A System for Compositional Verification of Asynchronous Objects

Wolfgang Ahrendt[a,1,*], Maximilian Dylla[b,2]

[a] Department of Computer Science and Engineering, Chalmers University of Technology, Sweden
[b] Saarbrücken Graduate School of Computer Science, Saarland University, Germany

Abstract

We present a semantics, calculus, and system for compositional verification of Creol, an object-oriented modeling language for concurrent distributed applications. The system is an instance of KeY, a framework for object-oriented software verification, which has so far been applied foremost to sequential Java. Building on KeY characteristic concepts, like dynamic logic, sequent calculus, symbolic execution via explicit substitutions, and the taclet rule language, the presented system addresses functional correctness of Creol models featuring local cooperative thread parallelism and global communication via asynchronous method calls. The calculus heavily operates on communication histories specified by the interfaces of Creol units. Two example scenarios demonstrate the usage of the system. This article is an extended version of [5].

Keywords: verification, concurrency, semantics, object-orientation

1. Introduction

The area of object-oriented program verification made significant progress during the last decade. Systems like Boogie [8], ESC/Java2 [32], KeY [12], and Krakatoa [31] provide a high degree of automation, elaborate user interfaces, extensive tool integration, support for various specification languages, and high coverage of a real world target language (Spec# in case of Boogie, Java in case of the other mentioned tools).

However, this development mostly concerns *sequential*, *free-standing* applications. When it comes to verifying functional properties of *concurrent* and *distributed* applications, the situation is different. Even if there is a rich literature on the verification of 'distributed formalisms' (based for instance on process calculi [46, 36, 47]), there are hardly any systems yet matching the aforementioned characteristics. Moreover, many formalisms lack a connection to the dominating paradigm of today's software engineering, *object-orientation*, which is an obstacle for the integration into software development environments and methods.

This work is a contribution towards effective and integrated verification of concurrent, distributed systems. We present a verification system that is built on two foundations: the Creol modeling language for concurrent and distributed object-oriented systems [43], and the KeY approach and system for the verification of object-oriented programs [12]. By combining KeY's proving technology with Creol's novel approach to modular modeling of components, which has been successfully applied to industrial scale problems [22], we achieve a *system for compositional verification of concurrent, distributed object-oriented applications*. While being a prototype system yet, past experience with the technological and conceptual basis justifies the perspective of future versions to enjoy similar features as state-of-the-art sequential verification systems already do. The scaling up of verification technology from the sequential to the concurrent/distributed

*Corresponding author
Email addresses: ahrendt@chalmers.se (Wolfgang Ahrendt), mdylla@mpi-inf.mpg.de (Maximilian Dylla)

[1] This work has partially been supported by the EU-project FP7-ICT-2007-3 *HATS: Highly Adaptable and Trustworthy Software using Formal Methods.*

[2] This work has partially been supported by the Saarbrücken Graduate School of Computer Science which receives funding from the DFG as part of the Excellence Initiative of the German Federal and State Governments, and by the EU COST action IC0701: *Formal Verification of Object-Oriented Software.*

setting would, however, not be possible without *modularity* being the central element in the design of Creol, as discussed in the following.

Creol is an executable object-oriented modeling language. It features concurrency in two ways. First of all, different objects execute truly in parallel, as if each object had its own processor. Objects have references to each other, but cannot access each others internal state. Consequently, there is no remote access to attributes, like '`o.a`' in other languages. The only way for objects to exchange information is through methods. Calls to methods are *asynchronous* [42], in the sense that the calling code is able to continue execution even before the callee replies. Mutual information hiding is further strengthened by object variables being typed by interfaces only, not by classes. The loose coupling of objects, their strong information hiding and true parallelism, is what suggests *distributed* scenarios, with each object being identified with a node. The second type of concurrency is object internal. Each call to a method spawns a separate thread of execution. Within one object, these threads execute *interleaved*, with only one thread running at a time. Here, the key to modularity is the cooperative nature of the scheduling: a thread is only ever interrupted when it actively releases control, at 'release points'.

Altogether, Creol allows *compositional verification*. Within one class, the various methods can be proved correct in isolation, in spite of the shared memory (the attributes), by guaranteeing and assuming a *class invariant* at each release point in the code. At the inter-object level, the vehicle to connect the verification of the various classes is the *'history'* of inter-object communications. *Interface invariants* are expressed in terms of the history only, while class invariants relate the history with the internal state. The fact that each object has only partial knowledge about the global communication history is modeled by *projecting* the global history onto the individual objects [41].

Our system is based on the KeY framework for verifying object-oriented software. The most elaborate instance of KeY is a verification system for sequential Java [12]. Other target languages of KeY are C [52], ASMs [53], and hybrid systems [56]. All these have in common that they use *dynamic logic*, *explicit substitutions*, and a *sequent calculus* realised by the *'taclet'* language. These concepts, to be introduced in the course of the paper, have proved to be a solid foundation of a long lasting and far reaching research project and system for verifying functional correctness of Java [12]. Dynamic logic features full source code transparency, like Hoare logic, but is more expressive than that. Explicit (simultaneous) substitutions, called *updates*, provide a compact representation of the symbolic state, and allow a natural forward style symbolic execution. Apart from verification, updates are also employed for test case generation and symbolic debugging. Sequent calculi are well-suited for the interleaved automated and interactive usage. And finally, *taclets* provide a high-level rule language capturing both the logical and the operational meaning of rules. They are well suited both for the base logic and for the axiomatization of application specific operations and predicates. KeY has been used in a number of case studies, like the verification of the *Java Card API Reference Implementation* [51], the *Mondex* case study (the most substantial benchmark in the Grand Challenge repository) [58], the *Schoor-Waite algorithm* [16], and the electronic purse application *Demoney* [50]. The system is also used for teaching in various courses at Chalmers University and several other universities.

However, the KeY approach has so far almost only been applied to the sequential setting.[3] It is precisely the described modularity of Creol that allowed us to base our verification system on the same framework. The main challenges for adjusting the KeY approach to Creol were the handling of asynchronous method calls, the handling of release points, and, most of all, the extensive usage of the communication history throughout the calculus.

The structure of the paper is as follows. Sect. 2 introduces Creol, and gives examples of its usage. Furthermore, in Sect. 3 denotational semantics for Creol are explained. In Sect. 4, we describe the logic and calculus characteristic for KeY, insofar as they are (largely) independent of the particular target language. Thereafter, Sect. 6 presents a KeY style logic and calculus for Creol specifically. Sect. 7 discusses system oriented aspects of KeY for Creol, including a small account on taclets. Sect. 8 then demonstrates the usage of the systems in examples. In Sect. 9, we discuss related work, and draw conclusions. Finally, in Appendix

[3] See Sect.9 for an exception.

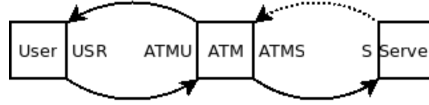

Figure 1: Communication of the automated teller machine

```
interface USR
begin
  with ATMU
    op giveCode(in; out code:Int)
    op withdraw(in; out amount:Int)
    op dispense(in amount:Int; out)
    op returnCard(in; out)
end

interface ATMU
begin
  with USR
    op insert(in cardId:Int; out)
end

interface S
begin
  with ATMS
    op authorize(in cardId:Int, code:Int; out ok:Bool)
    op debit(in cardId:Int, amount:Int; out ok:Bool)
end
```

Figure 2: The interfaces of the automated teller machine

A grammars describing the syntax of Creol, followed by Appendix B and Appendix C serving as a reference for the semantics and the calculus, respectively, conclude the paper.

This article extends the conference paper [5] with a denotational semantics of Creol (Sect.3) and with an assumption-commitment/rely-guarantee style semantics of the logic (Sect.5). Moreover, the calculus presented here is simplified wrt. [5]. On the implementation side, new strategies were realised, resulting in an automation degree of more than 98% in the examined case studies, see Sect.8.

2. Overview of Creol

In this section, we introduce Creol, using an automated teller machine scenario adapted from [39]. The example will also be used to discuss Creol verification in later sections. The scenario we consider has three kinds of actors. There are several teller machines (class ATM), several users (class User), and one server (class Server). In the course of a certain session, a teller machine communicates with one user, and with the server, as depicted in Fig. 1. The picture shows that, while User and Server implement one interface each (USR resp. S), the class ATM implements two interfaces, ATMU and ATMS, dedicated for the communication in either of the directions. The Creol definition of the interfaces is given in Fig. 2. (We omit ATMS, which is empty.)

We can observe that the signature of operations contains (possibly empty) lists for **in**- and **out**-parameters. The operations offered by interfaces appear in the scope of '**with** *cointerface*', with the meaning that those operations can only be called from instances of classes implementing that *cointerface*. For instance, the server cannot call insert on a teller machine, not even if it was in the possession of an ATMU typed reference. Another consequence of cointerfaces is that the implementations of operations have a well-typed reference to the caller, without that reference being passed explicitly as an input parameter.

The class ATM in Fig. 3 is an example for a class definition. Variables are implicitly initialised with *false* or 0 for primitive types, and null for labels and object references. Some variables are declared of type **Label**[...], like **var** li:**Label[Int]**. Later, the execution of the call li!caller.giveCode(), for instance, allocates

```
class ATM implements ATMS, ATMU
begin
  var server : S;
  with USR
    op insert(in card:Int; out) ==
      var li:Label[Int]; var lb:Label[Bool]; var l:Label[];
      var l2:Label[]; var code:Int; var ok:Bool; var am:Int;
      li!caller.giveCode(); li?(code);
      lb!server.authorize(card,code); lb?(ok);
      if ok
      then li!caller.withdraw(); li?(am);
        lb!server.debit(card,am); lb?(ok);
        if ok
        then l!caller.dispense(am); l2!caller.returnCard(); l?(); l2?()
        else l!caller.returnCard(); l?() end
      else l!caller.returnCard(); l?() end; return()
end
```

Figure 3: The class implementing the teller machine

a new label, and assigns it to li. The label is later used in the *reply* statement li?(code), to associate the reply with the respective call. The effect of the reply is that code is assigned the output of the (li-labeled) call to giveCode, *provided that the according reply message has already arrived.* Otherwise, the statement blocks, without the thread releasing control. (This 'busy waiting' can be avoided by the **await** statement, see below.) The effect of li?(x) is similar to treating x as a *future variable* [21, 7] or *promise* [45]. In a label type **Label**[T], the T indicates the type of the output of the called operation.

Note that the calls to dispense and returnCard are executed before any of the replies is asked back. This allows the two called methods to execute *interleaved* on the processor of the called object. (Note that the calls went to the same object.) In general, arbitrary code can be executed in between a call and the corresponding reply. We want to highlight that the implementation of insert extensively uses the caller reference, which is known to be of type USR, for callbacks. This style of coupling communicating objects might clarify the distribution of operations over interfaces in the teller machine scenario (cf. Fig. 2).

We discuss further features of Creol not captured by the above example. New objects are created by $x :=$ **new** $C(e^*)$, where C is a class identifier supplied with a list of class parameters. As indicated earlier, $l?(x^*)$ blocks execution, *without releasing control*, until the according reply message has not arrived. In contrast, the command **await** $l?$ releases control if the reply for l has not yet arrived, such that the scheduler can pass control to another thread of this object. Other release points are **await** b, releasing control if the Boolean expression b is false, and the unconditioned **release**. The example code above did not contain release points, but see the buffer example in Sect.8.1 (Fig. 8).

In Creol, expressions have no effect on the state. We model errors, like division by zero, by non-terminating (and non-releasing) blocking. The same holds for a call on the null reference and a reply on the null label.

3. Denotational Semantics for Creol

Previous work on the semantics of Creol focused *operational* semantics [40, 15]. In this article, we present a *denotational* semantics of Creol. It is an intrinsic feature of denotational semantics that they are *compositional.* This is a very good fit to our goal of compositional verification, because the compositional calculus can relate in a natural way to the semantics. The comprehensive surveys of de Roever et al. on compositional verification of concurrent programs [23, 37] were very inspiring for this work. One basic principle, invented by Zwiers [60], is to construct histories of process interactions for each process independently,

by non-deterministically 'guessing' the relevant observations on other processes. Then, in the composition of processes, we merge those histories which 'agree' on certain observations. That merge is defined as the inverse of a projection. We drive this 'guess-and-merge' principle very far, to cope with dynamic creation of arbitrarily many objects and threads. For instance, we will 'guess' the number of times a certain method is called, to then require that the result can only be merged with histories actually providing the right number of calls (among other things).

