple boolean expression and* π *is a program containing at least one global assignment,*
- **environment action***:* `env;`

2.1 Scheduler Assumptions

Our approach is widely scheduler agnostic. A formalization of scheduler policies is not part of this paper. We only make some basic assumptions:

- In any state, the scheduler selects an active thread, i.e., a thread which is not yet terminated.[6]
- The scheduler is fair. Otherwise we cannot guarantee termination. By 'fair' we understand the property that every thread will be chosen sufficiently often to terminate or infinitely often.
- The scheduler does not change the global heap state. It may have its own internal state, however.

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

2.2 Outlook: Adapting to actual Java

In comparison to the simplified language we use throughout this paper, the Java actually contains some additional caveats.

[6]Thread suspension is not yet considered.

Nonatomic compound statements are allowed and very common in Java. The assignment statement `F = G;` where both `F` and `G` are fields contains a read action first and then a write action. This one-liner is functionally equivalent to the fragment `x = G; F = x;` using a local variable for intermediate storage, where both statements are atomic.[7] There can be an arbitrary number of read and write actions in any expression. For instance, the statement `F /= G++ * --H;` contains three read and write actions each. The final write to `F` does not occur if there is a division by zero. It can be rewritten into

```
x = F; y = G; G = y+1; z = H-1; H = z; w = x/(y*z); F = w;
```

The exact order of evaluation is crucial here. The operands are evaluated from left to right; if the evaluation has side effects, then these apply; finally the operation is applied. The division operation itself is also not atomic—even though it only operates on local variables—but it may include raising an exception, which entails further actions on the heap. The symbolic execution embedded in Java Dynamic Logic performs these kinds of transformations lazily and produces a normal form.

However, this not quite enough for reasoning about concurrent programs. A particular issue is that in Java, non-volatile fields of type `long` (64bit integer) are not written in one atomic step, but in one for each 32bit half (cf. [Gosling et al., 2013, Sect. 17.7]). Writing to a `long` field `L` can thus be imagined of as applying two consecutive atomic writes:

```
L = x & 0xFFFFFFFF00000000L + L & 0x00000000FFFFFFFFL;
L = L & 0xFFFFFFFF00000000L + x & 0x00000000FFFFFFFFL;
```

The Java Memory Model (JMM) specifies that concurrent writes may not be immediately visible to other threads. This means that in Java sequential consistency of the memory is only given if the threads do not compete in data races. An analysis for data race freedom can, for instance, be found in [Klebanov, 2009]. In this paper, we assume that all writes to the global memory are immediately visible.

3 Logical Foundations

We start off with Dynamic Trace Logic (DTL) as described in [Beckert and Bruns, 2013], but replace the concept of global program variables by an explicit heap representation. In Sect. 3.1, we provide semantics based on traces of program states and an explicit heap representation. In addition, we use a (first order) theory of partial maps. Later in Sect. 5.1, we add explicit temporal operators for information flow.

3.1 Trace based Program Semantics

Expressions and formulae are evaluated over traces of states (which give meaning to program variables) and variable assignments (which give meaning to logical variables). Global memory is modelled using an explicit (ghost) program variable `heap`. The semantics of `heap` is a mapping from variable names to values.

[7] Except for the case where we have 64bit data types, see below.

Program semantics with explicit heap representations have been used in [Weiß, 2011], for instance. We use another variable `heap'` to denote the heap in the previous state.

Definition 2 (States, variable assignments)**.** *A* state s *is a function assigning integer values to all (local) program variables, including the special program variables* `heap` *and* `heap'`*. A* variable assignment β *is a function assigning values to all logical variables.*

We use the notation $s\{v \mapsto d\}$ to denote the state that is identical to s except that the local program variable v is assigned the value d. Likewise, we write $\beta\{x \mapsto d\}$. For global program variables, the special variable `heap` is updated to a new function using the (higher order) function *store*, see Sect. 3.3 below.

Definition 3 (Traces)**.** *A* trace τ *is a nonempty, finite or infinite sequence of (not necessarily different) states.*

We use the following notations related to traces: (i) $|\tau| \in \mathbb{N} \cup \{\infty\}$ is the *length* of a trace τ. (ii) $\tau_1 \cdot \tau_2$ is the *concatenation* of traces. (iii) $\tau[i, j)$ 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. Given a state s and a variable assignment β, the value of expression a is given by $a^{s,\beta}$ or just a^s. All functions have their standard interpretation.

The following defintion is close to [Beckert and Bruns, 2013], it gives the semantics of sequential noninterleaved programs. The difference is that we now have the special variables `heap` and `heap'` representing the current and the previous heap. In Sect. 6, we will extend this definition with environment actions.

Definition 4 (Trace of a noninterleaved program)**.** *Given an (initial) state s, the* trace of a program π*, denoted $trc(s, \pi)$, is defined by (the smallest fixpoint of):*

$$
\begin{aligned}
trc(s, \epsilon) &= \langle s \rangle \\
trc(s, \texttt{v = a; } \omega) &= trc(s\{v \mapsto a^s\}, \omega) \\
trc(s, \texttt{G = a; } \omega) &= \langle s \rangle \cdot trc(s\{\texttt{heap'} \mapsto \texttt{heap}^s, \texttt{heap} \mapsto store(\texttt{heap}^s, \texttt{G}, a^s)\}, \omega)
\end{aligned}
$$

$$
trc(s, \texttt{if (a) \{}\pi_1\texttt{\} else \{}\pi_2\texttt{\} } \omega) = \begin{cases} trc(s, \pi_1\ \omega) & \textit{if } s \vDash a \\ trc(s, \pi_2\ \omega) & \textit{if } s \nvDash a \end{cases}
$$

$$
trc(s, \texttt{while (a) \{}\pi\texttt{\} } \omega) = \begin{cases} trc(s, \pi\ \texttt{while (a) \{}\pi\texttt{\} } \omega) & \textit{if } s \vDash a \\ trc(s, \omega) & \textit{if } s \nvDash a \end{cases}
$$

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 information flow properties, it is important that not too many irrelevant intermediate states are included in a trace.

3.2 Dynamic Trace Logic

Dynamic logics (DL) [Harel, 1979] are multimodal first order logics where each legal sequential program fragment π (i.e., a sequence of statements) gives rise to modal operators $[\pi]$ and $\langle \pi \rangle$. The formula $[\pi]\phi$ expresses 'in any state in which π terminates, ϕ holds,' while the dual $\langle \pi \rangle \phi$ expresses 'there is a state in

which π terminates and ϕ holds in that one.' If programs are deterministic—i.e., there is at most one final state—the modality $\langle\cdot\rangle$ is a variant of $[\cdot]$ which demands termination. Programs in languages like Java are deterministic in the sense that, under some assumptions about the environment (e.g., the presence of unlimited memory), the program represents a function from one system state to another. Program logics like DL are more expressive than Hoare logics in that programs are part of formulae, which can be selfcomposed. This readily allows, for instance, to express information flow properties such as non interference, cf. [Scheben and Schmitt, 2012]. In other regards, however, classical dynamic logic lacks expressiveness: The semantics of a program is a relation between two states; formulae can only describe the input/output behaviour of programs. Standard dynamic logic cannot be used to reason about program behavior not manifested in the input/output relation. It is inadequate for reasoning about nonterminating programs and for verifying temporal properties.

Dynamic Trace Logic [Beckert and Bruns, 2013] is variant of dynamic logic with only one program modal operator, called *trace modality* $[\![\cdot]\!]$. We distinguish between state formulae and trace formulae. A *state formula* consists of the usual propositional and first order constructs plus subformulae of the form $\mathcal{U}[\![\pi]\!]\phi$ where $\mathcal{U}$ is a sequence of *updates* (see below), π is a program, and ϕ is a *trace formula* (that may contain temporal operators and further subformulae of the same form). Intuitively, $\mathcal{U}[\![\pi]\!]\phi$ 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. Since we have deterministic programs, traces are determined by their initial states. However, states are symbolic (i.e., possibly underspecified), and thus are traces.

