A system for compositional verification of asynchronous objects

Wolfgang Ahrendt, Maximilian Dylla

1. Introduction

The area of object-oriented program verification made significant progress during the last decade. Systems like Boogie [8], ESC/Java2 [31], KeY [12], and Krakatoa [30] 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 (C# in case of Boogie, Java in case of the other mentioned tools).

However, this development mostly concerns sequential, stand-alone applications. When it comes to verifying functional properties of concurrent and distributed applications, the situation is different. Even if there is a range of literature on the verification of ‘distributed formalisms’ (based for instance on process calculi [46,35,47]), 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 [21], we achieve a system for compositional verification of concurrent, distributed object-oriented applications. While still being a prototype system, 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 setting would, however, would 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 other’s 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 initial. 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]. A reader with no prior exposition to Creol may consider the tutorial [38].

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 [51], ASMs [52], and hybrid systems [55]. What all these have in common is 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 [50], the Mondex case study (the most substantial benchmark in the Grand Challenge repository) [57], the Schorr-Waite algorithm [16], and the electronic purse application Demoney [49]. 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.\footnote{See Section 9 for an exception.} 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.

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

Ultimately, the presented semantics and calculus should be connected, by (a) proving soundness of the calculus, (b) investigating the degree of completeness, and (c) precisely defining and proving compositionality in this setting. These issues are future work, see also Section 9.

The structure of the paper is as follows. Section 2 introduces Creol, and gives examples of its usage. Section 3 then presents a denotational semantics of Creol. In Section 4, we describe the logic and calculus characteristic for KeY, insofar as it is (largely) independent of the particular target language. Thereafter, Section 5 presents the semantics of the logic, followed by Section 6 containing the calculus for Creol specifically. Section 7 discusses system oriented aspects of KeY for Creol, including a small account on taclets. Section 8 then demonstrates the usage of the systems in examples. In Section 9, we discuss related work, and draw conclusions. Finally, Appendix A defines the syntax of Creol, followed by Appendix B and Appendix C serving as a reference for the semantics and the calculus, respectively.

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.)
Fig. 1. Communication of the automated teller machine, where the dotted arrow represents return messages only, whereas the other arrows stand for both initiated communication and replies.

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

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
          li!caller.dispense(am); l2!caller.returnCard(); l2?()
        else li!caller.returnCard(); l2?() end
      else li!caller.returnCard(); l2?() end; return()
end

Fig. 2. The interfaces of the automated teller machine.

Fig. 3. The class implementing the teller machine.

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 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 using the await statement, see below.) The effect of li?(x) is similar to treating x as a future variable [20,7] 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 $x := \text{new } C(e^*)$, 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 corresponding reply message has 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 Section 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 on 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 [22,36] were a great inspiration for this work. One basic principle, invented by Zwieterings [59], 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 $\sigma$. The state is a partial function pointing from the local variables of a thread and the object attributes to their current values. Sometimes we restrict the preimage of the state to the local variables or the object attributes by writing $\sigma|_1$ or $\sigma|_a$, respectively. The other important member of our semantics is the history $\theta$ being a sequence of messages. Our semantics is defined by a function on programs

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

where PROG is the set of programs, $\Sigma$ 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 of this very program, not any ‘initial’ history.

(But see sequential composition below, and the semantics of formulas relative to an initial history, Section 5). Programs that never terminate have the empty set as their semantics. As an example let us look at the block statement, which never terminates.

$$\mathcal{M}(\text{block})(\sigma) = \{\} \quad \mathcal{M}(\text{skip})(\sigma) = \{ (\sigma, ()) \}$$

The skip statement terminates, but causes no changes, so its set includes the tuple of the input state $\sigma$ and the empty history $()$. 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.