We proceed in a bottom-up manner, ranging from single statements via method bodies and objects to the semantics of the complete program. As a first ingredient we use a state denoted by σ. The state is a partial function pointing from the local variables of a thread and the class attributes to their current values. Sometimes we restrict the preimage of the state to the local variables or the class attributes by writing $\sigma|_l$ or $\sigma|_a$, respectively. The other important member of our semantics is the history θ being a sequence of messages. Our semantics is defined by a function on programs

$$\mathcal{M} : PROG \to (\Sigma \to 2^{\Sigma \times H})$$

where $PROG$ is the set of programs, Σ denotes the set of all states and H contains all histories. The function $\mathcal{M}$ associates to a program a function which relates every initial state with a set of pairs containing the possible states and histories the program terminates with. Note that the resulting history only reflects the run this very program, not any 'initial' history. (But see sequential composition below, and the semantics of formulas relative to an initial history, Sect. 5). Non-terminating program runs result in an empty set. As an example we have a look at the **block** statement, which never terminates.

$$\mathcal{M}(\textbf{block})(\sigma) = \{\} \qquad \mathcal{M}(\textbf{skip})(\sigma) = \{(\sigma, \langle\rangle)\}$$

The **skip** statement terminates, but causes no changes, so its set includes the tuple of the input state σ and the empty history $\langle\rangle$. To proceed with more interesting statements we turn our attention towards the assignment. Clearly, the state has to be modified, because the value of the assigned variable x might have changed.

$$\mathcal{M}(\mathsf{x} := \mathsf{e})(\sigma) = \{(\sigma', \langle\rangle) \mid \exists v.v = \mathcal{E}(e)\sigma, \sigma' = (\sigma : x \to v)\}$$

(By $(\sigma : x \to v)$ we mean the modification, or extension, of σ at x with v.) Thus the set contains the state σ' which equals the old state σ up to the value for x. Here we write $\mathcal{E}(e)\sigma$ for the evaluation of expression e with respect to the state σ. If $\mathcal{E}$ discovers a division by zero, it will not return any value, leading to an empty set, which is precisely why we existentially quantified the value v. This demonstrates that our semantics models 'abnormal' termination by non-termination.

For the sequential composition of statements S_1 and S_2 (which might be in turn sequences of statements) we give the following semantics.

$$\mathcal{M}(S_1;S_2)(\sigma) = \{(\sigma_2, \theta_1^\frown\theta_2) \mid \exists\sigma_1.(\sigma_1, \theta_1) \in \mathcal{M}(S_1)(\sigma),\ (\sigma_2, \theta_2) \in \mathcal{M}(S_2)(\sigma_1)\}$$

A more involved form of sequential composition is the loop statement.

$$\mathcal{M}(\textbf{while } b \textbf{ do } S \textbf{ end})(\sigma_0) = \left\{ (\sigma_k, \theta) \;\middle|\; \begin{array}{l} \exists k \in \mathbb{N}, (\sigma_1, \theta_1), \ldots, (\sigma_k, \theta_k) \text{ such that} \\ \theta = \theta_1^\frown \ldots ^\frown \theta_k,\ \mathcal{B}(\neg b)\sigma_k, \\ \text{for } i = 0, \ldots, k-1 :\ \mathcal{B}(b)\sigma_i,\ (\sigma_{i+1}, \theta_{i+1}) \in \mathcal{M}(S)(\sigma_i) \end{array} \right\}$$

We consider $k-1$ executions of the loop body S, where every time the body is started in an end-state of the previous execution. So the boolean condition b has to hold before those $k-1$ repetitions being expressed by $\mathcal{B}(b)\sigma_i$. Finally, when the entry condition b is checked the k-th time, we have $\mathcal{B}(\neg b)\sigma_k$. The history of all those runs is concatenated as in the previous semantics for sequential execution. If the loop is non-terminating there is no such k, leading to an empty set. The remaining sequential statements are given in Appendix B.

We continue with object internal parallelism via shared memory. The **release** statement allows another thread to run.

$$\mathcal{M}(\textbf{release})(\sigma) = \{(\sigma', \langle yield(\sigma|_a)\rangle^\frown\langle resume(\sigma'|_a)\rangle) \mid \sigma'|_l = \sigma|_l\} \tag{1}$$

The compositional semantics stores a *yield-resume* pair here, in order to mark the points where later, when merging 'parallel' histories, a thread switch is allowed. Locally, we mimic the effect of other threads by allowing all possible values of the object attributes, while the local variables are preserved in the new state σ'. Later, when we compose the histories of several threads, we will only allow the switch of control from a *yield* to the *resume* of some thread such that both store the same attribute values, see equations (7) and 8.

The other form of a releasing statement is the **await** statement where execution is released[4], and only continued if a condition b is fulfilled which is the meaning of $\mathcal{B}(b)\sigma'$.

$$\mathcal{M}(\textbf{await } b)(\sigma) = \{(\sigma', \langle yield(\sigma|_a)\rangle^\frown\langle resume(\sigma'|_a)\rangle) \mid \sigma'|_l = \sigma|_l, \mathcal{B}(b)\sigma'\} \tag{2}$$

await can also check for the termination of another thread, using the condition 'l?'. Here, a little complication is added to the semantics. The message $\langle comp(\mathcal{E}(l)\sigma, \bar{v})\rangle$ will later, in the parallel composition of histories, serve as an assertion to the history that the thread serving the call belonging to the label l is finished. (The return values $\bar{v}$ remain unused here in contrast to equation (4)).

$$\mathcal{M}(\textbf{await } l?)(\sigma) = \{(\sigma', \langle yield(\sigma|_a)\rangle^\frown\langle resume(\sigma'|_a)\rangle^\frown\langle comp(\mathcal{E}(l)\sigma', \bar{v})\rangle) \mid \sigma'|_l = \sigma|_l\} \tag{3}$$

Whether the callee thread really terminated cannot be checked here, as it would break the compositionality, and is therefore addressed in the composition of several objects to the semantics of a program run (see equation (12)).

Next, we turn our attention towards inter-object parallelism using message passing. Even though the communication is asynchronous in the sense that the invocation of a method is separated from the retrieval of its return parameters, the method invocation is modelled as an atomic *bidirectional* communication event. The caller provides the method name and the corresponding parameters and receives in turn the id of the thread assigned to accomplish the work.

$$\mathcal{M}(l!o.m(\bar{x}))(\sigma) = \left\{ (\sigma_1, \theta) \;\middle|\; \begin{array}{l} \exists i.\exists oid.oid = \mathcal{E}(o)\sigma, \\ \sigma_1 = (\sigma : l \rightarrow ((\mathcal{E}(this)\sigma, \mathcal{E}(me)\sigma)), (oid, (m, i))), \\ \theta = \langle invoc(\mathcal{E}(l)\sigma_1, \mathcal{E}(\bar{x})\sigma)\rangle \end{array} \right\}$$

By quantifying oid we ensure that the result of $\mathcal{E}(o)\sigma$ is not *null* if the set is non-empty. In the semantics we quantify existentially over the thread id i, which completes the identity $(oid, (m, i))$ of the callee. During the composition in equation (12) we will require the thread id to match the id of the actual communication partner. We identify the pair of current object and thread as the caller, where both *this* and *me* are special variables where the former is a class attribute keeping the object id and the latter is a thread-local variable storing its thread id. The pair of the identifier of the caller and the callee is assigned to the label. Because both identifiers consist of the object id and thread id, they uniquely determine the communication partners. The representation of the communication event in the history is the invocation message containing the label and the values of the method parameters $\mathcal{E}(\bar{x})\sigma$. The reply statement, which is the counterpart of the invocation, uses the same label to identify the corresponding completion message.

$$\mathcal{M}(l?(\bar{y}))(\sigma) = \{(\sigma_1, \theta) \mid \exists \bar{v}.\exists lv.lv = \mathcal{E}(l)\sigma, \sigma_1 = (\sigma : \bar{y} \rightarrow \bar{v}), \theta = \langle comp(lv, \bar{z})\rangle\} \tag{4}$$

Besides the label the completion message contains the values $\bar{v}$ of the return parameters of the method call, which are used to update the state as well. Similar to the previous definition we deal with the exception of the label l being uninitialised by the existence quantifier $\exists lv$. It is worthwhile to accentuate that the history we are constructing here, is purely thread local and will be composed to the semantics of the complete program hereafter. The last statement to cover is object creation. As reference to the object the pair (C, i) composed of the class C and an integer i is written to the variable o. Uniqueness of i is assured later in the parallel composition.

$$\mathcal{M}(o := \textbf{new } C)(\sigma) = \left\{ (\sigma_1, \theta) \;\middle|\; \begin{array}{l} \exists i.\sigma_1 = (\sigma : o \rightarrow (C, i)) \\ \theta = \langle new(\mathcal{E}(this)\sigma, \mathcal{E}(o)\sigma_1)\rangle \end{array} \right\} \tag{5}$$

[4]In other versions Creol, even the releasing is conditional.

In addition to the object id of the current object, the id of the newly created object is encoded in the *new* message.

Having dealt with single statements, we carry on with the semantics of a single thread as an instance of a method m.

$$\mathcal{M}(\textbf{op}\ m(\textbf{in}\ \bar{x};\ \textbf{out}\ \bar{y}) == body)(\sigma) = \left\{ (\sigma_2, \theta_1{}^\frown\theta^\frown\theta_2) \;\middle|\; \begin{array}{l} \exists\bar{v}.\sigma_1 = (\sigma : \bar{x} \to \bar{v}), (\sigma_2, \theta) \in \mathcal{M}(body)(\sigma_1), \\ o = \mathcal{E}(caller)\sigma, o_2 = (\mathcal{E}(this)\sigma, \mathcal{E}(me)\sigma) \\ \theta_1 = \langle resume(\sigma|_a)\rangle^\frown\langle begin((o, o_2), \bar{v})\rangle, \\ \theta_2 = \langle end((o, o_2), \mathcal{E}(\bar{y})\sigma_2)\rangle^\frown\langle yield(\sigma_2|_a)\rangle \end{array} \right\} \tag{6}$$

Some values $\bar{v}$ serve as the input parameters updating the initial state σ to σ_1 which in turn acts as input for the semantics of the method body. The resulting history θ includes all communication initiated by the method, but not its own invocation and completion message contained in θ_1 and θ_2. To model the asynchronism we write *begin* instead of *invoc* and *end* in place of *comp*. When composing the histories of caller and callee in equation (12) we will require that the *invoc* and *comp* messages of the caller frame the *begin* and *end* messages of the callee. In that way the point in time when a method is invoked is not necessarily the same point in time when the thread starts running. Additionally θ_1 starts with a *resume* as in general another thread might run in advance, and θ_2 ends with *yield* to allow other threads to work. Now, we are ready to compose the semantics of a number i of threads of the same method. Therefore we use another semantic function $\mathcal{M} : METHODS \times \mathbb{N} \to \Sigma \to 2^H$ which gives for a method m and a number of threads i the function relating the initial state to a set of histories. The initial state σ only consists of the value for *this* as defined in equation (10).

$$\mathcal{M}(m, i)(\sigma) = \left\{ \theta \;\middle|\; \begin{array}{l} \forall j \in \{1, \ldots, i\}.t = (m, j), \\ \exists o.\exists\bar{v}.\sigma_t = (\sigma : me \to t, caller \to o, \overline{ca} \to \bar{v}), \\ (\theta_t, \sigma'_t) \in \mathcal{M}(\textbf{op}\ m(\textbf{in}\ \bar{x};\ \textbf{out}\ \bar{y}) == m.body)(\sigma_t), \\ \theta \downarrow \{t\} = \theta_t, \theta \downarrow \{(m, j) \mid j \in \{1, \ldots, i\}\} = \theta, cond(\theta) \end{array} \right\} \tag{7}$$

First, we create i different thread ids t and corresponding initial states σ_t for each of them. Except for the thread id saved in *me* and the reference to the caller, the states differ in the class attributes $\overline{ca}$ as the threads start executing after some other thread yielded leaving the class attributes with any values. The second line of the conditions delivers the possible histories θ_t and states σ'_t the semantics of each thread can result in. Then we describe the *merging* θ of the histories θ_t, as the inverse of *projection*, as in [60]. The projection on sets of threads T removes the messages not involving any thread in T.

$$\langle\rangle \downarrow T = \langle\rangle \qquad (\langle m\rangle^\frown\theta) \downarrow T = \begin{cases} \langle m\rangle^\frown(\theta \downarrow T) & \text{if } m.caller.tid \in T \text{ or } m.callee.tid \in T \\ \theta \downarrow T & \text{otherwise} \end{cases}$$

So by writing $\theta \downarrow \{t\} = \theta_t$ we require that all messages of θ_t are contained in θ in the correct order. To restrict θ to messages which are part of any history θ_t, we write $\theta \downarrow \{(m, j) \mid j \in \{1, \ldots, i\}\} = \theta$. The notion of $cond(\theta)$ abbreviates the following condition[5] making sure that a switch between two different histories θ_t in θ is only possible at the release points of the methods.