In addition to propositional operators and quantification, trace formulae may contain temporal operators similar those in LTL: unary operators $\Box$ ('always') and $\Diamond$ ('eventually'), and binary operators $\mathbf{U}$ ('until'), $\mathbf{W}$ ('weak until'), and $\mathbf{R}$ ('release') with the obvious semantics. Since traces may be finite or infinite, there are weak ($\bullet$) and strong 'next' ($\circ$) operators, which are duals to each other. For example, the formula $\bullet \mathit{false}$ (with 'weak next') holds exactly in the final state of a trace. A formula is called *nontemporal* if it neither contains a temporal operator nor a program modality $[\![\pi]\!]$.

Standard dynamic logic is covered by DTL because the semantics of the standard $[\cdot]$ and $\langle\cdot\rangle$ modalities can be expressed in DTL: The formula $\bullet \mathit{false}$ holds exactly on a trace with only one (remaining) state, thus characterizing termination. We are then able to represent $[\pi]\phi$ by $[\![\pi]\!]\Box(\bullet \mathit{false} \rightarrow \phi)$ and $\langle\pi\rangle\phi$ by $[\![\pi]\!]\Diamond(\bullet \mathit{false} \wedge \phi)$. Yet, both can still coexist and we will use the $[\cdot]$ notation later.

An important property of the calculus for DTL is that programs are *symbolically executed* starting from an initial state—in contrast to *wp* calculi where one starts with a postcondition and works in a backwards manner. In order to capture the state transitions in between, we use state updates [Rümmer, 2006]. Updates can be thought of as 'delayed substitutions,' i.e., a substitution takes place once the program has been completely eliminated. For instance, $\{v := 4\}$ and $\{v := v + 1\}$ are updates. Applying these updates (after each other, from right to left) to the formula[8] $v \doteq 5$ yields $4 + 1 \doteq 5$. We will not go into much more detail; a complete calculus for updates can be found in [Rümmer, 2006].

[8]To avoid any confusion with meta-level symbols, $\doteq$ is the equality predicate in the logic.

Although the calculus is complete without them, parallel updates allow ad hoc simplifications where a modality is still present. For instance, the above two update can be simplified into the one (parallel) update $\{v := 5\}$. In general, the parallel composition operator $||$ allows such formulae as $\{x := y || y := x\} x \doteq y+1$ which simplifies to $y \doteq x+1$.

3.3 Explicit Heaps

First order dynamic logic with uninterpreted functions is already very expressive; it is able to express meta properties about programs such as (sequential) noninterference. However, it is convenient to add dedicated *theories* to lift the burden of axiomatizing commonly used functions over and over again. Even though theories usually encode higher order properties, they are axiomatizable in first order logic. In the following, we introduce the datatypes Field, LocSet, Heap, and Map; where the latter two are both based on the theory of arrays [McCarthy, 1962]. We omit formal syntax and semantics here as they should be intuitively clear.

While Hoare logic and classical dynamic logic [Harel, 1979] use function symbols for each memory locations, the concept of having just one mathematical object to represent the whole memory of a computer system has been proven to be more convenient regarding information flow properties: one does not need to enumerate all the locations[9] which are unchanged. In this paper, we incorporate the explicit heap data type of Weiß [2011], which is already implemented in the KeY verification system from version 2.0 onwards. The essential functions of this theory are (i) $\mathit{store}(h, \ell, v)$ of type Heap, representing a state change, where h is term of type Heap, ℓ is a location, and v a term of any value type (e.g., integer), (ii) $\mathit{select}(h, \ell)$ of type any, with the above definitions, representing value retrieval from a location, and (iii) $\mathit{anon}(h, L)$ of type Heap, represents a heap which is havocked on all location in the location set L, but agrees on h otherwise. The expression ℓ is of type Field. There are not functions of type Field, but constants for every global variable declared in the system. Location sets (type LocSet) are essentially (finite) sets of fields, using the usual set theoretic operators.

3.4 Base Calculus for DTL

The calculus for DTL is a sequent calculus; sequents are of the shape $\Gamma \Longrightarrow \Delta$ where Γ and Δ are multisets of formulae with the intuitive meaning that the formula $\bigwedge_{\gamma \in \Gamma} \gamma \rightarrow \bigvee_{\delta \in \Delta} \delta$ is true. The left hand side of $\Longrightarrow$ is also called *antecedent*, the right hand side is called *succedent*. Rules of this calculus have zero or more sequents as premisses and one sequent as conclusion; a zero premiss rule is an axiom rule. Rules typically focus on one formula on the left hand or right hand side of the conclusion, for instance the following rules for conjunction:

$$\frac{\Gamma, \phi, \psi \Longrightarrow \Delta}{\Gamma, \phi \wedge \psi \Longrightarrow \Delta}\ \mathsf{R3} \qquad \frac{\Gamma \Longrightarrow \phi, \Delta \qquad \Gamma \Longrightarrow \psi, \Delta}{\Gamma \Longrightarrow \phi \wedge \psi, \Delta}\ \mathsf{R4}$$

The calculus consists of four groups of rules:[10] (i) propositional and quan-

[9] We identify 'location' and 'field' here since we do not have the notion of objects. In [Weiß, 2011], a location is a pair of a receiver object and a field (which is just an identifier).

[10] The numbering of rules in this paper is the same as in [Beckert and Bruns, 2013].

tifier rules, (ii) temporal logic rules (see Tab. 1), (iii) dynamic logic (program) rules (see Tab. 2), and (iv) update rules. The main idea of proving temporal properties deductively is based on the interplay between program and temporal operator rules. Program rules allow symbolic execution of a program in a modality, i.e., simplifying the trace represented by the program. Temporal rules, on the other hand, make use of the fact that temporal formulae may be decomposed into a 'present' and 'future' part, e.g., the formula $\Box\phi$ is logically equivalent to $\phi \wedge \bullet\Box\phi$ on any trace. Since traces are defined through program semantics, and they are of infinite length if and only if we go through an infinite loop, we either reach the end of the program or a computational fixpoint.

$$\frac{\Gamma \Longrightarrow \mathcal{U}(\llbracket\pi\rrbracket\circ(\phi\,\mathbf{U}\,\psi) \wedge \llbracket\pi\rrbracket\phi),\ \mathcal{U}\llbracket\pi\rrbracket\psi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\llbracket\pi\rrbracket\phi\,\mathbf{U}\,\psi, \Delta} \quad \text{R19}$$

$$\frac{\Gamma \Longrightarrow \mathcal{U}\llbracket\pi\rrbracket\bullet\Box\phi, \Delta \qquad \Gamma \Longrightarrow \mathcal{U}\llbracket\pi\rrbracket\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\llbracket\pi\rrbracket\Box\phi, \Delta} \quad \text{R20}$$

$$\frac{\Gamma \Longrightarrow \mathcal{U}\llbracket\pi\rrbracket\circ\Diamond\phi,\ \mathcal{U}\llbracket\pi\rrbracket\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\llbracket\pi\rrbracket\Diamond\phi, \Delta} \quad \text{R21}$$

$$\frac{}{\Gamma \Longrightarrow \mathcal{U}\llbracket\rrbracket\bullet\phi, \Delta} \quad \text{R22}$$

Table 1: Rules for handling temporal operators.

The only rules which work on both the program and the temporal logic part of a formula are the rules for global assignments. As a reminder, traces have been defined in Def. 4 such that only assignments to the global state denote a transition. As a result, rule R26 is only applicable where the program modality contains an assignment and the trace formula begins with a 'next' operator (either weak or strong), which is 'consumed' by this rule. The actual state change is preserved in the update in front of the program modality. This rule has been changed compared with the original DTL definition in [Beckert and Bruns, 2013]: it now uses heap structures instead of global variables.

$$\frac{\Gamma \Longrightarrow \mathcal{U}\{x := a\}\llbracket\omega\rrbracket\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\llbracket\texttt{x = a;}\ \omega\rrbracket\phi, \Delta} \quad \text{R24}$$

$$\frac{\Gamma, \mathcal{U}b \Longrightarrow \mathcal{U}\llbracket\pi_1\ \omega\rrbracket\phi, \Delta \qquad \Gamma, \mathcal{U}\neg b \Longrightarrow \mathcal{U}\llbracket\pi_2\ \omega\rrbracket\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\llbracket\texttt{if (b)}\ \pi_1\ \texttt{else}\ \pi_2\ \omega\rrbracket\phi, \Delta} \quad \text{R25}$$

$$\frac{\Gamma \Longrightarrow \mathcal{U}\{\texttt{heap'} := \texttt{heap}\}\{\texttt{heap} := \mathit{store}(\texttt{heap}, \texttt{X}, a)\}\llbracket\omega\rrbracket\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\llbracket\texttt{X = a;}\ \omega\rrbracket\mathring{\bullet}\phi, \Delta} \quad \text{R26}$$

$$\frac{\Gamma, \mathcal{U}b \Longrightarrow \mathcal{U}\llbracket\pi\ \texttt{while (b)}\ \pi\ \omega\rrbracket\phi, \Delta \qquad \Gamma, \mathcal{U}\neg b \Longrightarrow \mathcal{U}\llbracket\omega\rrbracket\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\llbracket\texttt{while (b)}\ \pi\ \omega\rrbracket\phi, \Delta} \quad \text{R27}$$

Table 2: Program rules. The schematic symbol $\mathring{\bullet}$ stands for either $\bullet$ (weak next) or $\circ$ (strong next).