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

(By $(x : x \to v)$ we mean the modification, or extension, of $\sigma$ at $x$ with $v$.) Thus the set contains the state $\sigma'$ which equals the old state $\sigma$ up to the value for $x$. Here and later on, we write $e(\sigma)$ for the evaluation of expression $e$ with respect to the state $\sigma$. We chose $e$ to be a partial function, not always returning a value. In particular, if the evaluation of $e$ encounters a division by zero, $e$ does not return a value. As we quantify the value $v$ existentially, division by zero in $e$ makes the condition of the set always false, such that $\mathcal{M}(x := e)(\sigma)$ becomes the empty set. 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

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

---

2 This is equivalent to modelling $e$ as a relation which is functional.
where \( \sim \) denotes the concatenation of histories. A more involved form of sequential composition is the loop statement.

\[
\mathcal{M}(\text{while } b \text{ do } S \text{ end})(\sigma_0) = \left\{ (\sigma, \theta) \mid \exists k \in \mathbb{N}, (\sigma_1, \theta_1), \ldots, (\sigma_k, \theta_k) \text{ such that } \\
\sigma = \sigma_0, \theta = \theta_1 \sim \cdots \sim \theta_k, B(\neg b)\sigma_k, \\
\text{for } i = 0, \ldots, k - 1 : B(b)\sigma_i, (\sigma_{i+1}, \theta_{i+1}) \in \mathcal{M}(S)(\sigma_i) \right\}
\]

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 \( B(b)\sigma_k \). Finally, when the entry condition \( b \) is checked the \( k \)-th time, we have \( B(\neg b)\sigma_k \). The history of all those runs is concatenated as in the previous semantics for sequential execution. If the loop never terminates 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 \textit{release} statement allows another thread to run.

\[
\mathcal{M}(\text{release})(\sigma) = \{(\sigma', (\text{yield}(\sigma|_a)) \sim (\text{resume}(\sigma'|_a))) \mid \sigma'|_l = \sigma|_l \}
\]  

(1)

The compositional semantics stores a \textit{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 \( \sigma' \). Later, when we compose the histories of several threads, we will allow the switch of control from a \textit{yield} to the \textit{resume} of some thread such that both store the same attribute values, see Eqs. (7) and (8).

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

\[
\mathcal{M}(\text{await } b)(\sigma) = \{(\sigma', (\text{yield}(\sigma|_a)) \sim (\text{resume}(\sigma'|_a))) \mid \sigma'|_l = \sigma|_l, B(b)\sigma' \}
\]  

(2)

\textit{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 \( \langle \text{comp}(E(l)|_o, \bar{v}) \rangle \) 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 \( l \) has completed. (The return values \( \bar{v} \) remain unused here in contrast to Eq. (4).)

\[
\mathcal{M}(\text{await }?l)(\sigma) = \{(\sigma', (\text{yield}(\sigma|_a)) \sim (\text{resume}(\sigma'|_a)) \sim (\text{comp}(E(l)|_o, \bar{v})) ) \mid \sigma'|_l = \sigma|_l \}
\]  

(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 Eq. (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 \textit{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.

\[
\mathcal{M}(\text{!lo.m})(\sigma) = \left\{ (\sigma_1, \theta) \mid \exists \text{oid}. \exists v. \exists i. \text{oid} = \text{E}(o)|_k, v = \text{E}(\bar{x})|_a, \sigma_1 = (\sigma : l \rightarrow ((\text{E}(\text{this})|_a, \text{E}(\text{me})|_a), (\text{oid}, (m, i))))), \theta = (\text{invoc}(\text{E}(l)|_o, v)) \right\}
\]

By quantifying \text{oid} we ensure that the result of \( \text{E}(o)|_a \) is not \textit{null} if the set is non-empty. In the semantics we quantify existentially over the thread ID \( l \), which completes the identity \( \text{oid} = (m, i) \) of the callee. During the composition in Eq. (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 \textit{this} and \textit{me} are special variables where the former is a object 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 \( \text{E}(\bar{x})|_a \). The reply statement, which is the counterpart of the invocation, uses the same label to identify the corresponding completion message.

\[
\mathcal{M}(\text{?l.y})(\sigma) = \{(\sigma_1, \theta) \mid \exists \bar{v}. \exists l. \text{l} = \text{E}(l)|_a, \sigma_1 = (\sigma : \bar{y} \rightarrow \bar{v}), \theta = (\text{comp}(l|_v, \bar{v})) \}
\]  

(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 existential quantifier \( \exists l \). It is worthwhile emphasizing 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.