$$\forall\langle m_1\rangle^\frown\langle m_2\rangle \subset \theta.m_1 \in \theta_{t_1}, m_2 \in \theta_{t_2}, t_1 \neq t_2 \Rightarrow m_1 = yield(.), m_2 = resume(.) \tag{8}$$

We marked the release points in the equations (1), (2), and (3) by the *yield*-*resume* pair. Now, the histories of the threads can be split up in between those messages, and only there, as formally described by the above formula. The next step is to introduce the semantics of a method being the union over all possible numbers of threads.

$$\mathcal{M}(m)(\sigma) = \bigcup_{i \in \mathbb{N}} \mathcal{M}(m, i)(\sigma) \tag{9}$$

[5] $\theta_1 \subset \theta_2$ iff $\exists\theta', \theta''.\theta'^\frown\theta_1{}^\frown\theta'' = \theta_2$

The step is necessary as the number of threads can depend on the parameters of the program. By considering all methods m of the class C instantiated by the object o we obtain the semantics of an object o.

$$\mathcal{M}(o) = \left\{ \theta'' \;\middle|\; \begin{array}{l} \forall m \in C.\theta_m \in \mathcal{M}(m)(\mathit{this} \to o), \\ \theta \downarrow \{m\} = \theta_m, \theta \downarrow \{m \mid m \in C\} = \theta, \\ \theta' = \langle \mathit{yield}(\mathcal{M}(\mathit{constr}_C))\rangle^\frown \theta, \mathit{cond}(\theta') \\ \exists o'.\theta'' = \langle \mathit{created}(o', o)\rangle^\frown \theta' \downarrow_{MP} \end{array} \right\} \tag{10}$$

The composition essentially works the same way as for threads of the same method (equation (7)), explained by the fact that all threads are communicating using the same principle, namely shared memory. So we retrieve a possible history for each method where the state exclusively contains the object identifier o accessible by *this*. The projections maintain the property that every message of all histories θ_m is included in θ, but no more. The *yield* message, preceding θ, stores the result of the constructor of C, executing only assignments to class attributes. $\mathit{cond}(\theta')$ furthermore restricts the class attributes at thread switches to match:

$$\mathit{cond}(\theta') \quad \equiv \quad \forall \langle \mathit{yield}(\sigma')\rangle^\frown \langle \mathit{resume}(\sigma'')\rangle \subset \theta' \Rightarrow \sigma' = \sigma''$$

The resulting history θ'' starts with the object creation message *created* which will be enforced in equations (12), (13) to occur after the corresponding *new* message of the creator (see equation (5)). Finally, the shared memory concurrency is hidden by removing all related messages from θ' by the following projection (where $_{MP}$ reads as projection to method passing):

$$\langle\rangle \downarrow_{MP} = \langle\rangle \qquad (\langle m\rangle^\frown \theta) \downarrow_{MP} = \begin{cases} \langle m\rangle^\frown (\theta \downarrow_{MP}) & \text{if } m = \mathit{yield}(.) \text{ or } m = \mathit{resume}(.) \\ \theta \downarrow_{MP} & \text{otherwise} \end{cases}$$

Analogous to the semantics of methods (equation (7)) we define the semantics of a class C with respect to a given number of objects i.

$$\mathcal{M}(C, i) = \left\{ \theta \;\middle|\; \begin{array}{l} \forall j \in \{1, \ldots, i\}.o = (C, j), \theta_o \in \mathcal{M}(o), \\ \theta \downarrow \{o\} = \theta_o, \theta \downarrow \{(C, j) \mid j \in \{1, \ldots, i\}\} = \theta \end{array} \right\} \tag{11}$$

This is a case of message passing parallelism, which we can describe by using the same projections as before, but this time without any requirement on release points as equation (8) for shared memory parallelism. (Note that *yield* and *resume* are not contained anymore in $\mathcal{M}$(o).) Therefore, the only condition on θ is that it contains exactly all messages of the histories θ_o and the orderings of their messages are preserved, allowing all possible interleavings. We define the semantics of a class as the union over the number of instances.

$$\mathcal{M}(C) = \bigcup_{i \in \mathbb{N}} \mathcal{M}(C, i)$$

What is left to describe is the semantics of a complete program. This is more involved, as it is only here where we require the various communication histories to actually be consistent with each other, in order to be merged to a global one. As a tool we will reason about the set of messages of a certain type contained in a history, to be obtained by the following function.

$$\mathit{get}_{\mathit{type}}(\langle m\rangle^\frown \theta) = \begin{cases} \mathit{get}_{\mathit{type}}(\theta) \cup \{m\} & \text{if } m \text{ has type } \mathit{type} \\ \mathit{get}_{\mathit{type}}(\theta) & \text{otherwise} \end{cases}$$

type will be a name of a message type, like *invoc* or *end*. Another prerequisite is a Boolean function determining the order of two given messages within the history.

$$m_1 <_\theta m_2 \quad \equiv \quad \exists \theta_1, \theta_2, \theta_3.\ \theta = \theta_1^\frown \langle m_1\rangle^\frown \theta_2^\frown \langle m_2\rangle^\frown \theta_3$$

Finally, we turn to the actual semantics of a program P, given by a set of histories.

$$\mathcal{M}(P) = \left\{ \theta' \;\middle|\; \begin{array}{l} \forall C \in P.\theta_C \in \mathcal{M}(C), \theta \downarrow \{C\} = \theta_C, \theta \downarrow \{C \mid C \in P\} = \theta \\ \exists o.Main = o.class, \exists i.\exists \bar{v}, \theta' = \langle new(., o)\rangle ^\frown \langle invoc((., (main, i)), \bar{v})\rangle ^\frown \theta \\ \forall type.\forall \theta_1.\forall \theta_2.\theta' = \theta_1 ^\frown \theta_2 \Rightarrow get_{type}(\theta_1) \cap get_{type}(\theta_2) = \{\} \\ \text{exists a bijection } f : get_{new}(\theta') \rightarrow get_{created}(\theta'), cond(f) \\ \text{exists a bijection } g : get_{invoc}(\theta') \rightarrow get_{begin}(\theta'), cond(g) \\ \text{exists a function } h : get_{comp}(\theta') \rightarrow get_{end}(\theta'), cond(h) \end{array} \right\} \tag{12}$$

The history θ is composed of the histories θ_C given by all the classes C of the program P. Following the same lines as the previous compositions, e.g. the one for message passing in equation 11, the merging is described as the inverse of projection. We assume that every program has a class called *Main* which contains a method called *main*. This object is created initially as denoted by the $\langle new(., o)\rangle$ message and its method *main* is invoked afterwards which is the meaning of the message $\langle invoc((., (main, i)), \bar{v})\rangle$. As we were existentially quantifying about parameters of messages there could be identical messages in the history, which is prohibited by the third line. Now, a valid history θ' must ensure that every new object was created, which is achieved by the bijection f. In terms of messages it must hold, that a message $\langle created(o_1, o_2)\rangle$ of the history of the caller o_1 (see equation 10) is always preceded by the creation message $\langle new(o_1, o_2)\rangle$ of the callee o_2 (described in equation 5). Formally, we express this fact, which we abbreviated as $cond(f)$, as:

$$\forall x.\forall y.f(x) = y \Rightarrow x.caller = y.caller, x.callee = y.callee, x <_{\theta'} y \tag{13}$$

A positive side-effect of the above formula is that the resulting class hierarchy is cycle-free. The next function g does a similar job in relating a *begin* message to every *invoc* message. Analogous to equation 13 we ensure within $cond(g)$ that *invoc* appears in the history before its related *begin* message. The last function h connects every *comp* message with an *end* message ordering the first before the latter by $cond(h)$. As is it not necessary to ask for the result of a method h, may not be surjective. On the other hand the return values can be checked several times for a single method call, so in general the function h is not injective. In total for every method call the messages *invoc* and *comp* (if it exists) frame the pair of *begin*, *end* introduced by equation 6. Thus the callee starts executing after the message call and the caller receives its parameters after the callee has terminated, as one would expect. Using the functions f, g, and h we induced a partial ordering on messages within histories representing asynchronous communication which makes this approach comparable to the classical work of [30].

4. The KeY approach: Logic, Calculus, and System

4.1. Dynamic Logic with Explicit Substitutions

KeY is a deductive verification system for *functional correctness*. Its core is a theorem prover for formulas in *dynamic logic* (DL) [34], which, like Hoare logic [35], is transparent with respect to the programs that are subject to verification. DL is a particular kind of *modal logic*. Different parts of a formula are evaluated in different worlds (states), which vary in the interpretation of functions and predicates. The modalities are 'indexed' with pieces of program code, describing how to reach one world (state) from the other. DL extends full first-order logic with two additional (mix-fix) operators: $\langle\,.\,\rangle\,.$ (diamond) and $[\,.\,]\,.$ (box). In both cases, the first argument is a *program* (fragment), whereas the second argument is another DL formula. A formula $\langle p\rangle\phi$ is true in a state s if there is a terminating run of p, started in s, which results in a state where ϕ is true. As for the other operator, a formula $[p]\phi$ is true in a state s if all terminating runs of p, started in s, result in a state where ϕ is true. This implies that $\langle p\rangle\phi$ claims termination, whereas $[p]\phi$ does not.

DL is closed under all logical connectives. For instance, in a DL formula $\langle p\rangle\phi$, the postcondition ϕ may by any DL formula again, like in $\langle p\rangle\langle q\rangle\psi$. Also, arbitrary connectives can enclose box or diamond. For instance, the following formula states equivalence of p and q w.r.t. the "output", the program variable x.

$$\forall\ v.\ (\ \langle p\rangle\ \mathsf{x} \doteq v\ \leftrightarrow\ \langle q\rangle\ \mathsf{x} \doteq v\ ) \tag{14}$$

$$\textsf{impRight}\ \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \rightarrow \psi, \Delta} \quad \textsf{andRight}\ \frac{\Gamma \vdash \phi, \Delta \qquad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \land \psi, \Delta} \quad \textsf{allRight}\ \frac{\Gamma \vdash \phi[v/c], \Delta}{\Gamma \vdash \forall v.\phi, \Delta}$$

with c a new constant

Figure 4: A selection of first-order rules

Our version of dynamic logic distinguishes strictly between logical variables and program variables. Logical variables (like v in 14), can be quantified over, they cannot appear in programs, and their meaning does not vary among different states. Program variables (like x in 14), on the other hand depend on the state, but cannot be quantified over. Expressions in the logic, outside the box or diamond modality, can contain both types of variables.

A frequent pattern of DL formulas is $\phi \rightarrow \langle p \rangle \psi$, stating that the program p, when started from a state satisfying ϕ, terminates with ψ being true afterwards. The formula $\phi \rightarrow [p]\psi$, on the other hand, does not claim termination, and corresponds to the Hoare triple $\{\phi\}\ p\ \{\psi\}$.

The main advantage of DL over Hoare logic is increased expressiveness: pre- or postconditions can contain programs themselves, for instance to express that a linked structure is acyclic. Also, the relation of different programs to each other (like the correctness of transformations) can be expressed elegantly, see 14. but also security properties [20, 49]

All major program logics (Hoare logic, wp calculus, DL) have in common that the resolving of assignments requires substitutions in the formula, in one way or the other. In the KeY approach, the effect of substitutions is delayed, by having *explicit substitutions* in the logic, called *'updates'*. This allows for accumulating and simplifying the effect of a program, in a forward style. Elementary updates have the form $x := e$, where x is a location (in the case of Creol, an attribute or local variable) and e is a (side-effect free) expression. Elementary updates are combined to simultaneous updates, like in $x_1 := e_1 \,|\, x_2 := e_2$, where e_1 and e_2 are evaluated in the same state. For instance, $\mathsf{x} := \mathsf{y} \,|\, \mathsf{y} := \mathsf{x}$ stands for exchanging the values of x and y. Updates are brought into the logic via the update modality $\{\,.\,\}\,.$, connecting arbitrary updates with arbitrary formulas, like in $\mathsf{x} < \mathsf{y} \rightarrow \{\mathsf{x} := \mathsf{y} \,|\, \mathsf{y} := \mathsf{x}\}\,\mathsf{y} < \mathsf{x}$. A typical usage of updates during proving is in formulas of the form $\{\mathcal{U}\}\langle p \rangle \phi$, where $\mathcal{U}$ is an update, accumulating the effects of program execution up to a certain point, p is the remaining program yet to be executed, and ϕ a postcondition. A full account of KeY style DL is found in [14].