All other program rules can be applied with any trace formula appearing behind the program modality. Rule R27 captures the trace semantics of loops (cf. Def. 4). It can be used to unfold a loop a finite number of times. In order to obtain a complete calculus, however, we need rules to handle possibly infinite

loops.

3.5 A Theory of Partial Maps

In order to store subsequent heaps of an execution trace, we use the coalgebraic data type Map which represents partial functions. The defining observer is the function application $m(u)$. The main constructor of type Map is the 'guarded lambda' expression $\lambda x : T[\phi].e$ where x is a bound logical variable of type T, ϕ is a state formula, and e is an expression of any type. The intuitive meaning is that every element u for which $\phi[x/u]$ holds is mapped to $e[x/u]$—and otherwise to a unique error element $\perp$. A more thorough account on this theory is given in Appendix A.

4 Security Properties

For sequential programs, a well established security property is noninterference [Sabelfeld and Myers, 2003]. Intuitively, noninterference states that values of confidential (or 'high') inputs must not affect the values of public (or 'low') outputs in any way. In a language based approach, 'inputs' and 'outputs' are both realized as global variables. This definition can be extended to any security lattice; for the course of this paper, however, we stick to the two element lattice.

A bit more formally, the property can be stated as: Given two *low equivalent* (w.r.t. a set L of low variables) states s_1 and s_1', and a program π started in s_1 or s_1' terminates in s_2 or s_2', respectively, then again s_2 and s_2' must be low equivalent:[11]

$$\forall s_1, s_1', s_2, s_2'. \quad s_1 \approx_L s_1' \land s_1 \overset{\pi}{\rightsquigarrow} s_2 \land s_1' \overset{\pi}{\rightsquigarrow} s_2' \rightarrow s_2 \approx_L s_2'$$

We use the symbol $\overset{\pi}{\rightsquigarrow}$ for the state transition relation denoted by π. There are several possible instantiations of the low equivalence relation $\approx_L \subseteq \mathcal{S}^2$. A typical choice (and the most conservative) is equality w.r.t. the restriction to the set L: $s \approx_L s' :\Leftrightarrow s|_L = s'|_L$. Others include, for instance, object sensitive low equivalence [Amtoft et al., 2006, Beckert et al., 2013], which is very useful in the context of Java programs.

Non-interference can be lifted to a property on program traces. Similar to the sequential setting, we define non interference on traces.

Definition 5 (Maximal flow)**.** *Program traces are* low equivalent *if they are of same length and low equivalent on every position:*

$$\tau \approx_L \tau' :\Leftrightarrow |\tau| = |\tau'| \land \forall i \in [0, |\tau|).\tau[i] \approx_L \tau'[i]$$

A program π contains a maximal (information) flow *from locations L_1 to L_2 (up to an index point $i \in \mathbb{N} \cup \infty$) in state s when the two traces of π when started in any state s' which is L_1-equivalent to s yields L_2-equivalent traces:*

$$\text{flow}(s, i, L_1, L_2, \pi) := \quad \forall s' \in \mathcal{S}. \quad s \approx_{L_1} s' \rightarrow trc(s, \pi)[0, i) \approx_{L_2} trc(s', \pi)[0, i)$$

[11] This definition is termination insensitive; termination sensitive definitions are also common.

Note that, different from the state based definition of noninterference, this definition is already termination sensitive. The most typical constellation appears when information flows (at most) from L to itself and $i = \infty$. The more general definition, however, enables us to reason about flows both in a more modular way and including both temporal and subject declassification. This definition is similar to the well known notion of *observational determinism* [Zdancewic and Myers, 2003]. The main difference is that observational determinism only requires that traces are equivalent up to prefixing and stuttering. However, we allow arbitrary finite numbers of nonobservable operations (i.e., local assignments and control statements) to be taken in between observable global assignments.

4.1 Stutter Tolerant Noninterference

The above definition is quite strong in that it does not tolerate stuttering. However, only states which have been reached through assigning a global variable appear in the trace; the number of local assignments has no influence on the length of the trace. Stuttering is important since it is more realistic that the attacker can only observe *changes* to locations instead of states of the trace. We generalize the low equivalence relation $\approx$ to a stutter tolerant relation $\approx^\zeta$.[12]

For a given location set L and traces τ and τ', let $\zeta_L^{\tau,\tau'} \subseteq \mathbb{N} \times \mathbb{N}$ (or just ζ) be the largest relation satisfying the following:

(0) $\zeta(i,i') \to i \in [0,|\tau|) \land i' \in [0,|\tau'|)$

(1) $\zeta(i,i') \to \tau[i] \approx_L \tau'[i']$

(2) $\zeta(i,i') \land \zeta(j,i') \land i < j \to \forall k \in [i,j] : \zeta(k,i')$

(3) $\zeta(i,i') \land \zeta(i,j') \land i' < j' \to \forall k' \in [i',j'] : \zeta(i,k')$

In general, ζ is a many to many relation. Conditions 2f. require ζ to be monotonic (i.e., order preserving) on both traces. Figure 1 shows an example of two traces which are not strictly low equivalent (as in Def. 5), but low equivalent up to stuttering. We can now generalize a stutter tolerant low-equivalence relation on traces:

$$\begin{aligned} \tau \approx_L^\zeta \tau' \quad :\Leftrightarrow \quad & (|\tau| = \infty \leftrightarrow |\tau'| = \infty) \land \forall i \in [0,|\tau|)\ \exists i'.\ \zeta(i,i') \\ & \land \forall i' \in [i,|\tau'|)\ \exists i.\ \zeta(i,i') \end{aligned}$$

It is easy to see that $\approx_L^\zeta$ is again an equivalence relation. Formalizing and constructing proof obligations for stutter tolerant noninterference will be part of future work.

5 Reasoning about Information Flows in DTL

In Sect. 5.1, we add explicit temporal operators for (absence) of information flow, **H** ('hide') and **L** ('leak'). We extend the base calculus presented in Sect. 3.4

[12] Another possible weakening of the above definition would be to allow one trace to be equivalent to a *prefix* of the other one. We do not consider this here as we assume the attacker to always observe changes to the global state.

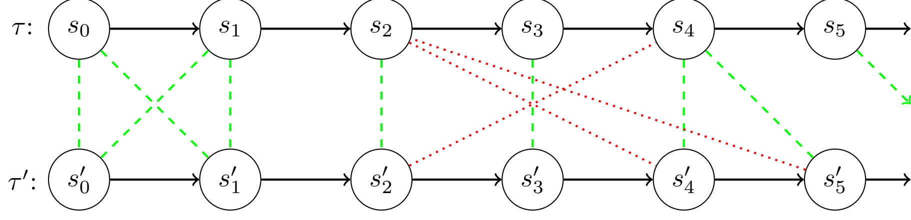

Figure 1: Example of low equivalent traces up to stuttering. Both dashed (green) and dotted (red) lines indicate that the states are low equivalent, but only those marked with dashed lines are in the ζ relation.