\[
\mathcal{M}(o := \text{new } C)(\sigma) = \left\{ (\sigma_1, \theta) \mid \exists i. \sigma_1 = (\sigma : o \rightarrow (C, i)), \theta = (\text{new}(\text{E}(\text{this})|_a, \text{E}(o)|_a)) \right\}
\]

(5)

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

---

In other versions of Creol, even the releasing is conditional.
Having dealt with single statements, we carry on with the semantics of a single thread as an instance of a method $m$.

$$
M(\text{op } m(\text{in } \bar{x}; \text{ out } \bar{y})) = \text{body}(\sigma) = \begin{cases} 
\exists \bar{\nu}. \exists \sigma_1. \exists \sigma_0. \\
\sigma_1 = (\sigma : \bar{x} \rightarrow \bar{\nu}), \\
(\sigma_2, \theta_1) \in M(\text{body}(\sigma_1)), \\
\theta_0 = \text{E(caller)}(\sigma), \\
\sigma_2 = (\text{E(this)}(\sigma), \text{E(me)}(\sigma)), \\
\theta_1 = (\text{resume}(\sigma_0)) \sim (\text{begin}((o, \sigma_2), \bar{\nu})), \\
\theta_2 = (\text{end}((o, \sigma_2), \text{E(\bar{y})}\sigma_2)) \sim (\text{yield}(\sigma_2)) 
\end{cases}
$$

(6)

Some values $\bar{\nu}$ serve as the input parameters updating the initial state $\sigma_0$, which in turn acts as an input for the semantics of the method body. The resulting history $\theta_0$ includes all communication initiated by the method, but not its own invocation and completion message contained in $\theta_1$ and $\theta_2$. To model the asynchrony we write $\text{begin}$ instead of $\text{invoc}$ and $\text{end}$ in place of $\text{comp}$. When composing the histories of the caller and callee in Eq. (12) we will require that the $\text{invoc}$ and $\text{comp}$ messages of the caller frame the $\text{begin}$ 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 $\theta_1$ starts with a $\text{resume}$ as in general another thread might run in advance, and $\theta_2$ ends with $\text{yield}$ to allow other threads to work.

Now we are ready to compose the semantics of a number $i$ of threads of the same method. We use another semantic function $M : \text{METHODS} \times \mathbb{N} \rightarrow \Sigma \rightarrow 2^H$ 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 $\sigma$ only consists of the value for this as defined in Eq. (10).

$$
M(m, i)(\sigma) = \begin{cases} 
\theta \downarrow \{ (m, j) | j \in \{1, \ldots, i\} \} = \theta, \\
\forall j \in \{1, \ldots, i\}, \exists t. \exists \bar{\nu}. \exists \sigma_t. \exists \sigma_i. \\
t = (m, j), \sigma_t = (\sigma : \text{me} \rightarrow t, \text{caller} \rightarrow o, \bar{\nu} \rightarrow \bar{\nu}), \\
(\theta_t, \sigma_i') \in M(\text{op } m(\text{in } \bar{x}; \text{ out } \bar{y}) = m\text{.body}(\sigma_i)), \\
\theta \downarrow \{ t \} = \theta_t, \text{cond}_m(\theta) 
\end{cases}
$$

(7)

With “$\forall j \in \{1, \ldots, i\}$. $\exists t.$”, we create $i$ different thread IDs $t$ and corresponding initial states $\sigma_t$ for each of them. Except for the thread ID saved in $\text{me}$ and the reference to the caller, the states differ in the object attributes $\bar{\nu}$, as the threads start executing after some other thread yielded, leaving the object attributes with any values. The next line of the conditions restricts the histories $\theta_t$ and states $\sigma_i'$ as a possible result of executing the method $m$ on the state $\sigma_i$. The merging $\theta$ of the histories $\theta_t$ is described as the inverse of projection to $t$, as in $[59]$. The projection on sets of threads $T$ removes the messages not involving any thread in $T$.