4.2. Sequent Calculus

The heart of KeY, the prover, uses a sequent calculus for reducing proof obligations to axioms. A sequent is a pair of sets of formulas written as $\phi_1, ..., \phi_m \vdash \psi_1, ..., \psi_n$. A sequent is valid if validity of all $\phi_1, ..., \phi_m$ implies validity of at least one of $\psi_1, ..., \psi_n$. We use capital Greek letters to denote (possibly empty) sets of formulas. For instance, by $\Gamma \vdash \phi \rightarrow \psi, \Delta$ we mean a sequent containing at least an implication formula on the right side. Sequent calculus rules always have one sequent as conclusion and zero, one, or more sequents as premises:

$$\frac{\Gamma_1 \vdash \Delta_1 \ \ldots\ \Gamma_n \vdash \Delta_n}{\Gamma \vdash \Delta}$$

Semantically, a rule states that the validity of all n premises implies the validity of the conclusion ('top-down'). Operationally, rules are applied bottom-up, reducing the provability of the conclusion to the provability of the premises. In Fig. 4 we present a selection of the rules dealing with propositional connectives and quantifiers (see [33] for the full set). $\phi_{\bar{x}}^{\bar{e}}$ denotes a formula resulting from replacing the variables $\bar{x}$ with expressions $\bar{e}$ in ϕ.

When it comes to the rules dealing with programs, many of them are not sensitive to the side of the sequent and can even be *applied to subformulas.* For instance, $\langle \textbf{skip};\ \omega \rangle \phi$ can be rewritten to $\langle \omega \rangle \phi$ regardless of where it occurs. For that we introduce the following syntax

$$\frac{\phi'}{\phi}$$

$$\textsf{assign}\ \frac{\{x := e\}⟪\omega⟫\phi}{⟪x := e;\, \omega⟫\phi} \qquad \textsf{if}\ \frac{(b \rightarrow ⟪s_1;\, \omega⟫\phi) \wedge (\neg b \rightarrow ⟪s_2;\, \omega⟫\phi)}{⟪\textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{end};\, \omega⟫\phi}$$

$$\textsf{unwind}\ \frac{⟪\textbf{if}\ b\ \textbf{then}\ s;\ \textbf{while}\ b\ \textbf{do}\ s\ \textbf{end}\ \textbf{end};\, \omega⟫\phi}{⟪\textbf{while}\ b\ \textbf{do}\ s\ \textbf{end};\, \omega⟫\phi}$$

Figure 5: Dynamic logic rules

where ϕ and ϕ' are formulas (i.e., there is no sequent arrow $\vdash$). This denotes a rule where the (only) premise sequent is constructed by replacing ϕ with ϕ' *anywhere* in the conclusion sequent ϕ. In Fig. 5 we present some rules dealing with statements. (assign and if are simplified, see Sect. 6.1.) The schematic modality $⟪\cdot⟫$ can be instantiated with both $[\cdot]$ and $\langle\cdot\rangle$, though consistently within a single rule application. Total correctness formulas of the form $\langle$**while** ...$\rangle\phi$ are proved by combining induction with unwind.

Because updates are essentially delayed substitutions, they are eventually resolved by application to the succeeding formula, e.g., $\{u := e\}(u > 0)$ leads to $e > 0$. Update application is only defined on formulas *not* starting with box or diamond. For formulas of the form $\{\mathcal{U}\}\langle s\rangle\phi$ or $\{\mathcal{U}\}[s]\phi$, the calculus first applies rules matching the first statement in s. This leads to nested updates, which are in the next step merged into a single simultaneous update. Once the box or diamond modality is completely resolved, the entire update is applied to the postcondition.

5. Semantics of Creol Dynamic Logic

Syntactically, we arrive at Creol dynamic logic simply by having Creol statements within the modalities box and diamond. However, we need to significantly extend the meaning of formulas, to be able to *compositionally* verify programs, one operation at a time. The various operations of one class rely on each other respecting the *class invariant* for the shared variable concurrency to function correctly. Also, caller objects rely on the callees *interface invariant*. Correctness of Creol code must therefore include that this invariants are respected. We formalise in the following what this means exactly.

In order to evaluate formulas, we need the following semantic artifacts: a state σ, i.e., an assignment of object attributes and local variables to values, a (semantic) history θ, and an assignment of logical variables γ. Further, *IInv* and *CInv* map interfaces and classes to their invariant, respectively. Interface invariants do not 'talk' about attributes or local variables. Instead, they talk about the history, for which we use the reserved symbol $\mathcal{H}$, which is interpreted by the given semantic history (typically θ). Moreover, in interface invariants there appear the logical variables *caller*[6] and *callee*, typically as argument of the projection operator, see Sect. 6.2. Finally, in place of class invariants, code is ever only correct relative to a class, C.

For notational simplicity, we assume each method m to appear in exactly one interface. (This can always be achieved by renaming or qualifying of methods, and if necessary, cloning of the method bodies.) The function $\mathit{intf}(m)$ returns the interface m belongs to.

The following sets of histories are used in the definition of formula semantics.

$$\mathit{commit}_{\mathit{IInv},\gamma} = \left\{ \theta \;\middle|\; \begin{array}{l} \text{if } \theta = \theta_0 ^\frown \langle \mathit{invoc}((\mathit{oid}, t), (\mathit{oid}', (m, i)), \bar{v})\rangle \\ \text{then } (\theta, \gamma) \models \mathit{IInv}(\mathit{intf}(m))^{\mathit{oid},\mathit{oid}'}_{\mathit{caller},\mathit{callee}} \end{array} \right\}$$

$$\mathit{assume}_{\mathit{IInv},\gamma} = \left\{ \theta \;\middle|\; \begin{array}{l} \text{if } \theta = \theta_0 ^\frown \langle \mathit{comp}((\mathit{oid}, t), (\mathit{oid}', (m, i)), \bar{v})\rangle \\ \text{then } (\theta, \gamma) \models \mathit{IInv}(\mathit{intf}(m))^{\mathit{oid},\mathit{oid}'}_{\mathit{caller},\mathit{callee}} \end{array} \right\}$$

$$\mathit{guarantee}_{\mathit{CInv},C,\gamma} = \left\{ \theta \;\middle|\; \begin{array}{l} \text{if } \theta = \theta_0 ^\frown \langle \mathit{yield}(\sigma)\rangle \\ \text{then } (\sigma, \theta, \gamma) \models \mathit{CInv}(C) \end{array} \right\}$$

[6] The logical variable *caller* is related, but not identical to the implicit local variable in Sect. 3.

$$rely_{CInv,C,\gamma} = \left\{ \theta \;\middle|\; \begin{array}{l} \text{if } \theta = \theta_0 ^\frown \langle resume(\sigma) \rangle \\ \text{then } (\sigma, \theta, \gamma) \models CInv(C) \end{array} \right\}$$

With help of these sets, we can define the semantics of (the base case of) dynamic logic formulas, relative to an *initial* state σ, an *initial* history θ:

$$(\sigma, \theta, \gamma, IInv, CInv, C) \models [S]\varphi$$

iff

for all $(\sigma_1, \theta_1) \in \mathcal{M}(S)(\sigma)$:
if
$\{\theta' \mid \theta' \leq \theta ^\frown \theta_1\} \subseteq assume_{IInv,\gamma} \cap rely_{CInv,C,\gamma}$
then
$\{\theta'' \mid \theta'' \leq \theta_1\} \subseteq commit_{IInv,\gamma} \cap guarantee_{CInv,C,\gamma}$ and $(\sigma_1, \theta ^\frown \theta_1, \gamma, IInv, CInv, C) \models \varphi$

Intuitively, this means that S (if executed in class C) has to *commit* to the invariants of interfaces it calls, and has to *guarantee* the class invariant of C at release points, and has to establish φ on termination. For that, S can *assume* the invariant of replying interfaces (both before and during S), and can *rely* on other threads of *this* object to establish the invariant of C when releasing control (both before and during S). The definition combines assumption-commitment style reasoning [48], adapted to asynchronous method calls, with rely-guarantee style reasoning [44], adapted to object local cooperative parallelism. See the discussion in section 9.

Note that $[S]\varphi$ does not claim termination, as the set $\mathcal{M}(S)(\sigma)$ of terminating runs is allowed to be empty. On the other hand, $(\sigma, \theta, \gamma, IInv, CInv, C) \models \langle S \rangle \varphi$ is defined by replacing "for all $(\sigma, \theta) \in \mathcal{M}(S)(\sigma)$" with "there exists $(\sigma, \theta) \in \mathcal{M}(S)(\sigma)$" in the above definition. Thereby, we claim, among other things, the existence of a terminating run.

The semantics of Boolean connectives and quantifiers is defined as in first-order logic. As for the update operator, its semantics is straight-forward:

$$(\sigma, \theta, \gamma, IInv, CInv, C) \models \{x_1 := e_1 \,|\, \ldots \,|\, x_n := e_n\}\varphi$$

iff

$$((\sigma : x_1 \to \mathcal{E}(e_1)\sigma : \ldots : x_n \to \mathcal{E}(e_n)\sigma), \theta, \gamma, IInv, CInv, C) \models \varphi$$

Now, *validity* of formulas, in the context of invariants *IInv*, *CInv*, and class C, is defined by:

$$(IInv, CInv, C) \models \varphi$$

iff

$$\text{for all } \sigma, \theta, \gamma : \; (\sigma, \theta, \gamma, IInv, CInv, C) \models \varphi$$

6. A Calculus for Creol Dynamic Logic

Building on the logic and the calculus presented in the previous sections, we proceed with the sequent rules handling Creol statements. For the full set of rules, see [28].

6.1. Sequential Constructs

We start with assignments. As soon as the right side is simply a variable or literal (summarised as 'terminal expression', te) the assignment can be transformed to an update, such that the effect will eventually (not immediately) be applied to the postcondition. The same applies for implicit assignments in variable declarations. We give only the rule for integer variable declaration.

$$\textsf{assign}\ \frac{\{x := te\}\langle\!|\omega|\!\rangle\phi}{\langle\!|x := te;\ \omega|\!\rangle\phi} \qquad \textsf{intDecl}\ \frac{\{i := 0\}\langle\!|\omega|\!\rangle\phi}{\langle\!|\textbf{var}\ i : \textbf{Int};\ \omega|\!\rangle\phi}$$

The same mechanism can be used for operator expressions, as long as all arguments are terminal *and* errors can be excluded. For instance, a division can be shifted to an update iff the divisor is not zero. Otherwise, execution blocks. This semantics is captured by the following rule.

$$\textsf{DivTerminal}\ \frac{(\neg te_2 \doteq 0 \to \{x := te_1/te_2\}\langle\!|\omega|\!\rangle\phi) \wedge (te_2 \doteq 0 \to \langle\!|\textbf{block};\ \omega|\!\rangle\phi)}{\langle\!|x := te_1/te_2;\ \omega|\!\rangle\phi}$$

An error could occur arbitrary deep in an expression. Therefore, expressions are unfolded until they consist only of a top level operator applied to terminal expressions. This is exemplified by the following rules (x' and x'' are new program variables).

$$\frac{\langle\!|x' := e_1;\ x'' := e_2\ ;\ x := x' + x'';\ \omega|\!\rangle\phi}{\langle\!|x := e_1 + e_2;\ \omega|\!\rangle\phi} \qquad \frac{\{x := te_1 + te_2\}\langle\!|\omega|\!\rangle\phi}{\langle\!|x := te_1 + te_2;\ \omega|\!\rangle\phi}$$

In the left rule e_i are non-terminal expressions. As all expressions are unfolded every division will eventually be analysed by DivTerminal. Other statements using expressions, like **if**, are unfolded in the same way, until the condition is terminal and the following rule applies:

$$\textsf{if}\ \frac{(tb \doteq \textsf{true} \to \langle\!|p;\ \omega|\!\rangle\phi) \wedge (tb \doteq \textsf{false} \to \langle\!|q;\ \omega|\!\rangle\phi)}{\langle\!|\textbf{if}\ tb\ \textbf{then}\ p\ \textbf{else}\ q\ \textbf{end};\ \omega|\!\rangle\phi}$$

Note that application of this rule may lead to proof branching in subsequent steps. As for **while**, the unwind rule was presented in Sect. 4.2. An alternative rule using a loop invariant is discussed in section 6.3. That rule, however, only covers the box operator. Finally, the rules for the **block** statement reflect the fact that a non-terminating program is always partially correct, but never totally correct:

$$\textsf{blockBox}\ \frac{true}{[\textbf{block};\ \omega]\phi} \qquad \textsf{blockDia}\ \frac{false}{\langle\textbf{block};\ \omega\rangle\phi}$$

6.2. Interface and Class Invariants

The verification process of Creol programs is compositional. This means we verify only one method (of one class) at the time and do not consider any other code during this process. Instead, we take into account the other threads of the object by guaranteeing the class invariant at release points and relying on it it again when execution proceeds. As for the behaviour of other objects, that is represented by using the invariants of their interfaces. An additional construct in the proof is the communication history, which both the specifications as well as the class invariants talk about. These concepts for reasoning about Creol were introduced in [25, 27].