ϕ	::=	$\neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \forall x : T.\phi \mid \exists x : T.\phi \mid \mathcal{U}\phi \mid [\![\pi]\!]\psi \mid b$
ψ	::=	$\bullet\psi \mid \circ\psi \mid \Box\psi \mid \Diamond\psi \mid \psi\,\mathbf{U}\,\psi \mid \psi\,\mathbf{W}\,\psi \mid \psi\,\mathbf{R}\,\psi \mid \mathbf{H}\,\psi \mid \mathbf{L}\,\psi \mid \phi$
π	::=	`G = `z | `v = `z | π`;`π | `if (`b`) {`π`} else {`π`}` | `while (`b`) {`π`}`
$\mathcal{U}$	::=	$(\ \{\mathtt{v} := z\} \mid \{\mathtt{heap} := h\}\ )^*$
T	::=	int | Heap | Field | Map
b	::=	`true` | `false` | $e \doteq e$ | $z < z$ | ...
e	::=	$x \mid \mathit{select}(h, \mathtt{G}) \mid \mathrm{ifthenelse}(b, e, e) \mid \bot \mid m(e) \mid z \mid h \mid L \mid m$
z	::=	$z + z \mid z * z \mid -z \mid 0 \mid 1 \mid 2 \mid \ldots$
h	::=	`heap` | `heap'` | $\mathit{store}(h, \mathtt{G}, e) \mid \mathit{anon}(h, L)$
L	::=	$\emptyset \mid \{\mathtt{G}\} \mid L \cup L \mid L \cap L \mid L \setminus L$
m	::=	$\lambda x : T[\phi].e \mid m \oplus m \mid \dot{\varnothing}$

Table 3: Syntax of SecDTL

by adding a few rules for the **H** and **L** operators. In Sect. 5.2, we first show that any property of the form $\mathbf{H}_{L_1,L_2}\,\phi$ can be expressed without this operator. This technique is complete, i.e., it does not report false positives or unknown results, but not very efficient. In Sect. 5.3, we develop additional rules which are applicable to a certain subset of formulae where we can make use of the local reasoning of **H**.

5.1 Extending DTL with Information Flow Operators

We extend DTL by adding the temporal operator **H** ('hide') and its dual **L** ('leak'). These are motivated by operators of the same name in the SecLTL logic [Dimitrova et al., 2012]. The intuition behind the formula $\mathbf{H}_{H,L}\,\phi$ is that no information (valued in the current state) flows from the locations in H to the locations in L (i.e., H is hidden from L) as long as ϕ does not hold (i.e., until ϕ becomes true, but this does not necessarily happen). The formula ϕ can be seen as a release condition which includes the timing of declassification. The dual formula $\mathbf{L}_{H,L}\,\phi$ means that a flow from H to L releases ϕ.

Definition 6 (Syntax of SecDTL). *In addition to the syntax of [Beckert and Bruns, 2013], $\mathbf{H}_{H,L}\,\phi$ is a trace formula where H and L are sets of global variables and ϕ is a formula. We define $\mathbf{L}_{H,L}\,\phi := \neg\,\mathbf{H}_{H,L}\,\neg\phi$.*

Definition 7 (Semantics of **H**).

$$
\begin{aligned}
\mathit{trc}(s,\pi) \vDash \mathbf{H}_{H,L}\,\phi \quad &\text{iff} \quad \mathrm{flow}(s, \infty, \bar{H}, L, \pi) \\
&\text{or} \quad \text{for some } i,\ \mathit{trc}(s,\pi)[i,\infty) \vDash \phi \text{ and } \mathrm{flow}(s, i, \bar{H}, L, \pi)
\end{aligned}
$$

where $\bar{H}$ denotes the complement of H in the set of all locations.

5.2 Formalizing Noninterference

Thanks to the expressiveness of DTL, noninterference can be formalized directly.[13] In our formalization using explicit heaps, this means that heaps must equal on the partition induced by a set of low location L. This technique has been presented before in [Scheben and Schmitt, 2012] for standard JavaDL (input/output determinism). There, it is sufficient to compare the respective heaps in the poststates of each run (if they exist).

In our case, a crucial point is that we need to pairwise compare an unbounded (possibly infinite) number of heaps. For each state on the trace, we add the heap of each execution to the respective sequence in order to compare the resulting sequences. We use the following abbreviation for a temporal formula

$$\begin{aligned}\mathrm{trace}_{Q,\phi} := \quad & Q(0) \doteq \texttt{heap} \wedge \\ & \Box \forall i : \mathrm{int}.(0 \leq i \wedge Q(i) \doteq \texttt{heap} \\ & \quad \rightarrow ((\neg \bullet\phi \rightarrow \bullet Q(i+1) \doteq \texttt{heap}) \\ & \quad\quad \wedge (\bullet\phi \rightarrow \forall j : \mathrm{int}. j > i \rightarrow Q(j) \doteq \bot)))\end{aligned}$$

with a free logical variable Q of type Map and a formula ϕ. Intuitively, it states that Q contains exactly all heaps occurring on the program trace unless formula ϕ holds. Furthermore, we use the notation $h \doteq\!\!\approx_L h'$ with free variables h and h' of type Heap and a set of locations L as shorthand for the formula $\forall \ell.\ \ell \in L \rightarrow \mathrm{select}(h, \ell) \doteq \mathrm{select}(h', \ell)$.

Lemma 1 (Characterization of $\mathbf{H}$)**.** *Let $\mathcal{U}$ be an update, π be a program, τ be a trace, β a variable assignment, and H and L location sets. Let η be the following formula:*

$$\begin{aligned}\forall h' : \mathrm{Heap}\ \forall Q, Q' : \mathrm{Map}.\ (\quad & \texttt{heap} \doteq\!\!\approx_{\bar{H}} h' \wedge \mathcal{U}[\![\pi]\!]\mathrm{trace}_{Q,\phi} \\ & \wedge \mathcal{U}\{\texttt{heap} := h'\}[\![\pi]\!]\mathrm{trace}_{Q',\phi} \\ & \rightarrow \forall i : \mathrm{int}.(i \geq 0 \rightarrow Q(i) \doteq\!\!\approx_L Q'(i)))\end{aligned}$$

If $\tau, \beta \vDash \eta$ then also $\tau, \beta \vDash \mathcal{U}[\![\pi]\!]\, \mathbf{H}_{H,L}\, \phi$

This lemma ensures that we can give a sound rule to translate $\mathbf{H}$. See rule R28 below for the succedent (for the antecendent analogously). The resulting formula also is simpler in some sense ($\mathbf{H}$ does occur at most in ϕ, which occurs twice but under different signs and under a 'next'). As a result, this formalization of noninterference yields a sound and *relatively complete*[14] calculus for SecDTL.

$$\frac{\Gamma \Longrightarrow \eta, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\, \mathbf{H}_{H,L}\, \phi, \Delta}\ \mathsf{R28}$$

[13] An established, alternative technique would be *self composition* [Barthe et al., 2011], which is based on program transformations.

[14] I.e., complete up to arithmetic.

5.3 Local Reasoning about Flows

The downside of the direct formalization of noninterference is that produces quite complex formulae, which lead to a high number of branches in a proof tree. This is particularly an annoyance where (absence of) information flow is more or less 'obvious.' Consider for instance the following program:

```
{ if (L > 0) {x = L;} else {y = L;} }
```

Nothing interesting is happening here. One would like to reason about this program *locally*, i.e., being able to deduce information flow security of the whole program from the premiss that both branches are secure and the branch condition does not have an effect on security. In this way, this reasoning is similar to flow sensitive type systems. A similar calculus has been presented by Ruch [2013] as an improvement over [Scheben and Schmitt, 2012]. The price for this is, of course, losing completeness. For instance, the secure Program 4 from Listing 1 could not be dealt with since the branch condition does actually depend on a high location, and there is a direct assignment from high to low in one branch.

Calculus rules for **H** and **L** are shown in Tab. 4. Rule R35 captures the property that, for any program, the release condition ϕ is sufficient for $\mathbf{H}\,\phi$, i.e., declassification occurs. In the case for the empty program (rule R36), all that is left to show is that high and low locations are disjoint. For a conditional statement (rule R37), we have to show that the condition e is not affected by the choice of values of the locations in H. We use the $e \ltimes_{\mathcal{U}} H$ as shorthand for the formula

$$\mathcal{U}e \doteq \{\mathtt{heap} := anon(\mathtt{heap}, H)\}\mathcal{U}e$$

—to be read 'e is independent of H under update $\mathcal{U}$.' For the special case that e is a global variable X, this is equivalent to $\mathcal{U}\mathtt{X} \notin H$, which we write instead. The rules for global (R38) and local assignments (R39) are similar and straight forward. The only difference is that in R38, we have to prove that the assignment does not constitute an illegal flow already. There is no rule for loops as local reasoning about loops is quite complex already and there would be no obvious benefit from it.