$$
\langle \rangle \downarrow T = \langle \rangle \quad \langle (m) \rangle \downarrow T = \begin{cases} 
\langle (m) \rangle \downarrow (\theta \downarrow T) \\
\theta \downarrow T 
\end{cases} 
$$

if $m.\text{caller}.\text{tid} \in T$ or $m.\text{callee}.\text{tid} \in T$

otherwise

So by writing $\theta \downarrow \{ t \} = \theta_t$ we require that all messages of $\theta_t$ are contained in $\theta$ in the correct order. To restrict $\theta$ to messages which are part of any history $\theta_t$, we write $\theta \downarrow \{ (m, j) | j \in \{1, \ldots, i\} \} = \theta$. In particular, $M(m, 0)(\sigma) = \{ \langle \rangle \}$, because $\theta \downarrow \{ \} = \langle \rangle$. The notion of $\text{cond}_m(\theta)$ abbreviates the following condition$^4$ making sure that a switch between two different histories $\theta_t$ in $\theta$ is only possible at the release points of the methods.

$$
\forall m_1, m_2, t_1, t_2, \theta_1, \theta_2. \{ (m_1) \} \subseteq \theta, m_1 \in \theta_1, m_2 \in \theta_2, t_1 \neq t_2 \Rightarrow [m_1 = \text{yield}(\cdot), m_2 = \text{resume}(\cdot)]
$$

(8)

We marked the release points in the Eqs. (1)-(3) by the $\text{yield}$-$\text{resume}$ pair. Now, the histories of the threads can be split up in between those messages, and only then, 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.

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

(9)

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 $o$.

$$
M(o) = \begin{cases} 
\exists \theta. \theta \downarrow \{ m | m \text{ is method of } C \} = \theta, \\
\forall \text{ methods } m \text{ of } C. \exists \theta_m. \exists \theta'. \exists \theta''. \\
\theta_m \in M(m(\text{this} \rightarrow o)), \theta \downarrow \{ m \} = \theta_m. \\
\theta'' = (\text{yield}(M(\text{const } C))) \sim \theta, \text{cond}_o(\theta'), \\
\theta'' = (\text{created(\theta', o)}) \sim \theta'' \downarrow T
\end{cases}
$$

(10)

The composition essentially works the same way as for threads of the same method (Eq. (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 $\text{this}$. The projections maintain the property that every message of all histories $\theta_m$ is included in $\theta$, but no more. The $\text{yield}$ message, preceding $\theta$, stores the result of the

$^4 \theta_1 \subseteq \theta_2$ if $\exists \theta'. \theta' \sim \theta_1 \sim \theta_2$. 

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

\[
(\langle \rangle \downarrow_{MP} \theta) \downarrow_{MP} = \begin{cases} 
\theta \downarrow_{MP} & \text{if } m = \text{yield}(.) \text{ or } m = \text{resume}(.) \\
\theta \downarrow_{MP} & \text{otherwise}
\end{cases}
\]

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

\[
\mathcal{M}(C, i) = \left\{ \theta | \begin{array}{c}
\theta \downarrow \{(C, j) | j \in \{1, \ldots, i\}\} = \theta, \\
\forall j \in \{1, \ldots, i\}. \exists o. \exists \theta_o. o = (C, j), \theta_o \in \mathcal{M}(o), \theta \downarrow \{o\} = \theta_o
\end{array} \right\}
\]

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 Eq. (8) for shared memory parallelism. (Note that \( \text{yield} \) and \( \text{resume} \) are not contained anymore in \( \mathcal{M}(o) \)). Therefore, the only condition on \( \theta \) is that it contains exactly all messages of the histories \( \theta_o \) 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.

\[
\text{get}_{\text{type}}((\langle \rangle \downarrow \theta) \downarrow_{MP} m) = \begin{cases} 
\text{get}_{\text{type}}(\theta) \cup \{m\} & \text{if } m \text{ has type } \text{type} \\
\text{get}_{\text{type}}(\theta) & \text{otherwise}
\end{cases}
\]

type will be a name of a message type, like \( \text{invoc} \) or \( \text{end} \). Another prerequisite is a Boolean function determining the order of two given messages within the history.

\[
m_1 \prec_o m_2 \equiv \exists \theta_1, \theta_2, \theta_3, \theta = \theta_1 \downarrow \theta_2 \downarrow \theta_3
\]