Every interface is specified by an interface invariant $IInv(I)(\mathcal{H})$. (Strictly, $IInv(I)$ is a formula, and '$(\mathcal{H})$' only indicates that $\mathcal{H}$ appears in that formula.) For the reasoning to be compositional, it is required that the system wide history $\mathcal{H}$ appears only projected to the logical variables *caller* and *callee*, like in $\mathcal{H}/\{caller, callee\}$, with '/' being the syntactical representation of the semantical projection '$\downarrow$'. This specification is used during verification at method calls and replies.

Continuing the previous example of Fig. 2 the interface USR is equipped with the following invariant:

$$\mathcal{H}/\{caller, callee\}/\to\ \ \le\ (\to \textsf{giveCode}[\cdot \to \textsf{withdraw}[\cdot \to \textsf{dispense}]]\cdot \to \textsf{returnCard})^*$$

where $/\rightarrow$ projects on invocation messages, $\cdot$ is appending, $\rightarrow$ are invocation messages, $\leftarrow$ are completion messages and brackets are used for optional occurrence. The parameters and communication partners are omitted for brevity. The invariant expresses that the history of this interface is always a *prefix* of this regular expression, such that an interaction with the user always begins with requesting PIN code and ends with requesting removal of the card. The interface S is specified by:

$$\mathcal{H}/\{caller, callee\} \leq \left(\rightarrow \mathsf{authorize}(cid, .) \cdot \left(\begin{array}{l} \leftarrow \mathsf{authorize}(false) | \\ (\leftarrow \mathsf{authorize}(true) \cdot \rightarrow \mathsf{debit}(cid, .) \cdot \leftarrow \mathsf{debit}(.)) \end{array} \right)\right)^*$$

Communication partners are omitted. The dot '.' is used as a wildcard for a parameter. Parameters (including the card id *cid*) and communication partners are quantified universally. The meaning of the invariant is that only after authorisation the debit procedure can be attempted.

We turn to the class invariant $CInv(C)(\mathcal{H}, \overline{\mathcal{W}})$, which forms a contract between all threads of the object. $\overline{\mathcal{W}}$ is the vector of class attributes. Those might get overwritten by other threads during suspension of this thread, but the invariant expresses properties of $\overline{\mathcal{W}}$ every thread is supposed to respect. A class invariant consists of several parts:

$$CInv(C) \triangleq \begin{array}{l} F(\mathcal{H}, \overline{\mathcal{W}}) \\ \wedge \ \bigwedge_{I \text{implemented}} IInv(I)(\mathcal{H})^{\mathsf{this}}_{callee} \\ \wedge \ \bigwedge_{I \text{invoked}} IInv(I)(\mathcal{H})^{\mathsf{this}}_{caller} \end{array}$$

$F(\mathcal{H}, \overline{\mathcal{W}})$ relates the state of the ordinary class attributes $\overline{\mathcal{W}}$ with the history, reflecting the refinement of the fully abstract interface specification to the local state. Then, all invariants of all interfaces I invoked or implemented by the class of this put in a conjunction to ensure that all methods respect them. Now we can formulate the proof obligation for a method. The pre-condition is the class invariant, instantiated with a history ending on an invocation of the method. After executing the *body* the invariant holds again for the history ending with its completion message. For each method **op** $m($**in** $\bar{x}$; **out** $\bar{y})==$ *body* of class C and interface I, where $\overline{\mathcal{W}}$ are the attributes of class C we have the following *proof obligation*:

$$\begin{array}{c} (IInv, CInv, C) \models \\ IInv(I)(\mathcal{H})^{\mathsf{this}}_{callee} \wedge Wf(\mathcal{H}) \wedge Invoc(\mathcal{H}, (caller, \mathsf{this}), \bar{x}) \wedge CInv(C)(\mathcal{H}, \overline{\mathcal{W}}) \\ \rightarrow [body;\ \mathbf{return}(\bar{y})] IInv(I)(\mathcal{H})^{\mathsf{this}}_{callee} \wedge Wf(\mathcal{H}) \wedge CInv(C)(\mathcal{H}, \overline{\mathcal{W}}) \end{array} \tag{15}$$

$Wf(\mathcal{H})$ holding for well-formed histories. A well-formed history starts with the creation message of this, contains invocation messages for all completion messages, and does not include any object references being null.

Let us proceed with an example for a class invariant. For class ATM of Fig. 3, the formula F is:

$$F_{ATM}(\mathcal{H}, \overline{\mathcal{W}}) \triangleq \neg \mathsf{server} \doteq \mathsf{null} \wedge \forall cid.sum_{dis}(\mathcal{H}/cid) \doteq sum_{deb}(\mathcal{H}/cid)$$

It states that the reference server is never null and the sum of all withdrawn money for all cards *cid* equals the sum of the money debited. More detailed, $sum_{dis}(h)$ calculates the sum of the money withdrawn in the history h. (In the equations, *msg* is used as the 'otherwise case'.)

$$\begin{array}{rcl} sum_{dis}(\epsilon) & = & 0 \\ sum_{dis}(h \cdot \rightarrow \mathsf{withdraw}(am)) & = & sum_{dis}(h) + am \\ sum_{dis}(h \cdot msg) & = & sum_{dis}(h) \end{array}$$

$sum_{deb}(h)$ is the sum of the money debited from the corresponding bank account. Only successful debit calls are counted.

$$\begin{array}{rcl} sum_{deb}(\epsilon) & = & 0 \\ sum_{deb}(h \cdot \rightarrow \mathsf{debit}(am, cid) \cdot \leftarrow \mathsf{debit}(true)) & = & sum_{deb}(h) + am \\ sum_{deb}(h \cdot msg) & = & sum_{deb}(h) \end{array}$$

In the system such equations are realised as taclets (see Sect. 7).

6.3. Concurrent Constructs

There are two different levels of communication, namely inter-thread communication within one object via shared memory (the class attributes $\overline{\mathcal{W}}$) and inter-object communication via method calls and replies. We start with the rules concerning the first and focus on the latter further below.

The simplest form of a release point is **release**. As mentioned before the class invariant forms a contract between all threads of an object. So the rule for **release** forces us to show that the class invariant is established in the current state, before releasing the processor. When this thread resumes, the invariant can be assumed before the remaining code ω is executed.

$$\textsf{release}\ \frac{\Gamma \vdash \mathit{CInv}(\mathcal{H}, \overline{\mathcal{W}}), \Delta \qquad \Gamma \vdash \{U_{\mathcal{H},\overline{\mathcal{W}}}\}[\omega]\phi, \Delta}{\Gamma \vdash [\textbf{release};\ \omega]\phi, \Delta}$$

Here, $U_{\mathcal{H},\overline{\mathcal{W}}}$ is the update $\mathcal{H}, \overline{\mathcal{W}} := \textsf{home}\ H, \overline{W}.(\mathit{CInv}(H, \overline{W}) \wedge \mathcal{H} \leq H)$. This update represents an arbitrary but fixed system state, satisfying the class invariant, in which execution continues. This is necessary because values of the class attributes could have been overwritten by other threads. By $\mathcal{H} \leq H$ we denote that the old history $\mathcal{H}$ is a prefix of the new one H. Note that this rule, as well as all rules in this section, can also be applied when the modality is preceded by updates, which is the typical scenario. These updates are preserved in the instantiation of the premises (see [14]).

The **await** b statement is handled by a similar rule, with the additional assumption that the guard b holds when execution resumes. A minor complication is that we also must assume that evaluation of b does not block due to an error. The two assumptions together are expressed via $\langle x := b\rangle x \doteq \texttt{true}$.

$$\textsf{awaitExp}\ \frac{\Gamma \vdash \mathit{CInv}(\mathcal{H}, \overline{\mathcal{W}}), \Delta \quad \Gamma \vdash \{U_{\mathcal{H},\overline{\mathcal{W}}}\}(\langle x := b\rangle x \doteq \textsf{true} \rightarrow [\omega]\phi), \Delta}{\Gamma \vdash [\textbf{await}\ b;\ \omega]\phi, \Delta}$$

By replacing $\langle x := b\rangle x \doteq \textsf{true}$ with $\mathit{Comp}(\mathcal{H}, l)$ in the above rule, we get a rule for **await** $l?$. The predicate $\mathit{Comp}(\mathcal{H}, l)$ is valid if a completion message with the label l is contained in the history $\mathcal{H}$. The handling of $\mathit{Comp}(\mathcal{H}, l)$ in the proof is discussed further below.

Partial correctness of a loop can also be shown with help of a loop invariant $\mathit{inv}_{\mathit{loop}}(\mathcal{H}, \overline{\mathit{mod}})$, where $\overline{\mathit{mod}}$ is the modifier set of the loop (all variables assigned in the loop). To be most general, all class attributes could be included in the modifier set. The history could be omitted as a parameter of the loop invariant if there are no method calls, method completions or object creations in the loop body.

$$\textsf{loopInv}\ \frac{\begin{array}{ll} \Gamma \vdash \langle x{:=}b\rangle \mathit{true} \rightarrow \mathit{inv}_{\mathit{loop}}(\mathcal{H}, \overline{\mathit{mod}}), \Delta & \text{(init. valid)} \\ \Gamma \vdash \{U^{\mathit{loop}}_{\mathcal{H},\overline{\mathit{mod}}}\}(\langle x{:=}b\rangle x \doteq \textsf{true} \rightarrow [p]\mathit{inv}_{\mathit{loop}}(\mathcal{H}, \overline{\mathit{mod}})), \Delta & \text{(preserving)} \\ \Gamma \vdash \{U^{\mathit{loop}}_{\mathcal{H},\overline{\mathit{mod}}}\}(\langle x{:=}b\rangle x \doteq \textsf{false} \rightarrow [\omega]\phi), \Delta & \text{(use-case)} \end{array}}{\Gamma \vdash [\textbf{while}\ b\ \textbf{do}\ p\ \textbf{end};\ \omega]\phi, \Delta}$$

The update $U^{\mathit{loop}}_{\mathcal{H},\overline{\mathit{mod}}}$ is defined as:

$$\mathcal{H}, \overline{\mathit{mod}} := \textsf{some}\ H, \overline{m}.(\mathcal{H} \leq H \wedge \mathit{inv}_{\mathit{loop}}(H, \overline{m}))$$

It creates a new history H and a new modifier set, such that the loop invariant holds. If the condition b throws an exceptions, the implication of all branches are true.

Analogous to $\mathit{Comp}(\mathcal{H}, l, \bar{y})$ there are predicates $\mathit{Invoc}(\mathcal{H}, l, \bar{x})$ and $\mathit{New}(\mathcal{H}, o)$ which guarantee the existence of an invocation message with label l, parameters $\bar{x}$ and an object creation message with reference o in the history $\mathcal{H}$, respectively. To exemplify some properties of the predicates dealing with the history we give the following formula which is a tautology.

$$\mathit{Comp}(\mathcal{H}_0, l, \bar{y}) \wedge \mathcal{H}_0 \leq \mathcal{H}_1 \rightarrow \mathit{Comp}(\mathcal{H}_1, l, \bar{y}) \qquad (16)$$

Besides Comp, New, as well as Invoc are monotonous w.r.t. $\leq$. Additionally, the contra-position is used in our proof system.

We turn attention towards method invocation $l!o.mtd(\bar{x})$. Its execution assigns a unique reference to l, and extends the history by the corresponding invocation message:

$$\text{invoc}\ \frac{\begin{array}{l}\Gamma \vdash o \doteq \mathsf{null} \rightarrow \langle\!\langle \textbf{block};\ \omega \rangle\!\rangle \phi, \Delta \\ \Gamma \vdash \neg o \doteq \mathsf{null} \rightarrow \{l := (\mathsf{this}, o)\}\{U^{invoc}_{\mathcal{H}}\}(IInv(I)(\mathcal{H})^{\mathsf{this},o}_{caller,callee} \wedge \langle\!\langle \omega \rangle\!\rangle \phi, \Delta)\end{array}}{\Gamma \vdash \langle\!\langle l!o.mtd(\bar{x});\ \omega \rangle\!\rangle \phi, \Delta}$$

i is a new constant symbol[7]. If o is null, execution blocks. In the first branch, the invariant of the remote interface I must be shown (I being the type of o). The index i is new and assures uniqueness of the label l. The abbreviation $U^{invoc}_{\mathcal{H}}$ for the update, is in its full form:

$$\mathcal{H} := \mathsf{some}\ H.(\mathcal{H} \leq H \wedge Invoc(H, l, \bar{x}))$$

The new history contains the invocation message $Invoc(H, l)$.