The rules for **L** are dual. Proofs of soundness will be part of future work.

6 Rely/Guarantee Reasoning

The 'classical' approach to reasoning about concurrent programs by Owicki and Gries [1976] is based on the construction of interleaved programs where every possible interleaved execution trace is considered. This technique has some limitations: firstly, the number of concurrent threads is fixed; secondly, it is not compositional in the sense that parts of the concurrent program can be verified in isolation; and finally, through the vast possiblities of interleaved executions, it suffers from a high complexity (which however can be reduced in parts by making stronger assumptions about the scheduling process).

A step ahead is the *rely/guarantee* approach [Jones, 1983, Xu et al., 1997, Henzinger et al., 1998], which tries to avoid these problems. The main idea is similiar to design by contract, though not on the level of a public interface but of atomic program steps. This makes it possible to modularily reason about

$$\frac{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\pi]\!]\, \mathbf{H}_{H,L}\, \phi, \Delta} \quad \text{R35}$$

$$\frac{\Gamma \Longrightarrow \mathcal{U} H \cap L \doteq \emptyset, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![]\!]\, \mathbf{H}_{H,L}\, \phi, \Delta} \quad \text{R36}$$

$$\frac{\Gamma \Longrightarrow e \ltimes_{\mathcal{U}} H, \Delta \qquad \Gamma, \mathcal{U}e \Longrightarrow \mathcal{U}[\![\pi_1\ \omega]\!]\, \mathbf{H}_{H,L}\, \phi, \Delta \qquad \Gamma, \neg\mathcal{U}e \Longrightarrow \mathcal{U}[\![\pi_2\ \omega]\!]\, \mathbf{H}_{H,L}\, \phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\ \texttt{if (e) \{}\pi_1\texttt{\} else \{}\pi_2\texttt{\}}\ \omega]\!]\, \mathbf{H}_{H,L}\, \phi, \Delta} \quad \text{R37}$$

$$\frac{\Gamma \Longrightarrow \mathcal{U} X \notin L, e \ltimes_{\mathcal{U}} H, \Delta \qquad \Gamma \Longrightarrow \mathcal{U}\{X := e\}[\![\omega]\!]\, \mathbf{H}_{H,L}\, \phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\texttt{X = e;}\ \omega]\!]\, \mathbf{H}_{H,L}\, \phi, \Delta} \quad \text{R38}$$

$$\frac{\Gamma \Longrightarrow \mathcal{U}\{v := e\}[\![\omega]\!]\, \mathbf{H}_{H,L}\, \phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\texttt{v = e;}\ \omega]\!]\, \mathbf{H}_{H,L}\, \phi, \Delta} \quad \text{R39}$$

$$\frac{\Gamma \Longrightarrow \mathcal{U}([\![]\!]\phi \wedge H \cap L \not\doteq \emptyset), \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![]\!]\, \mathbf{L}_{H,L}\, \phi} \quad \text{R40}$$

$$\frac{\Gamma, e \ltimes_{\mathcal{U}} H, \mathcal{U}e \Longrightarrow \mathcal{U}[\![\pi_1\ \omega]\!]\, \mathbf{L}_{H,L}\, \phi, \Delta \qquad \Gamma, e \ltimes_{\mathcal{U}} H, \neg\mathcal{U}e \Longrightarrow \mathcal{U}[\![\pi_2\ \omega]\!]\, \mathbf{L}_{H,L}\, \phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\texttt{if (e) \{}\pi_1\texttt{\} else \{}\pi_2\texttt{\}}\ \omega]\!]\, \mathbf{L}_{H,L}\, \phi, \Delta} \quad \text{R41}$$

$$\frac{\Gamma, \mathcal{U}\texttt{X}_0 \notin L, \mathcal{U}\texttt{X}_1 \notin H \Longrightarrow \mathcal{U}[\![\texttt{X}_0\texttt{= X}_1\texttt{;}\ \omega]\!](\phi \wedge \circ\, \mathbf{L}_{H',L}\, \phi), \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\texttt{X}_0\texttt{= X}_1\texttt{;}\ \omega]\!]\, \mathbf{L}_{H,L}\, \phi, \Delta} \quad \text{R42}$$

Table 4: Rules for local reasoning about **H** and **L** formulae.

one single thread in isolation, while there may be an unbounded number of others in a partially specified environment. It is, in particular, not of any interest which program other threads execute or in which (thread local) state they are. Sequential programs (single threads) are evaluated over traces of states. We instrument noninterleaved programs with the special statement `env;` representing an environment action.

Let for a given sequential program π the trace be given as $\langle s_0, \ldots, s_k \rangle$ (for finite traces) or, more generally, for the initial state s_0 the remaing trace be given through the function $\rho : \mathcal{S} \to \mathcal{S}$ which maps any state s_i to its successor s_{i+1}. We assume one definite trace here since a sequential program is deterministic. We now take the environment into consideration; we assume that it is also deterministic (i.e., it contains other deterministic sequential programs which are executed according to a deterministic scheduler) and yields another state transition function $\sigma : \mathcal{S}^2 \to \mathcal{S}$. This function additionally depends on an environment state which is disjoint with the state of π. The resulting trace of the concurrent program π in the given environment then is $\langle s_0, \rho(s_0), \sigma(\rho(s_0), s'), \rho(\sigma(\rho(s_0), s')), \ldots \rangle$. The central idea of *rely/guarantee* is to describe these functions ρ and σ in specifications using formulae *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 σ, i.e., it defines on which properties the execution of π may rely upon. The formula *guar* describes ρ, i.e., it defines which properties the execution of π has to guarantee.

Obviously, there always are strongest formulae satisfying these conditions (if the environment is perfectly known). In practice, this will not be necessary. It is sufficient that *rely* is strong enough to imply the postcondition of π in a final state and that *guar* is strong enough—in conjunction with the *guar* specification of other threads—to imply the *rely* specification of a third party thread.

We combine the well known two state invariant specification of threads with framing, to specify what locations a thread writes to at most. Frame specifications alone can be very expressive, in the dynamic frame 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.

6.1 Definitions and Notations

We have already defined sequential programs in Sect. 2. For now—since we do not have thread identities, objects, or methods—we define a *thread* as containing a sequential program.[15] For a thread t we denote its program by π_t. There may be more than one thread using the same program. A *thread specification* is a pair $(guar_t, M_t)$ where *guar* is a two-state formula and M_t is a term of type LocSet. The intuitive understanding is that for any starting state s_0 for any pair of consecutive states (s_i, s_{i+1}) in the trace $trc(\pi_t, s_0)$, it holds that $s_{i+1} \vDash guar_t$ and that $\mathtt{heap}^{s_i}$ and $\mathtt{heap}^{s_{i+1}}$ are equal up to M_t. The formula *rely* is still a single-state formula in the sense that it must not include temporal operators,

[15] In future work concerning actual Java, we will speak of a thread's `run()` method, more precisely.

but it is expected to refer to the builtin heap variables `heap`and `heap'`, for which we justify it as 'two-state'. The expression M_t can be nontrivial, e.g., depending on the state or including if-then-else operators.

Definition 8. *Two states s and s'* coincide *up to a location set M, written as $s \approx_M s'$, if for all global variables F either $select(\mathtt{heap}^s, F) = select(\mathtt{heap}^{s'}, F)$ or $F \in M$.*

It is obvious to see that $\approx_M$ is an equivalence relation.

For a given thread t we denote the set of concurrently running threads as $\mathcal{T}$, also called *thread configuration*. This will now have an influence on the execution trace of a program. We extend Def. 3 with the environment action `env`.

Definition 9 (Trace of an interleaved program)**.** *Let everything be as in Def. 3, but the trace of a program π is now denoted $trc(s, \mathcal{T}, \pi)$ where $\mathcal{T}$ is a set of threads.*

$$trc(s, \mathcal{T}, \mathtt{env;}\ \omega) = trc(s', \mathcal{T}, \omega)$$

where s' is some state such that $s \approx_{\bigcup_{t\in\mathcal{T}} M_t} s'$ (i.e., s and s' coincide on all heap locations except for $\bigcup_{t\in\mathcal{T}} M_t$) and $s'\{\mathtt{heap'} \mapsto \mathtt{heap}^s\} \vDash guar_t$, for each $t \in \mathcal{T}$. The other defintions are updated accordingly to include $\mathcal{T}$.