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

\[
\mathcal{M}(P) = \left\{ \theta | \begin{array}{c}
\exists \theta. \theta \downarrow \{C | C \text{ is class of } P\} = \theta, \\
\forall \text{ classes } C \text{ of } P. \exists \theta_C. \exists \bar{v}. \exists i. \\
\theta_C \in \mathcal{M}(C), \theta \downarrow \{C\} = \theta_C, \\
o.\text{class} = \text{Main}, \theta' = \langle \text{new}(., (\bar{v}) \rangle \downarrow \langle \text{invoc}(., (\text{main}, i), \bar{v}) \rangle \downarrow \theta \\
\forall \text{type}. \forall \theta_1, \forall \theta_2, \theta' = \theta_1 \downarrow \theta_2 \Rightarrow \text{get}_{\text{type}}(\theta_1) \cap \text{get}_{\text{type}}(\theta_2) = \{\} \\
\exists \text{exists bijection } f : \text{get}_{\text{new}}(\theta') \rightarrow \text{get}_{\text{created}}(\theta'), \text{cond}_f(f) \\
\exists \text{exists bijection } g : \text{get}_{\text{invoc}}(\theta') \rightarrow \text{get}_{\text{begin}}(\theta'), \text{cond}_g(g) \\
\exists \text{exists function } h : \text{get}_{\text{comp}}(\theta') \rightarrow \text{get}_{\text{end}}(\theta'), \text{cond}_h(h)
\end{array} \right\}
\]

The history \( \theta' \) is composed of the histories \( \theta_C \) 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 Eq. (11), the merging is described as the inverse of projection. We assume that every program has a class called \( \text{Main} \) which contains a method called \( \text{main} \). An object \( o \) is created initially as denoted by the \( \langle \text{new}(., o) \rangle \) message and its method \( \text{main} \) is invoked afterwards which is the meaning of the message \( \langle \text{invoc}(., (\text{main}, i), \bar{v}) \rangle \). As we are existentially quantifying over the parameters of messages there could be identical messages in the history, which is prohibited by the third line. Now, a valid history \( \theta' \) 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 \text{created}(o_1, o_2) \rangle of the history of the caller \( o_1 \) (see Eq. (10)) is always preceded by the creation message \langle \text{new}(o_1, o_2) \rangle of the callee \( o_2 \) (described in Eq. (5)). Formally, we express this fact, which we abbreviated as \( \text{cond}_f(f) \), as:

\[
\forall x. \forall y. f(x) = y \Rightarrow x.\text{caller} = y.\text{caller}, x.\text{callee} = y.\text{callee}, x \prec_o y
\]

The next function \( g \) does a similar job in relating a begin message to every invoc message. Analogous to Eq. (13) we ensure within \( \text{cond}_g(g) \) 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 \( \text{cond}_h(h) \). As it is not necessary to ask for the result of a method, \( h \) need 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 Eq. (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 induce a partial ordering on messages within histories representing asynchronous communication which makes this approach comparable to the classical work of [29].
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) [33], which, like Hoare logic [34], 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 typed first-order logic with two additional (mix-fix) operators: (..) . (diamond) and [. ] . (box). In both cases, the first argument is a program (fragment), whereas the second argument is another DL formula. A formula (p)φ is true in a state s if there is one terminating run of p, started in s, which results in a state where φ is true. As for the other operator, a formula [p]φ is true in a state s if all terminating runs of p, started in s, result in a state where φ is true. Note that (p)φ implies termination of at least one run, whereas [p]φ is trivially true if there is no terminating run. (DL is not able to express termination of all possible runs.) For deterministic program( fragment)s p, the difference between (p)φ and [p]φ is only termination.

DL is closed under all logical connectives. For instance, in a DL formula (p)φ, the postcondition φ may be any DL formula again, like in (p)ψ. Also, arbitrary connectives can enclose a box or diamond. For instance, the following formula states equivalence of p and q w.r.t. the “output”, the program variable x.

\[ \forall \nu. \left( (p)x := \nu \leftrightarrow (q)x := \nu \right) \] (14)

Our version of dynamic logic distinguishes strictly between logical variables and program variables. Logical variables (like 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 φ \rightarrow ⟨p⟩ψ, stating that the program p, when started from a state satisfying φ, has a terminating run with ψ being true afterwards. The formula φ \rightarrow [p]ψ, on the other hand, does not claim termination, and corresponds to the Hoare triple [φ]p [ψ].