A completion statement $l?(\bar{y})$ assigns the return parameters of the method call identified by the label l to $\bar{y}$. If the label l is null, execution blocks.

$$\text{comp}\ \frac{\begin{array}{c}\Gamma \vdash l \doteq \mathsf{null} \rightarrow [\textbf{block};\ \omega]\phi, \Delta \\ \Gamma \vdash \neg l \doteq \mathsf{null} \rightarrow \{U^{comp}_{\mathcal{H},\bar{y}}\}[\omega]\phi, \Delta\end{array}}{\Gamma \vdash [l?(\bar{y});\ \omega]\phi, \Delta}$$

As we are extending the history with a completion message, we check the existence of the corresponding invocation message by $Invoc(\mathcal{H}, l)$ to ensure well-formedness. $U^{comp}_{\mathcal{H},\bar{y}}$ is analogous to $U^{invoc}_{\mathcal{H}}$, where the only difference is that $\bar{y}$ is overwritten and $Comp$ is used instead of $Invoc$.

$$\mathcal{H}, \bar{y} := \mathsf{some}\ H, \bar{p}.(\mathcal{H} \leq H \wedge IInv(I)(H)^{\mathsf{this},l.callee}_{caller,callee} \wedge Comp(H, l, \bar{y}))$$

I is obtainable from the label l as it contains the method which was called.

We omit the rule for object creation, mentioning only that the new reference is constructed by the pair (this, i), here i is a object local, successively incremented index. An alternative, fully abstract modeling of object creation in DL is investigated in [4] and can be adapted also here. Finally, we consider the **return** statement. It sends the completion message belonging to the method call of the verification process and the thread terminates afterwards. The class invariant is not explicitly mentioned in the following rule as it is contained in ϕ (see previous section).

$$\text{return}\ \frac{\Gamma \vdash \{U^{return}_{\mathcal{H}}\}\phi, \Delta}{\Gamma \vdash \langle\!\langle \textbf{return}(\bar{y}) \rangle\!\rangle \phi, \Delta}$$

The update $U^{return}_{\mathcal{H}}$ adds the completion message to the history which must not occur in the previous history.

$$\mathcal{H} := \mathsf{some}\ H.(\mathcal{H} \leq H \wedge Comp(H, (caller, \mathsf{this}), \bar{y}))$$

7. A System for Creol Verification

The verification system for Creol is based on KeY[12]. Written in Java and published under the GNU general public license, it is available from the projects website[8]. The current version is a prototype which provides the functionalities presented in this article. In the following paragraphs of this section we briefly describe selected aspects of the system, namely its graphical user interface, its architecture, the implementation of the calculus, and the proof strategy.

[7] To be precise we would have to require, that i is new with respect to the history, which could be done by a predicate ranging over the history.

[8] www.key-project.org

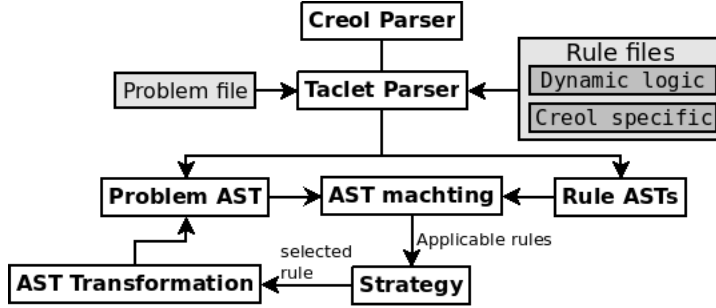

Figure 6: Architecture

```
impRight {\find(==> phi -> psi)
          \replacewith(==> psi)
          \add(phi ==>) }
```

```
compMon {\find(Comp(H1,L) ==>)
         \assumes(Prefix(H1,H2) ==>)
         \add(Comp(H2,L) ==>) }
```

Figure 7: Rules in the taclet language

In the graphical user interface the proof tree and open proof goals are displayed. Other features are pretty-printing and syntax-highlighting of the subformula/subterm currently pointed at with the mouse pointer. This enables a context sensitive menu offering only the rules applicable to the highlighted subformula/subterm. Apart from the rule name, tool-tips describe the effect of a rule. Besides interactive application of rules, automatic strategies can be configured. A more detailed description of the KeY interface is available in [3].

We describe the architecture of the prototype by means of Fig. 6. The problem file contains Creol code and its specification. Together with the specification specific rules a problem file is about 600 lines long. In a first step the file is handed to a parser which passes the code residing in the modalities to the Creol parser. Both parsers use the ANTLR [55] parser generator at which only the Creol parser was created from scratch taking about 3900 lines of code. The output of the parsers is an abstract syntax tree (AST) of logical formulae containing an program AST at each modality. For reading the rules of the calculus into memory the same parsers are used where the Creol specific rules are written in 1400 lines. The applicable rules are determined by a tree matching procedure providing the input for the strategy. Each rule is equipped with an heuristic tag which is used by the strategy together with information about the context of application (e.g. the term to be replaced resides on the right side of the sequence arrow) to rank the applicable rules. Finally the chosen rule is applied to the current sequence by transforming the AST as described by the rule which brings us back to the situation where applicable rules should be identified. Overall, the adaptions in the KeY-system took about 5000 lines.

Problem files, files containing logical rules, or axiomatizations of data types are written in the *taclet* language [57]. In Fig. 7 the rule `impRight` from Fig. 4 and the equation Eq. (16) are defined in the taclet language. A `find` describes the formula the rule is applicable to, `replacewith` specifies the replacement for the `find` formula, `assumes` characterises further assumptions not subject to replacements, and `add` causes its argument to be added. The arrow `==>` indicates on which side of the sequent the formulas are found, replaced or added. Writing a semicolon between two occurrences of `replacewith` or `add` causes a branching. Taclets omitting the sequence arrow `==>` are rewriting rules applicable in all contexts (the equivalent to the notation in this paper).

The theory explained in the previous section needed some small extensions to be run in the system. First, the some quantifier was not implemented, but is expressed by another formula. For example, the update formula like $\{\mathcal{H} := \mathsf{some}\ H.(Wf(H) \wedge \mathcal{H} \leq H)\}\phi$ is rewritten to:

$$\forall H_0.(\mathcal{H} \doteq H_0 \rightarrow \forall H_1.\{\mathcal{H} := H_1\}((Wf(H_1) \wedge H_0 \leq H_1) \rightarrow \phi)) \tag{17}$$

The old value of $\mathcal{H}$ is saved in H_0, and the new variable H_1 is assigned to $\mathcal{H}$. The implication assures that

H_1 has the desired properties when evaluating ϕ.

Finally, there are different prefix predicates $\leq_I$ where I is an interface. Thereby the interface invariant for I' is monotonous on $\leq_I$ if $I' \neq I$. The rules invoc, comp, and return use $\leq_I$ where I is the interface the message the rule adds corresponds to. Release points and the loop invariant use a prefix predicate $\leq_{all}$ which is not monotonous for interface specifications.

To achieve the remarkable high degree of automation a specialised proof heuristic for the Creol specific layout of the sequents was coined. Typically the challenging parts of a correctness proof of a Creol-method is to verify that class invariant holds after the symbolic execution of the method or the interface invariant holds when asynchronous communication is performed. This instantiation of an invariant has to be related with the assumption of the invariant at the last release point. Therefore the backwards-monotonicity of the prefix predicate is applied to the invariant being the proof obligation until the last release point is reached. While symbolically executing Creol code, a great number of equalities is induced by the implementation of the non-deterministic updates (see equation (17)). To avoid the existence of formulae expressing properties of the same history but being stated by different variables the equalities are applied eagerly. Not surprisingly, by defining a normal form for the terms and formulae expressing lists and prefix relationships among them the ratio of automated steps was significantly enhanced. Up to the decision whether a loop is unrolled or a (given) invariant is applied, it should be possible in theory to design a fully-automated strategy for the verification of single methods. During the development of the proof heuristics it was highly beneficial to draw the proof tree and mark fully-atomised leaves such that in the next attempt another issue could be tackled. Further research in computer-aided in visualising proof trees following the lines of [19, 11] could greatly simplify this process.

8. Verification Examples

8.1. Unbounded buffer

We give an implementation for an unbounded first-in-first-out (FIFO) buffer. This example is adapted from [26]. The interface contains two methods put and get which can be used to put into and to obtain an element from the buffer.

```
interface FifoBuffer
begin with Any
        op put(in x:Any; out)
        op get(in; out x:Any)
end
```

The interface invariant expresses that the sequence of elements retrieved from the buffer are a prefix of the elements put into the buffer. This ensures the FIFO property. Additionally, all elements must not equal null. We define $IInv(\mathsf{FifoBuffer})(\mathcal{H}/\{caller, callee\})$ (we write $\mathcal{H}$ instead of $\mathcal{H}/\{caller, callee\}$) as:

$$out(\mathcal{H}, callee) \leq in(\mathcal{H}, callee) \wedge \forall x.(x \in in(\mathcal{H}, callee) \rightarrow \neg x \doteq \mathsf{null})$$

where in, out are defined as:

$$\begin{array}{lcllcl} in(\epsilon, o) & = & \epsilon & out(\epsilon, o) & = & \epsilon \\ in(h \cdot o_2 \leftarrow o.put(x;), o) & = & in(h, o) \cdot x & out(h \cdot o_2 \leftarrow o.get(;x), o) & = & out(h, o) \cdot x \\ in(h \cdot msg, o) & = & in(h, o) & out(h \cdot msg, o) & = & out(h, o) \end{array}$$

Note that we do not guarantee that a caller gets the same objects it has put into the buffer. Such a buffer can be used for fair work balancing where a request is put into the buffer and workers take them out again.

The implementation of the buffer, given in Fig. 8, uses a chain of objects where each of them can store one element. The attribute cell is null if the object does not store an element. In next the reference to the following chain of objects is stored. Requests are forwarded to it if the object cannot serve them alone. The variable cnt holds the number of elements stored in cell and all following objects. Calls of get on an empty buffer are suspended until there are elements in the buffer.

```
class BufferImpl implements FifoBuffer
var cell:Any; var cnt:Int; var next:FifoBuffer;
begin with Any
        op put(in x:Any; out) ==
           if cnt=0 then cell:=x
                    else if next=null then next:=new Buffer end;
                         var l:Label[]; l!next.put(x); l?()
           end;
           cnt:= cnt+1; return()
        op get(in ; out x:Any) ==
           await (cnt>0);
           if cell=null then var l:Label[Any]; l!next.get(); l?(x)
                    else x:=cell; cell:=null
           end;
           cnt:=cnt−1; return(x)
end
```

Figure 8: The class implementing the buffer

For the class invariant we define another term $buf(o_1, o_2, h)$ which for an object o_1 and its next object o_2 reconstructs from the history h the elements in cell and all following objects.

$$buf(o_1, o_2, h) = \begin{cases} \epsilon & \text{if } h \doteq \epsilon \vee o_1 \doteq \mathsf{null} \vee o_2 \doteq \mathsf{null} \\ buf(o_1, o_2, h') \cdot x & \text{if } h \doteq h' \cdot o_1 \leftarrow o_2.put(x;) \\ rest(buf(o_1, o_2, h')) & \text{if } h \doteq h' \cdot o_1 \leftarrow o_2.get(;x) \\ buf(o_1, o_2, h') & \text{otherwise } h \doteq h' \cdot msg \end{cases}$$

rest removes the first element of a sequence.

Let us proceed with the class invariant. The attribute cnt equals the number of elements in cell and all following buffer cells. The interface invariant of FifoBuffer has to hold for both the interface called and implemented by the class. Additionally, we state that the sequence of values put into the current cell equals the sequence of values obtained from the buffer with the cell and the content of the following buffer appended. (Again, we write $\mathcal{H}$ instead of $\mathcal{H}/\{caller, callee\}$.)

$$\begin{gathered} |\mathsf{cell} \cdot buf(\mathcal{H}, \mathsf{this}, \mathsf{next})| \doteq \mathsf{cnt} \\ \wedge(\neg \mathsf{next} \doteq \mathsf{null} \rightarrow IInv(\mathsf{FifoBuffer})(\mathcal{H}, \mathsf{next})) \wedge IInv(\mathsf{FifoBuffer})(\mathcal{H}, \mathsf{this}) \\ \wedge in(\mathcal{H}, \mathsf{this}) \doteq out(\mathcal{H}, \mathsf{this}) \cdot \mathsf{cell} \cdot buf(\mathcal{H}, \mathsf{this}, \mathsf{next}) \end{gathered}$$

If cell is null it is omitted. The example with the given specifications was proved interactively by the system. The method put was verified in 2846 steps and 85 branches, whereas get needed 2614 steps and 66 branches. With respect to degree of automation these proofs were very promising, as the system achieved 99,1% of automated steps over the total number of steps for the proof of put and 98,4% when proofing get. The proofs were performed by a system implementing an older version of the calculus described in the previous version of this paper [5]. Due to the fact, that the calculus was simplified since then, the authors claim, that the degree of automation can easily be transferred to the new setting.