We will write $trc(s, \pi)$ whenever the thread configuration $\mathcal{T}$ is not relevant.

Later, we expect that programs under investigation are already instrumented with environment actions before every heap read or write action and the termination action.

Definition 10 (Instrumented program)**.** *Let $\pi = \langle stm_0, \ldots, stm_n \rangle$ 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 or stm_i is a global assignment, the statement `env;` 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 `env;` is inserted after stm_n.*

Please note that this instrumentation is independent of a thread configuration.

6.2 Guarantees

In order to establish that a thread t satisfies its specification, need to prove that it only write to locations specified in M_t and that it fulfills the two state invariant $guar_t$. This relation needs to be proven for any two succeeding states in the trace. The property about M_t is also known as *strict modifies*, in contrast to *weak modifies* properties as imposed in standard JavaDL [Beckert et al., 2007, Weiß, 2011]. A proof obligation can be formulated using the trace modality as

$$\{h^{\mathrm{pre}} := \mathtt{heap}\}[\![\underline{\pi_t}]\!]\Box(frame \wedge \bullet guar_t) \tag{1}$$

where *frame* stands for the formula

$$\forall F : \mathit{Field}.\ (F \in M_t \vee \mathit{select}(\texttt{heap}, F) \doteq \mathit{select}(\texttt{heap'}, F)) \qquad .$$

The formula *frame* is similar the one used to formalize weak modifies properties in [Weiß, 2011, Sect. 6.4.1]. The difference are that not only the very first and the final state are in relation, but every pair of consecutive states.[16] The second part $\bullet \mathit{guar}_t$ entails the two state invariant property. Note that this proof obligation could be written as two separate ones, since the formula $[\![\pi]\!]\Box(\phi_1 \wedge \phi_2)$ is equivalent to $[\![\pi]\!]\Box\phi_1 \wedge [\![\pi]\!]\Box\phi_2$.

Lemma 2. *Let s be a state; let t be a thread with thread configuration $\mathcal{T}$ and thread specification (guar_t, M_t). Let $\tau = \mathit{trc}(s\{h^{\mathrm{pre}} \mapsto \texttt{heap}\}, \mathcal{T}, \pi_t)$. If $\tau \vDash \Box(\mathit{frame} \wedge \bullet \mathit{guar}_t)$ (as in Eq. 1), then all consecutive states $(s_{i-1}, s_i) \in \tau$ pairwise coincide up to M_t.*

Proof: The two-state formula *frame* formalizes $s_0 \approx_{M_t} s_i$ for all $i \in [0, |\tau|)$. By symmetry and transitivity, it follows $s_{i-1} \approx_{M_t} s_i$.

6.3 Rely Component: New Symbolic Execution Rules

By giving a calculus rule to the synthetic environment action statement `env`, 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 R43 below can be applied on a program modality where the environment action `env` is the active statement.

$$\frac{\Gamma, \mathcal{U}\mathit{rely} \Longrightarrow \mathcal{U}\mathcal{V}[\![\omega]\!]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\![\texttt{env;}\ \ \omega]\!]\phi, \Delta}\ \mathsf{R43}$$

where

$$\mathit{rely} := \bigwedge_{t \in \mathcal{T}} \{\texttt{heap'} := \texttt{heap}\}\{\texttt{heap} := \mathit{anon}(\texttt{heap}, M_t)\}\mathit{guar}_t$$

and $\mathcal{V} := \{\texttt{heap} := \mathit{anon}(\texttt{heap}, \bigcup_{t \in \mathcal{T}} M_t)\}$. The rely formula ψ on the left hand side of the premiss is the conjunction of implications over all threads in the runtime environment $\mathcal{T}$. The heap variable is anonymized on M_t. Note that this rule can be applied on any temporal formula ϕ since it does not include a step.

Lemma 3. *Rule R43 is sound.*

In future work, we could do without instrumentation, but then we would have to change all symbolic execution rules concerned with read or write actions. The concurring threads must named somewhere, e.g., in the method frame. No additional rules are required; the present rules must be amended with anonymization like in Rule R43 above.

[16] 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.

6.4 Concurrent Thread Specification in JML

For specifications, we extend the Java Modeling Language (JML) [Leavens et al., 2013], which is already used in the KeY system. JML is a popular and powerful specification language for Java programs based on the design by contract paradigm; the main concepts are class invariants are method contracts. JML integrates seamlessly into Java as it is embedded inside comments in Java source code and JML expressions extend Java expressions in a natural way. By now, JML has become the de facto standard in formal specification of Java source code.

in JML essentially consist of preconditions, indicated by the keyword `requires`, postconditions (`ensures`), and frame conditions (`assignable`). More than one of each of these specification constructs may be declared, in which case they are conjoined.

On the 'rely' side, we introduce a new clause to JML method contracts, the `competing` clause. It specifies a list of instances of `java.lang.Runnable` (which is an interface type declaring only method `run()`), together with their runtime type, which may run concurrently to the specified method. This identifies the threads of a thread configuration and points to their respective `run()` methods. Method preconditions may impose additional constraints on the concurrently running `Runnable` instances. For example:

```
/*@ normal_behavior
  @ competing (RunnableImpl) r;
  @ competing (YetAnotherRunnable) t;
  @ requires r.x >= 0 && t.y;
  @ ensures z == \old(z)+1;
  @*/
public void inc() { this.z++; }
```

On the 'guarantee' side, we use the well known `assignable` clause and introduce a new `guarantees` clause. The `assignable` clause is standard JML; it is followed by a list of heap locations (global variables) and intuitively means that only those may be assigned *throughout* the execution, or equivalently, the value of all other locations must not be changed in any state reached throughout the execution. The described property is exactly 'strict modifies' as mentioned in Sect. 6.2. The `guarantees` clause is new. It is being followed by a boolean expression (i.e., formula), which may use the special operator `old`, which allows to refer to a previous state. In postconditions of (DbC) method contracts this is the prestate, here it denotes the previous intermediate state on the trace. Thus, it describes a two state property on the trace.

```
class RunnableImpl implements Runnable {
    private int x;

    //@ guarantees x>= 0 && x >= \old(x);
    //@ assignable x;
    public void run() { ... }
}
```

To make specifications easier to read/write, we assume that we have named implementations of `Runnable` (as above) instead of the more common pattern of anonymous classes.

In further work, the `competing` clause could be replaced by a predicate, which could appear also in postconditions, allowing for dynamic thread creation/destruction. For example:

```
/*@ normal_behavior
  @ requires \competing(\nothing);
  @ ensures \competing((RunnableImpl) r);
  @*/
public void foobar () {
    Runnable r = new RunnableImpl();
    Thread t = new Thread(r);
    t.start();
}
```

6.5 Example

In this section, we outline how a proof of functional correctness for multithreaded Java might look like. Although there is no formal translation of the above introduced JML constructs to dynamic logic, we assume that this is intuitively clear.

Take the above specification for method `inc()`. The goal is to show that the field `z` increases by one while instances of `RunnableImpl` may run concurrently (we ignore the other class `YetAnotherRunnable` here). The proof is shown in Fig. 2. As usual in tableau-like calculi, it is to be read bottom up.

To keep the example simple, yet expressive, we make some assumptions about the final calculus and some simplifications to the presentation:

- We use the 'box' modality $[\cdot]$ from classical dynamic logic here. The rules presented for the $[\![\cdot]\!]$ modality are applied accordingly.
- We deliberately use the Java postincrement statement here, which is decomposed into subsequent read/write through program transformation in the symbolic execution (rule postinc)—such rules already exist in the JavaDL calculus.
- There is no explicit instrumentation in the program; rule R43 is miraculously applied in the right spot.
- We denote the set of concurrently running threads with an superscript index in the program modality.
- We also use the notion of parallel updates to allow intermediate update simplifications (rule sUpd).
- The functional part of r's guarantee is not important to the proof; we indicate this through an ellipsis on the left hand side.