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 program to each other (like the correctness of transformations) can be expressed elegantly, see Eq. (14).

What all major program logics (Hoare logic, wp calculus, DL) have in common that the resolution 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, x := y | y := 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 x < y \rightarrow [x := y | y := x] y < x. A typical usage of updates during proving is in formulas of the form \{U\}(p)φ, where 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 φ_1, . . . , φ_n \vdash ψ_1, . . . , ψ_n. A sequent is valid if the validity of all φ_1, . . . , φ_n implies validity of at least one of ψ_1, . . . , ψ_n. We use capital Greek letters to denote (possibly empty) sets of formulas. For instance, by Γ \vdash φ \rightarrow ψ. Δ 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:

\[
\frac{\Gamma_1 \vdash A_1 \ldots \Gamma_n \vdash A_n}{\Gamma \vdash \Delta}
\]

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 [32] for the full set). φ^c denotes a formula resulting from replacing the variables \overline{x} with expressions \overline{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, (skip; ω)φ can be rewritten to ⟨ω⟩φ regardless of where it occurs. For that we introduce the following syntax

\[
\frac{\phi'}{\phi}
\]
where \( \phi \) and \( \phi' \) are single formulas (i.e., there is no sequent arrow \( \vdash \), neither \( \Gamma' \) or \( \Delta \)). This denotes a rule where the (only) premise sequent is constructed by replacing \( \phi \) with \( \phi' \) anywhere in the conclusion sequent \( \phi \). In Fig. 5 we present some rules dealing with statements. (assign and if are simplified, see Section 6.1.) The schematic modality \( \langle \cdot \rangle \) can be instantiated with both \([ \cdot ]\) and \(\langle \cdot \rangle\), though consistently within a single rule application. Total correctness formulas of the form \(\langle \cdot \rangle \phi\) are proved by combining induction with \(\langle \cdot \rangle\).

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 \([\tau]s\phi\) or \([\tau]\langle \cdot \rangle\phi\), 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 method at a time. The various methods 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 these 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 \(\sigma\), i.e., an assignment of object attributes and local variables to values, a (semantic) history \(\theta\), and an assignment of logical variables \(\gamma\). 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 \(H\), which is interpreted by the given semantic history (typically \(\theta\)). Moreover, interface invariants have the logical variables caller\(^5\) and callee, typically as argument of the projection operator, see Section 6.2. Finally, in place of class invariants, the code of a method body is ever only correct relative to its 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 \(intf(m)\) returns the interface \(m\) belongs to.

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

\[
\begin{align*}
\text{commit}_{\text{inv},\gamma} = \{ \theta : \begin{cases} 
\text{if } \theta = \theta_0^- & \text{return invoc((oid, t), (oid', (m, i)), \bar{v}))} \\
\text{then } (\theta, \gamma) & \models IInv(intf(m))_{\text{oid}.oid'}_{\text{caller}.callee} 
\end{cases} \} \tag{15}
\end{align*}
\]

\[
\begin{align*}
\text{assume}_{\text{inv},\gamma} = \{ \theta : \begin{cases} 
\text{if } \theta = \theta_0^- & \text{comp((oid, t), (oid', (m, i)), \bar{v}))} \\
\text{then } (\theta, \gamma) & \models IInv(intf(m))_{\text{oid}.oid'}_{\text{caller}.callee} 
\end{cases} \} \tag{16}
\end{align*}
\]

\[
\begin{align*}
\text{guarantee}_{\text{inv},C,\gamma} = \{ \theta : \begin{cases} 
\text{if } \theta = \theta_0^- & \text{yield(\sigma))} \\
\text{then } (\sigma, \theta, \gamma) & \models CInv(C) 
\end{cases} \}
\end{align*}
\]

\[
\begin{align*}
\text{rely}_{\text{inv},C,\gamma} = \{ \theta : \begin{cases} 
\text{if } \theta = \theta_0^- & \text{resume(\sigma))} \\
\text{then } (\sigma, \theta, \gamma) & \models CInv(C) 
\end{cases} \}
\end{align*}
\]