8.2. Automated teller machine

The example of the automated teller machine distributed throughout the paper was successfully verified by usage of 45 branches and 7480 steps in total where 98,4% of them were automatic. As the implementation of the class makes heavy use of asynchronous method calls and (co)interfaces, it has been shown that our system can easily deal with them. The experiences with specifications in form of regular expressions were

promising. They are easy to write down and an automated strategy can deal with them as the number of successor states is usually limited which narrows the search space of the proof. A further step in generalising the system could be the introduction of a logical toolbox expressing sets, relations and other well-understood mathematical notions simplifying the process of specifying and verifying other case studies.

9. Discussion and Conclusion

Creol's notion of inter-object communication is inspired by notions from process algebras (CSP [36], CCS [46], π-calculus [47]), which however model synchronous communication mostly. Moreover, Creol differs from those in integrating the notion of processes in the object-oriented setting, using named objects and methods rather than named channels. This also introduces more structure to the message passing (calls, replies, caller references, cointerfaces). The message passing paradigm on the inter-object level is combined with the shared memory paradigm on the local inter-thread level.

An early approach to the verification of shared-variable concurrency is 'interference freedom tests' [54]. A corresponding method targeting synchronous message passing is 'cooperation tests' [6]. Both the above are based on proof outlines of the composed processes, and therefore *non-compositional*. The first *compositional* proof methods were proposed by Cliff Jones for shared-variable concurrency, called 'rely-guarantee' [44], and by Jay Misra and Mani Chandy for synchronous message passing, called 'assumption-commitment' [48]. In both cases, 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."[23]. Our definition of the validity of formulas $[S]\varphi$, along with our calculus, combines assumption-commitment style reasoning, adapted to *asynchronous* method calls, with rely-guarantee style reasoning, adapted to object local *cooperative* parallelism. The notational style we used for semantic definitions is inspired by Hooman, de Roever et.al. [37].

Extending on the above principles of compositional verification, object invariants are used as a combined assumption/commitment or rely/guarantee conditions, respectively, both in the sequential setting to achieve modularity [9, 10], and in the concurrent setting [38]. Compared to the last mentioned works, Creol is more restrictive in that it forces shared memory to be entirely object internal. All knowledge of remote data is contained in fully abstract interface specifications talking about the communication history. Communication histories appeared originally both in the CSP as well as the object-oriented setting [18, 36]. A sound and complete compositional proof system based on history invariants and history projections was presented by Job Zwiers [60]. For other usages of communication histories in specification and verification, see for instance [59, 24].

KeY is among the state-of-the-art approaches to the verification of (at first) sequential object-oriented programs, together with systems like Boogie [8], ESC/Java2 [32], and Krakatoa [31]. In comparison to those, KeY is unique in that it does not merely generate verification conditions for an external off-the-shelf prover, but employs a calculus where symbolic execution of programs is interleaved with first-order theorem proving strategies. This goes together with the nature of first-order DL, which syntactically interleaves modalities and first-order operators. The cornerstone for KeY style symbolic execution, the updates, have similarities to generalised substitutions in formalisms such as the B method [2]. Updates are, however, tailored to symbolic execution rather than modeling (for instance, conflicts are resolved via right-win). The KeY tool uses these updates not only for verification, but also for test case generation with high code based coverage [29] and for symbolic debugging. The role of updates is largely orthogonal to the target language, allowing us to fully reuse this machinery for Creol.

As for Creol's thread concurrency model, this differs from many other languages in that it is *cooperative*, meaning the programmer actively releases control (conditionally). This simplifies reasoning considerably as compared to reasoning about *preemptive* concurrency, where atomicity has to be enforced by dedicated constructs. There is work on verifying a limited fragment of concurrent Java with KeY [13]. Here, the main idea is to prove the correctness of all permutations of schedulings at once. In [1], concurrent correctness of Java threads is addressed by combining sequential correctness with interference freedom tests and cooperation tests.

Very related to our work is the extension of the Boogie methodology to concurrent programs [38], targeting concurrent Spec#. From the beginning, this work is deeply integrated into an elaborate formal development environment, with all the features mentioned the first paragraph of this paper. The methodology requires users to annotate code with commands in between which an object is allowed to violate its invariant. This is combined with ownership of objects by threads. Just as in our system, invariants have to be established at specific points, and can be assumed at others. Also similar is the erasing of knowledge, there with the havoc statement, here with the *some* operator. Differences (apart from the asynchronous method calls) are the purely cooperative nature of our threads, and that our shared memory is object local, which makes ownership trivial. Connected to this is the inherently fully abstract specification of remote object interfaces, employing histories. The Boogie approach can simulate histories as well (see Fig. 1 in [38]), but it lies in the responsibility of the user whether or not the simulated history reflects the real one.

The system presented in this paper is still a prototype. It supports Creol dynamic logic, but the front-end for loading code and generating proof obligations is yet unfinished. This however will not be a real challenge, given the KeY infrastructure. The impressively high degree of automation shows great potential for further applications of the system, which could be eased even more by providing better support for history based specifications, like a library of frequently used queries on histories, or the usage of specification patterns [17], extended and configurable proof support for history based reasoning, and improved presentation on the syntax level and in the user interface.

We consider Creol's approach to modular object-oriented modeling as a good basis for scaling 'sequential formal methods' to the concurrent distributed setting, in particular when targeting functional correctness. The key is a very strong separation of concerns, which however naturally follows ultimate object-oriented principles. KeY has proved to be a good conceptual and technical basis for such an undertaking, which we argue can lead to an efficient and user-friendly environment for the verification of distributed object applications.

Acknowledgments

The authors would like to thank Richard Bubel, Jasmin Christian Blanchette, Frank de Boer, Einar Broch Johnsen, Olaf Owe, and Martin Steffen for fruitful discussions on the subject and feedback on earlier versions of the paper. We thank Richard Bubel moreover for his guidance concerning implementation issues.

References

[1] E. Ábrahám, F. S. de Boer, W.-P. de Roever, and M. Steffen. An assertion-based proof system for multithreaded Java. *Theor. Comput. Sci.*, 331(2-3):251–290, 2005.

[2] J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge Universtity Press, 1996.

[3] W. Ahrendt. Using KeY. In Beckert et al. [12], pages 409–451.

[4] W. Ahrendt, F. S. de Boer, and I. Grabe. Abstract object creation in dynamic logic. In A. Cavalcanti and D. Dams, editors, *Proc. 16th International Symposium on Formal Methods (FM'09)*, LNCS. Springer, 2009. to appear.

[5] W. Ahrendt and M. Dylla. A verification system for distributed objects with asynchronous method calls. In K. Breitman and A. Cavalcanti, editors, *Formal Methods and Software Engineering, International Conference on Formal Engineering Methods, ICFEM'09*, volume 5885 of *LNCS*, pages 387–406. Springer, 2009.

[6] K. Apt, N. Francez, and W. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2:359–385, 1980.

[7] H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. *SIGPLAN Not.*, 12(8):55–59, 1977.

[8] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *4th International Symposium on Formal Methods for Components and Objects, Amsterdam, The Netherlands*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.

[9] M. Barnett, R. DeLine, M. Fändrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[10] M. Barnett and D. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *Mathematics of Program Construction, 7th International Conference, Stirling, Scotland*, volume 3125 of *LNCS*, pages 54–84. Springer, 2004.

[11] M. Baum. Proof Visualization. Studienarbeit, Department of Computer Science, University of Karlsruhe, 2006.

[12] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.

[13] B. Beckert and V. Klebanov. A dynamic logic for deductive verification of concurrent programs. In M. Hinchey and T. Margaria, editors, *Conference on Software Engineering and Formal Methods (SEFM)*. IEEE Press, 2007.

[14] B. Beckert, V. Klebanov, and S. Schlager. Dynamic logic. In Beckert et al. [12], pages 69–177.

[15] J. C. Blanchette. Verification of assertions in Creol programs. Master's thesis, University of Oslo, Oslo, Norway, May 2008.

[16] R. Bubel. The Schorr-Waite-Algorithm. In Beckert et al. [12], pages 569–587.

[17] R. Bubel and R. Hähnle. Pattern-driven formal specification. In Beckert et al. [12], pages 295–315.

[18] O.-J. Dahl. Can program proving be made practical? *Les Fondements de la Programmaion*, pages 57–114, Dec. 1977.

[19] M. A. D. Darab. Towards a GUI for program verification with key. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, Jan 2010.

[20] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Proceedings, 2nd International Conference on Security in Pervasive Computing*, volume 4454 of *LNCS*, pages 193–209. Springer, 2005.

[21] F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 316–330. Springer, Mar. 2007.

[22] F. S. de Boer, I. Grabe, M. M. Jaghoori, A. Stam, and W. Yi. Modeling and analysis of thread-pools in an industrial communication platform. In K. Breitman and A. Cavalcanti, editors, *ICFEM*, volume 5885 of *Lecture Notes in Computer Science*, pages 367–386. Springer, 2009.

[23] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, Nov. 2001.

[24] F. S. deBoer. A Hoare logic for dynamic networks of asychronously communicating deterministic processes. *Theor. Comput. Sci.*, 274(1-2):3–41, 2002.

[25] J. Dovland, E. B. Johnsen, and O. Owe. A Hoare logic for concurrent objects with asynchronous method calls. Technical Report 315, Department of Informatics, University of Oslo, 2006.

[26] J. Dovland, E. B. Johnsen, and O. Owe. A compositional proof system for dynamic object systems. Technical Report 351, Department of Informatics, University of Oslo, 2008.

[27] J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of dynamic systems: Component reasoning for concurrent objects. *Electron. Notes Theor. Comput. Sci.*, 203(3):19–34, 2008.

[28] M. Dylla. A verification system for the distributed object-oriented language Creol. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, June 2009.

[29] C. Engel and R. Hähnle. Generating unit tests from formal proofs. In Y. Gurevich and B. Meyer, editors, *Proceedings, 1st International Conference on Tests And Proofs (TAP), Zurich, Switzerland*, volume 4454 of *LNCS*. Springer, 2007.

[30] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Communications*, 10:55–66, 1988.

[31] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *Conference on Computer Aided Verification*, volume 4590 of *LNCS*. Springer, 2007.

[32] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Conference on Programming Language Design and Implementation, Berlin*, pages 234–245. ACM Press, 2002.

[33] M. Giese. First-order logic. In Beckert et al. [12], pages 21–68.

[34] D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.

[35] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 583, Oct. 1969.

[36] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[37] J. Hooman, W.-P. de Roever, P. Pandya, Q. Xu, P. Zhou, and H. Schepers. A compositional approach to concurrency and its applications. unfinished manuscript. Available online at `http://www.informatik.uni-kiel.de/inf/deRoever/books/`, Apr. 2003.

[38] B. Jacobs, K. R. M. Leino, F. Piessens, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *Conference on Software Engineering and Formal Methods*, pages 137–147. IEEE Computer Society, 2005.

[39] E. B. Johnsen and O. Owe. A compositional formalism for object viewpoints. In B. Jacobs and A. Rensink, editors, *Proceedings of the 5th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002)*, pages 45–60. Kluwer Academic Publishers, Mar. 2002.

[40] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. In *Proc. 2nd Intl. Conf. on Software Engineering and Formal Methods (SEFM'04)*, pages 188–197. IEEE Computer Society Press, Sept. 2004.

[41] E. B. Johnsen and O. Owe. Object-oriented specification and open distributed systems. In O. Owe, S. Krogdahl, and T. Lyche, editors, *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, volume 2635 of *LNCS*, pages 137–164. Springer, 2004.

[42] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

[43] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2), 2006.

[44] C. B. Jones. *Development Methods for Computer Programs Including a Notion of Interference*. PhD thesis, Oxford University, UK, 1981.

[45] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Conference on Programming Language design and Implementation*, pages 260–267, New York, NY, USA, 1988. ACM.

[46] R. Milner. *A Calculus for Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.

[47] R. Milner. *Communicating and Mobile Systems: the Pi Calculus*. Cambridge Universtity Press, 1999.

[48] J. Misra and K. Chandy. Proofs of networkes and processes. *IEEE Transactions on Software Engineering*, 7(7):417–426, 1981.