7 Related Work

Temporal reasoning as well as verification of concurrent programs has traditionally been the domain of model checkers such as Java PathFinder [Havelund and

$$\ldots \Longrightarrow \mathit{true}$$

$$\ldots \Longrightarrow \mathit{select}(\mathtt{heap}, \mathtt{z}) + 1 \doteq \mathit{select}(\mathtt{heap}, \mathtt{z}) + 1$$

$$\text{selectOfAnon}$$

$$\ldots \Longrightarrow \mathit{select}(\mathit{anon}(\mathtt{heap}, \mathtt{x}), \mathtt{z}) + 1 \doteq \mathit{select}(\mathtt{heap}, \mathtt{z}) + 1$$

$$\text{selectOfStore}$$

$$\ldots \Longrightarrow \mathit{select}(\mathit{store}(\mathit{anon}(\mathtt{heap}, \mathtt{x}), \mathtt{z}, \mathit{select}(\mathit{anon}(\mathtt{heap}, \mathtt{x}), \mathtt{z}) + 1), \mathtt{z}) \doteq \mathit{select}(\mathtt{heap}, \mathtt{z}) + 1$$

$$\ldots \Longrightarrow \{h^{\mathrm{pre}} := \mathtt{heap} || \mathtt{heap} := \mathit{store}(\mathit{anon}(\mathtt{heap}, \mathtt{x}), \mathtt{z}, \mathit{select}(\mathit{anon}(\mathtt{heap}, \mathtt{x}), \mathtt{z}) + 1)\} \mathit{select}(\mathtt{heap}, \mathtt{z}) \doteq \mathit{select}(h^{\mathrm{pre}}, \mathtt{z}) + 1$$

$$\ldots \Longrightarrow \{h^{\mathrm{pre}} := \mathtt{heap} || \mathtt{heap} := \mathit{store}(\mathit{anon}(\mathtt{heap}, \mathtt{x}), \mathtt{z}, \mathit{select}(\mathit{anon}(\mathtt{heap}, \mathtt{x}), \mathtt{z}) + 1)\} [^r] \mathit{select}(\mathtt{heap}, \mathtt{z}) \doteq \mathit{select}(h^{\mathrm{pre}}, \mathtt{z}) + 1$$

$$\text{sUpd}$$

$$\ldots \Longrightarrow \{h^{\mathrm{pre}} := \mathtt{heap} || \mathtt{heap} := \mathit{anon}(\mathtt{heap}, \mathtt{x}) || \mathtt{t} := \mathit{select}(\mathit{anon}(\mathtt{heap}, \mathtt{x}), \mathtt{z})\} \{\mathtt{heap} := \mathit{store}(\mathtt{heap}, \mathtt{z}, \mathtt{t} + 1)\} [^r] \mathit{select}(\mathtt{heap}, \mathtt{z}) \doteq \mathit{select}(h^{\mathrm{pre}}, \mathtt{z}) + 1$$

$$\text{R26}$$

$$\ldots \Longrightarrow \{h^{\mathrm{pre}} := \mathtt{heap} || \mathtt{heap} := \mathit{anon}(\mathtt{heap}, \mathtt{x}) || \mathtt{t} := \mathit{select}(\mathit{anon}(\mathtt{heap}, \mathtt{x}), \mathtt{z})\} [^r \mathtt{z = t+1;}] \mathit{select}(\mathtt{heap}, \mathtt{z}) \doteq \mathit{select}(h^{\mathrm{pre}}, \mathtt{z}) + 1$$

$$\text{sUpd}$$

$$\ldots \Longrightarrow \{h^{\mathrm{pre}} := \mathtt{heap} || \mathtt{heap} := \mathit{anon}(\mathtt{heap}, \mathtt{x}) || \mathtt{t} := \mathit{select}(\mathit{anon}(\mathtt{heap}, \mathtt{x}), \mathtt{z})\} \{\mathtt{heap} := \mathit{anon}(\mathtt{heap}, \mathtt{x})\} [^r \mathtt{z = t+1;}] \mathit{select}(\mathtt{heap}, \mathtt{z}) \doteq \mathit{select}(h^{\mathrm{pre}}, \mathtt{z}) + 1$$

$$\text{R43}$$

$$\ldots \Longrightarrow \{h^{\mathrm{pre}} := \mathtt{heap} || \mathtt{heap} := \mathit{anon}(\mathtt{heap}, \mathtt{x}) || \mathtt{t} := \mathit{select}(\mathit{anon}(\mathtt{heap}, \mathtt{x}), \mathtt{z})\} [^r \mathtt{z = t+1;}] \mathit{select}(\mathtt{heap}, \mathtt{z}) \doteq \mathit{select}(h^{\mathrm{pre}}, \mathtt{z}) + 1$$

$$\text{sUpd}$$

$$\ldots \Longrightarrow \{h^{\mathrm{pre}} := \mathtt{heap}\} \{\mathtt{heap} := \mathit{anon}(\mathtt{heap}, \mathtt{x})\} \{\mathtt{t} := \mathit{select}(\mathtt{heap}, z)\} [^r \mathtt{z = t+1;}] \mathit{select}(\mathtt{heap}, \mathtt{z}) \doteq \mathit{select}(h^{\mathrm{pre}}, \mathtt{z}) + 1$$

$$\text{R24}$$

$$\mathtt{x} \geq 0 \land \{h^{\mathrm{pre}} := \mathtt{heap}\} \{\mathtt{heap'} := \mathtt{heap}\} \{\mathtt{heap} := \mathit{anon}(\mathtt{heap}, \{\mathtt{x}\})\} (\mathtt{x} \geq 0 \land \mathtt{x} \geq \mathit{select}(h^{\mathrm{pre}}, \mathtt{x}))$$

$$\Longrightarrow \{h^{\mathrm{pre}} := \mathtt{heap}\} \{\mathtt{heap} := \mathit{anon}(\mathtt{heap}, \mathtt{x})\} [^r \mathtt{int\ t = z;\ z = t+1;}] \mathit{select}(\mathtt{heap}, \mathtt{z}) \doteq \mathit{select}(h^{\mathrm{pre}}, \mathtt{z}) + 1$$

$$\text{R43}$$

$$\mathtt{x} \geq 0 \Longrightarrow \{h^{\mathrm{pre}} := \mathtt{heap}\} [^r \mathtt{int\ t = z;\ z = t+1;}] \mathit{select}(\mathtt{heap}, \mathtt{z}) \doteq \mathit{select}(h^{\mathrm{pre}}, \mathtt{z}) + 1$$

$$\text{postinc}$$

$$\mathtt{x} \geq 0 \Longrightarrow \{h^{\mathrm{pre}} := \mathtt{heap}\} [^r \mathtt{z++;}] \mathit{select}(\mathtt{heap}, \mathtt{z}) \doteq \mathit{select}(h^{\mathrm{pre}}, \mathtt{z}) + 1$$

Figure 2: Proof tree for the example

Pressburger, 2000] or Bogor [Robby et al., 2006]. Other common techniques to make the behavior of concurrent programs more expectable are permission systems and ownership annotations, which are checked at runtime. Notable examples are built into the Spec# [Barnett et al., 2005] and Dafny [Leino and Müller, 2009] languages.

7.1 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 ressemble 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*—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. For this reason, the formula $\langle \pi_1 \cap \pi_2 \rangle \phi$ with π_1 and π_2 executed in parallel is just equivalent to $\langle \pi_1 \rangle \phi \wedge \langle \pi_2 \rangle \phi$.

A closely related work is [Schellhorn et al., 2011] which extends 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, which is implemented in the KIV theorem prover.

Another approach using dynamic logic is taken in [Beckert and Klebanov, 2013], which uses a realistic 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 program statements are deleted and the one program pointer is implicitly at the beginning of the remainder. They use a Java-like language, but impose the strong assumption that all loops are atomic. It also includes atomic blocks which are symbolicly executed in another kind of DL modality.

7.2 Temporal Behavior of Java Programs

Programs of concrete programing languages like Java are usually reasoned about in a state based manner. There are a few runtime checking approaches which check for trace properties using LTL-like specification [Bartetzko et al., 2001, Stolz and Bodden, 2006]. Hussain and Leavens [2010] also check assertions at runtime, but in addition use temporalJML as an extension to the JML specification language, which allows to write high level temporal properties, but is not as expressive as LTL.