With the help of these sets, we can define the semantics of (the base case of) dynamic logic formulas, relative to an initial state \(\sigma\), an initial history \(\theta\), and an assignment \(\gamma\) of logical variables to values:

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

\(^5\) The logical variable caller is related, but not identical to the implicit local variable in Section 3.
The same mechanism can be used for operator expressions, as long as all arguments are terminal releasing control (both before and during $S$), reply interfaces (both before and during $C$). Throughout the class invariant of $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 $\varphi$ 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. The reader may have noted the slight asymmetry in the above definition, between $\theta \preceq \theta_1$ (in assume $\cap$ rely) and $\theta_1$ only (in commit $\cap$ guarantee). The reason is that, in a compositional setting, each unit $S$ can assume (rely on) the contracts of other units in both its own trace $\theta_1$ and its pre-history $\theta$. But a unit $S$ can only ever commit to (guarantee) anything about its own trace $\theta_1$, not its pre-history $\theta$.

Note that $[S]_{\varphi}$ does not claim termination of any run, as the set $\mathcal{M}(S)(\sigma)$ of terminating runs is allowed to be empty. On the other hand, $(\sigma, \theta, \gamma, IInv, CInv, C) \models [S]_{\varphi}$ is defined by replacing in the above definition “for all $(\sigma_1, \theta_1) \in \mathcal{M}(S)(\sigma)$” with “there exists $(\sigma_1, \theta_1) \in \mathcal{M}(S)(\sigma)$”. Thereby, we claim, among other things, the existence of a terminating run (but not termination of all runs, cf. Section 4.1).

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

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

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 [27].

#### 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.

\[
\begin{align*}
\text{assign} & \quad \{x := te_1\} \{\omega\} \varphi & \text{intDecl} & \quad \{i := 0\} \{\omega\} \varphi \\
\{x := te_1; \omega\} \varphi & \quad \{\text{var } i : \text{int}; \omega\} \varphi
\end{align*}
\]

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 if the divisor is not zero. Otherwise, execution blocks. This semantics is captured by the following rule.

\[
\text{DivTerminal} \quad \{\neg te_2 \equiv 0 \rightarrow \{x := te_1/te_2\} \{\omega\} \varphi \} \land \{te_2 \equiv 0 \rightarrow \{\text{block}; \omega\} \varphi\}
\]

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).

\[
\begin{align*}
\{x' := e_1; x'' := e_2; x := x' + x''; \omega\} \varphi & \quad \{x := te_1 + te_2\} \{\omega\} \varphi \\
\{x := e_1 + e_2; \omega\} \varphi & \quad \{x := te_1 + te_2; \omega\} \varphi
\end{align*}
\]
In the left rule $e$, 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 as a division by zero could be contained in the expression.

\[
\text{ifUnfold } \frac{\{x := b; \text{if } x \text{ then } p \text{ else } q \text{ end}; \omega\} \phi}{\{\text{if } b \text{ then } p \text{ else } q \text{ end}; \omega\} \phi}
\]

In the above rule $x$ is a new program variable. 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 $tb$.

\[
\text{if } \frac{(tb = \text{true} \rightarrow \{p; \omega\} \phi) \land (tb = \text{false} \rightarrow \{q; \omega\} \phi)}{\{\text{if } tb \text{ then } p \text{ else } q \text{ end}; \omega\} \phi}
\]

Note that application of this rule may lead to proof branching in subsequent steps. As for while, the unwind rule was presented in Section 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:

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

### 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 a 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 again when execution proceeds. As for the behaviour of other objects, it 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 [24,26].