[49] W. Mostowski. Formalisation and verification of Java Card security properties in dynamic logic. In M. Cerioli, editor, *Proceedings, Fundamental Approaches to Software Engineering (FASE), Edinburgh*, volume 3442 of *LNCS*, pages 357–371. Springer, 2005.

[50] W. Mostowski. The Demoney case study. In Beckert et al. [12], pages 533–568.

[51] W. Mostowski. Fully verified Java Card API reference implementation. In B. Beckert, editor, *Verify'07*, volume 259 of *CEUR WS*, July 2007.

[52] O. Mürk, D. Larsson, and R. Hähnle. KeY-C: A tool for verification of C programs. In F. Pfenning, editor, *Proc. 21st Conference on Automated Deduction (CADE), Bremen, Germany*, volume 4603 of *LNCS*, pages 385–390. Springer-Verlag, 2007.

[53] S. Nanchen, H. Schmid, P. H. Schmitt, and R. Stärk. The ASMKeY theorem prover. Technical Report 436, ETH Zürich, 2004.

[54] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[55] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages.* The Pragmatic Bookshelf, Raleigh, 2007.

[56] A. Platzer and J.-D. Quesel. KeYmaera: A hybrid theorem prover for hybrid systems. In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR*, volume 5195 of *LNCS*, pages 171–178. Springer, 2008.

[57] P. Rümmer. Construction of proofs. In Beckert et al. [12], pages 179–242.

[58] P. H. Schmitt and I. Tonin. Verifying the Mondex case study. In M. Hinchey and T. Margaria, editors, *Proc. 5.IEEE Int.Conf. on Software Engineeging and Formal Methods (SEFM)*, pages 47–56. IEEE Press, 2007.

[59] N. Soundararajan. Axiomatic semantics of communicating sequential processes. *ACM Trans. Program. Lang. Syst.*, 6(4):647–662, 1984.

[60] J. Zwiers. *Compositionality, Concurrency and Partial Correctness*, volume 321 of *LNCS*. Springer-Verlag, 1989.

Appendix A. Syntax

This part of the appendix contains the grammars describing the syntax of Creol code. We start with a grammar for interfaces and classes where the statements will be given by another grammar. Braces are used as part of the grammar and do not occur in programs. By * we denote the Klenee star, $^+$ stands for multiple occurrences but at least once and $^?$ is used for optional occurrence.

$$
\begin{array}{rll}
f ::= & \{c \mid i\}^* & \text{file} \\
c ::= & \mathsf{class}\ C\ \mathsf{begin} & \text{class} \\
 & \{\mathsf{var}\ x : T;\}^* & \text{class attributes} \\
 & \{\mathsf{with}\ I\ \{\mathsf{op}\ mtd(\mathsf{in}\ \{d\}^?;\ \mathsf{out}\ \{d\}^?) == s;\ \mathsf{return}(e^*)\}^+\}^* & \text{methods} \\
 & \mathsf{end} & \\
i ::= & \mathsf{interface}\ I\ \mathsf{begin} & \text{interface} \\
 & \{\mathsf{with}\ I\ \{\mathsf{op}\ mtd(\mathsf{in}\ \{d\}^?;\ \mathsf{out}\ \{d\}^?)\}^+\}^* & \text{methods} \\
 & \mathsf{end} & \\
d ::= & x : T\ \{,\ x : T\}^+ & \text{parameters} \\
T ::= & \mathsf{Int} \mid \mathsf{Bool} \mid I & \text{types}
\end{array}
$$

A source code file contains a number of classes and interfaces. An interface is similar to a class but contains only method declarations. A class can contain class attributes indicated by variable declarations **var** $x : T$ as in the previous grammar. Those are followed by the definition of a cointerface I indicated by **with** I. For a cointerface several methods can be defined by **op** $mtd($**in** $\{d\}^?$; **out** $\{d\}^?) == s$ where mtd is the identifier, d are optional lists of declarations of input and output parameters, and s are statements of the next grammar. The next to last line of the grammar describes the syntax of a list of method parameter declarations with variable identifiers x which are typed by **Int**, **Bool**, or an interface I. We continue with the grammar defining statements.

$$
\begin{array}{rll}
s ::= & \mathsf{skip} \mid s;s \mid \mathsf{var}\ x : D \mid x := e & \\
 & \mid x := \mathsf{new}\ C \mid \mathsf{if}\ b\ \mathsf{then}\ s\ \{\mathsf{else}\ s\}^?\ \mathsf{end} \mid \mathsf{while}\ b\ \mathsf{do}\ s\ \mathsf{end} & \text{statements} \\
 & \mid l!x.mtd(\{e\ \{,\ e\}^*\}^?) \mid l?(\{x\ \{,\ x\}^*\}^?) \mid \mathsf{release} \mid \mathsf{await}\ b \mid \mathsf{await}\ l? \mid \mathsf{block} & \\
e ::= & b \mid i \mid e = e \mid \mathsf{null} \mid \mathsf{this} \mid \mathsf{me} \mid \mathsf{caller} & \text{expressions} \\
b ::= & x \mid \mathsf{true} \mid \mathsf{false} \mid b \wedge b \mid b \vee b \mid \neg b \mid i < i \mid i > i \mid e = e & \text{boolean} \\
i ::= & x \mid i + i \mid i - i \mid i * i \mid i/i \mid \ \ldots \ \mid -1 \mid 0 \mid 1 \mid 2 \mid \ \ldots & \text{integer} \\
D ::= & T \mid \mathsf{Label}[\{\ T\ \{,\ T\}^*\ \}^?] & \text{types} \\
T ::= & \mathsf{Bool} \mid \mathsf{Int} \mid I &
\end{array}
$$

Besides the **skip** statement, a statement can be the sequential concatenation of two statements expressed by a semicolon. Variables are declared by use of the keyword **var** followed by the a variable identifier x a colon and a type. The assignment uses := as the single equal sign is reserved for equality. A special assignment is the object creation where the expression e of the usual assignment is replaced by **new** C with C being a class identifier. At the well-known **if** construct the **else** branch is optional. The **while** loop is conditioned by the Boolean expression b and contains another statement s in its body. A method invocation uses l as a label and furthermore contains a variable x, a method mtd with a possibly empty list of parameters. The method completion makes use of the label l and has a possible empty list of variables as parameters. The statement **await** can be followed either by a Boolean expression b or a label l and a question mark. Expressions can either be an integer expression, a Boolean expression, equality of two expressions or the special keywords null, this, me, and caller. Boolean expressions b and integer expressions i follow the principles of most programming languages. Types are either **Bool**, **Int**, an interface I or a **Label** which contains a list of types not containing **Label** in the brackets.

Appendix B. Semantics

We give the semantics of the sequential statements not presented in the main part.

$$\mathcal{M}(\textbf{var } \mathsf{i} : \textbf{Int})(\sigma) = \{(\sigma', \langle\rangle) \mid \exists v.\sigma' = (\sigma : i \to v)\}$$

Note that i is a local variable, so it holds $\mathcal{E}(i)\sigma'|_l = v$.

$$\mathcal{M}(\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end})(\sigma) = \begin{array}{l}\{(\sigma_1, \theta) \mid \mathcal{B}(b)\sigma, (\sigma_1, \theta) \in \mathcal{M}(S_1)(\sigma)\} \\ \cup\{(\sigma_1, \theta) \mid \mathcal{B}(\neg b)\sigma, (\sigma_1, \theta) \in \mathcal{M}(S_2)(\sigma)\}\end{array}$$

Appendix C. Calculus

Most of the rules in this section were explained in the main sections of this article and therefore we do not give further details on them here.

$$\mathsf{skip}\ \frac{\langle\!|\omega|\!\rangle\phi}{\langle\!|\textbf{skip};\ \omega|\!\rangle\phi} \qquad \mathsf{assign}\ \frac{\{x := te\}\langle\!|\omega|\!\rangle\phi}{\langle\!|x := te;\ \omega|\!\rangle\phi}$$

There are four different variable declaration and each of them comes with its own implicit initialisation which is 0 for integers, false for booleans, and null for object references and labels.

$$\mathsf{intDecl}\ \frac{\{i := 0\}\langle\!|\omega|\!\rangle\phi}{\langle\!|\textbf{var } i : \textbf{Int};\ \omega|\!\rangle\phi} \qquad \mathsf{boolDecl}\ \frac{\{b := false\}\langle\!|\omega|\!\rangle\phi}{\langle\!|\textbf{var } b : \textbf{Bool};\ \omega|\!\rangle\phi}$$

$$\mathsf{objDecl}\ \frac{\{o := \mathsf{null}\}\langle\!|\omega|\!\rangle\phi}{\langle\!|\textbf{var } o : I;\ \omega|\!\rangle\phi} \qquad \mathsf{labelDecl}\ \frac{\{l := \mathsf{null}\}\langle\!|\omega|\!\rangle\phi}{\langle\!|\textbf{var } l : \textbf{Label}[T];\ \omega|\!\rangle\phi}$$

A division by zero could occur arbitrary deep in an expression. Thus those are disassembled leading to four rules per boolean and integer operator which follow the scheme below. They introduce new variables x' and x'' which save the arguments of the expressions. By te a terminal expression is denoted meaning that te cannot be further taken apart. Instances of terminal expressions are variables and constants. In contrast e stands for expressions which contain an operator at their top level.

$$\frac{\langle\!|x' := e_2;\ x := te_1 + x';\ \omega|\!\rangle\phi}{\langle\!|x := te_1 + e_2;\ \omega|\!\rangle\phi} \qquad \frac{\langle\!|x' := e_1;\ x := x' + te_2;\ \omega|\!\rangle\phi}{\langle\!|x := e_1 + te_2;\ \omega|\!\rangle\phi}$$

$$\frac{\langle\!|x' := e_1;\ x'' := e_2\ ;\ x := x' + x'';\ \omega|\!\rangle\phi}{\langle\!|x := e_1 + e_2;\ \omega|\!\rangle\phi} \qquad \frac{\{x := te_1 + te_2\}\langle\!|\omega|\!\rangle\phi}{\langle\!|x := te_1 + te_2;\ \omega|\!\rangle\phi}$$

Only one rule for the division differs which checks for divisions by zero.

$$\mathsf{DivTerminal}\ \frac{(\neg te_2 \doteq 0 \to \{x := te_1/te_2\}\langle\!|\omega|\!\rangle\phi) \land (te_2 \doteq 0 \to \langle\!|\textbf{block};\ \omega|\!\rangle\phi)}{\langle\!|x := te_1/te_2;\ \omega|\!\rangle\phi}$$

As a division by zero could be nested with in the expression being the condition of an **if** statement, there is a rule which equivalently rewrites the condition be introducing a new variable x.

$$\text{ifUnfold}\ \frac{\langle\!|x := b;\ \textbf{if}\ x\ \textbf{then}\ p\ \textbf{else}\ q\ \textbf{end};\ \omega|\!\rangle\phi}{\langle\!|\textbf{if}\ b\ \textbf{then}\ p\ \textbf{else}\ q\ \textbf{end};\ \omega|\!\rangle\phi}$$

A similar rule exists for the while loop, but it is omitted here. Once, the condition of the **if** statement has been analysed the rule below checks whether both branches have to be symbolically executed depending on the truth value of the terminal Boolean expression tb.

$$\text{if}\ \frac{(tb \doteq \mathsf{true} \rightarrow \langle\!|p;\ \omega|\!\rangle\phi) \wedge (tb \doteq \mathsf{false} \rightarrow \langle\!|q;\ \omega|\!\rangle\phi)}{\langle\!|\textbf{if}\ tb\ \textbf{then}\ p\ \textbf{else}\ q\ \textbf{end};\ \omega|\!\rangle\phi}$$

$$\text{blockBox}\ \frac{true}{[\textbf{block};\ \omega]\phi} \qquad \text{blockDia}\ \frac{false}{\langle\textbf{block};\ \omega\rangle\phi}$$

The rule for object creation involves the history, but does not involve any invariants.

$$\text{new}\ \frac{\Gamma \vdash \{o := (\mathsf{this}, (c, i))\}\{U_{\mathcal{H}}^{new}\}\langle\!|\omega|\!\rangle\phi, \Delta}{\Gamma \vdash \langle\!|o{:=}\ \textbf{new}\ c\ ;\ \omega|\!\rangle\phi, \Delta}$$

In the above rule i is a new constant symbol with respect to the history. To the variable o the reference being a pair of caller (the object subject to verification) and new object id is assigned. The update reads as follows $U_{\mathcal{H}}^{new} = \mathcal{H} := \mathsf{some}\ H, \mathcal{H} \leq H \wedge New(\mathcal{H}, o))$.