7.3 Information Flow Analysis Based on Theorem Proving

Amtoft and Banerjee [2004] were among the first to encode noninterference in a program logic and therefore to lay the foundation for sound and complete reasoning about information flows. Since classical Hoare logic cannot express

noninterference, they employed a dedicated extension of it. Darvas et al. [2005] continue this idea and use dynamic logic instead, which can express such properties readily.

Hähnle et al. [2008] and Popescu et al. [2012] embed traditional type systems into first/higher order logic. Even though reasoning is done using a theorem prover (KeY or Isabelle, respectively), it still suffers from the inherent incompleteness of type systems and only checks for sufficient conditions for secure information flow. For instance, it is not able to verify program 4 in Listing 7.

A A Theory of Partial Mappings

In Sect. 5.2, we need to pairwise compare the states (or, more precisely: the value of the builtin `heap` variable) which occur in the sequences produced from two runs of the same program. However, we cannot express this with the sequence data type because it is restricted to finite sequences. We would need a formula like the following (with a free variable Q of type Seq):

$$\Box\forall i.(0 \leq i \land Q[i] \doteq \texttt{heap} \rightarrow (\bullet Q[i+1] \doteq \texttt{heap} \land (\bullet \mathit{false} \rightarrow \mathrm{len}(Q) \doteq i+1)))$$

Through instantiation of i with $len(Q)$, it can be shown to be unsatisfiable.

This is the reason for which we introduce a more general theory, the theory of partial mappings (partial functions). Maps are a kind of *coalgebraic data type*, cf. [Jacobs and Rutten, 1997]. While *algebraic* data structures are defined through their *constructurs*, coalgebraic data types are defined through their *observers* (also called *destructors*). We can also say that algebraic data types are constructed, while coalgebraic data types are observed.

Our data type Map is defined through the observer function $mapGet : \mathrm{Map} \times \mathrm{Any} \rightarrow \mathrm{Any}$ and the predicate $inDomain : \mathrm{Map} \times \mathrm{Any}$. The core constructor is $foreach(x : T)(\phi, t)$ where x is a (bound) logical variable of type T,[17] ϕ is a formula, and t is an expression. The intended meaning is that it maps each x for which ϕ is true to t and is undefined otherwise. The axiomatic base rules are shown in Tab. 5. The squiggely arrow $\rightsquigarrow$ means that they are rewrite rules, which may be applied on either side of the sequent and in the scope of any formula/term.

$$\lambda x[\phi].t\,(v) \rightsquigarrow \mathrm{if}(\phi[v/x])\mathrm{then}(t[v/x])\mathrm{else}(\bot) \qquad \text{R44}$$

$$v \in \mathrm{dom}(\lambda x[\phi].t) \rightsquigarrow \phi[v/x] \qquad \text{R45}$$

Table 5: Base axioms for the theory of partial maps

In order to practically work with the Map data type, we define additional constructor functions derived from *foreach*, i.e., conservative extensions.[18] Those are given in Tab. 7. The definitions of the empty map and a singleton map are straight forward. Maps are joined with the *mapOverride* function. It intuitively means that the resulting domain is the union of the subdomains m_0 and m_1, and for each element in the intersection, m_1 overrides m_0. We also define a wrapper function for finite sequences, which delegates to the *len* and *seqGet* functions of the Seq data type. For readability, we will use the more common mathematical notations as displayed in Tab. 6 instead of the notation above.

As the last definition in Tab. 7 shows, finite sequences can be easily embedded into the Map type. Sets can also be embedded by restricting the domain and defining an arbitrary application term t. Containment can just be expressed using *inDomain*.

Note that given these core axioms, that it cannot be proven that maps are wellfounded, i.e., that they do not contain themselves. This can be solved by

[17] As we work with a typed logic, x must be declared with a type. For convenience reasons, we omit it if any type is appropriate.

[18] We do not prove here that these extensions are actually conservative.

KeY notation	math notation
$foreach(x:T)(\phi,t)$	$\lambda x:T[\phi].t$
$mapUndef$	$\bot$
$mapGet(m,v)$	$m(v)$
$mapEmpty$	$\dot{\varnothing}$
$mapSingleton(y,t)$	$\{y \mapsto t\}$
$mapOverride(m_0,m_1)$	$m_0 \oplus m_1$
$inDomain(m,v)$	$v \in \mathrm{dom}(m)$
$mapDepth(m)$	$\downarrow m$

Table 6: Mathematical notions for the Map data type

$$\begin{array}{lll}
\dot{\varnothing}^s & := & (\lambda x[\mathit{false}].42)^s \\
\{y \mapsto t\}^s & := & (\lambda x[x \doteq y].t[x/y])^s \\
(m_0 \oplus m_1)^s & := & \begin{array}{l}(\lambda x[x \in \mathrm{dom}(m_0) \vee x \in \mathrm{dom}(m_1)]. \\ \quad \mathrm{if}(x \in \mathrm{dom}(m_1))\mathrm{then}(m_1(x))\mathrm{else}(m_0(x)))^s\end{array} \\
(\mathit{seq2map}(\mathit{seq}))^s & := & (\lambda i : \mathrm{int}[0 \leq i < \mathit{len}(\mathit{seq})].\mathit{seqGet}(\mathit{seq}, i)))^s
\end{array}$$

Table 7: Conservative extensions

adding a third observer $mapDepth : \mathrm{Any} \to \mathbb{N}$ (mathematical notation $\downarrow$) with the obvious semantics:

$$(\downarrow \lambda x[\phi].t)^s = \begin{cases} 0 & s \vDash \forall x.\neg\phi \\ \max_{x.\phi}\{(\downarrow t)^s\} & \text{otherwise} \end{cases}$$

However, this kind of completeness property is not needed for this paper.

References

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.

Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In Roberto Giacobazzi, editor, *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, volume 3148 of *Lecture Notes in Computer Science*, pages 100–115. Springer, 2004. ISBN 3-540-22791-1. URL http://dx.doi.org/10.1007/978-3-540-27864-1_10.

Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 91–102. ACM, 2006. ISBN 1-59593-027-2.

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.

Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011. URL http://dx.doi.org/10.1017/S0960129511000193.

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, 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, 2007.

Bernhard Beckert, Daniel Bruns, Vladimir Klebanov, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. Information flow in object-oriented

software. In Gopal Gupta, editor, *Logic-Based Program Synthesis and Transformation, LOPSTR 2013*, 2013.

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

Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010. URL http://dx.doi.org/10.3233/JCS-2009-0393.

Ádám Darvas, Reiner Hähnle, and Dave Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *Proc. 2nd International Conference on Security in Pervasive Computing*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer-Verlag, 2005.

Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N. Rabe, and Helmut Seidl. Model checking information flow in reactive systems. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, January 2012.

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2013. ISBN 978-0133260229.

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, Jing Pan, Philipp Rümmer, and Dennis Walter. Integration of a security type system into a program logic. *Theor. Comput. Sci*, 402(2-3): 172–189, 2008. URL http://dx.doi.org/10.1016/j.tcs.2008.04.033.

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.

Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You assume, we guarantee: Methodology and case studies. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*, pages 440–451. Springer, 1998. ISBN 3-540-64608-6. URL http://dx.doi.org/10.1007/BFb0028765.

C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1:271–281, 1972.

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.

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.

Rajeev Joshi and K. Rustan M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1-3):113–138, 2000.

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.

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

Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, and Werner Dietl. *JML Reference Manual*, May 31, 2013. URL http://www.eecs.ucf.edu/~leavens/JML/documentation.shtml. Draft Revision 2344.

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.

Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, New York, 1995. ISBN 0-387-94459-1.

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

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

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

Andrei Popescu, Johannes Hölzl, and Tobias Nipkow. Proving concurrent noninterference. In Chris Hawblitzel and Dale Miller, editors, *CPP*, volume 7679 of *Lecture Notes in Computer Science*, pages 109–125. Springer, 2012. ISBN 978-3-642-35307-9.

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.

Fabian Ruch. Efficient logic-based information flow analysis of object-oriented programs. Bachelor thesis, Karlsruhe Institute of Technology, 2013. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000036850.

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.

Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

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*, 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, *TIME*, pages 99–106. IEEE, 2011. ISBN 978-1-4577-1242-5. URL http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6063703.

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.

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

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

Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop*, page 29. IEEE Computer Society, 2003. ISBN 0-7695-1927-X. URL http://doi.ieeecomputersociety.org/10.1109/CSFW.2003.1212703.