Every interface is specified by an interface invariant $IInv(I(\mathcal{H}))$. (Strictly, $IInv(I)$ is a formula, and $\langle \mathcal{H} \rangle$ 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 $\mathcal{H}/\langle \text{caller, callee} \rangle$, with '/' being the syntactical representation of the semantical projection '↓'. 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:

\[
\mathcal{H}/\langle \text{caller, callee} \rangle/ \rightarrow_{\leq} (\rightarrow \text{giveCode}[\cdot \rightarrow \text{withdraw}[\cdot \rightarrow \text{dispense}][\cdot \rightarrow \text{returnCard}])^*
\]

where $/ \rightarrow$ projects on invocation messages, $\cdot$ is concatenation, $\rightarrow$ 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:

\[
\mathcal{H}/\langle \text{caller, callee} \rangle \leq (\leftarrow \text{authorize(cid, .)} \cdot (\leftarrow \text{authorize(false)} | (\leftarrow \text{authorize(true)} \rightarrow \text{debit(cid, .)} \cdot \leftarrow \text{debit(.)})^*)^*
\]

Communication partners are omitted. The dot '.' is used as a wildcard for a parameter. Parameters (including the card ID cid) and communication partners are fixed during one iteration of the Kleene star. The meaning of the invariant is that only after authorisation can the debit procedure be attempted.

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

\[
\mathcal{H}/\langle \text{caller, callee} \rangle \leq (\rightarrow \text{authorize(cid, .)} \cdot (\leftarrow \text{authorize(false)} | (\leftarrow \text{authorize(true)} \rightarrow \text{debit(cid, .)} \cdot \leftarrow \text{debit(.)})^*)^*
\]

\[
\mathcal{H}/\langle \text{caller, callee} \rangle \leq (\rightarrow \text{authorize(cid, .)} \cdot (\leftarrow \text{authorize(false)} | (\leftarrow \text{authorize(true)} \rightarrow \text{debit(cid, .)} \cdot \leftarrow \text{debit(.)})^*)^*
\]

$F(\mathcal{H}, \overline{W})$ relates the state of the ordinary object attributes $\overline{W}$ 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 $C$ are put in a conjunction to ensure that all methods respect them. Now we can formulate the proof obligation for a method. The precondition 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 the completion message of the method. For each method

\[
\mathcal{H}/\langle \text{caller, callee} \rangle \leq (\rightarrow \text{authorize(cid, .)} \cdot (\leftarrow \text{authorize(false)} | (\leftarrow \text{authorize(true)} \rightarrow \text{debit(cid, .)} \cdot \leftarrow \text{debit(.)})^*)^*
\]

\[
\mathcal{H}/\langle \text{caller, callee} \rangle \leq (\rightarrow \text{authorize(cid, .)} \cdot (\leftarrow \text{authorize(false)} | (\leftarrow \text{authorize(true)} \rightarrow \text{debit(cid, .)} \cdot \leftarrow \text{debit(.)})^*)^*
\]
The simplest form of a release point is \( \text{release} \). As mentioned before the class invariant forms a contract between all threads of an object. So the rule for \( \text{release} \) forces us to show that the class invariant is established in the current state (being the commitment of \((15)\)), before releasing the processor. When this thread resumes, it can rely on the invariant for the remaining code \( \omega \) to be executed.

\[
\text{release} \quad \Gamma \vdash \text{Clnv}(\mathcal{H}, \overline{W}), \Delta \quad \Gamma \vdash \{U_{\mathcal{H}, \overline{W}}\}[\omega] \varphi, \Delta \\
\Gamma \vdash \text{release; } \omega[\varphi, \Delta]
\]

We omit the class as an argument of \( \text{Clnv} \) for brevity as it can be obtained from this. Here, \( U_{\mathcal{H}, \overline{W}} \) is the update \( \mathcal{H}, \overline{W} := \text{some } H, \overline{W}, (\text{Clnv}(H, \overline{W}) \land \mathcal{H} \leq H) \). This update represents an arbitrary but fixed system state, satisfying the class invariant (thus relying on it), in which execution continues. This is necessary because values of the object 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 \( \text{await } b \) 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 the evaluation of \( b \) does not block due to an error. The two assumptions together are expressed via \( (x := b)x \doteq true \).

\[
\text{awaitExp} \quad \Gamma \vdash \text{Clnv}(\mathcal{H}, \overline{W}), \Delta \quad \Gamma \vdash \{U_{\mathcal{H}, \overline{W}}\}(x := b)x \doteq true \rightarrow [\omega] \varphi, \Delta \\
\Gamma \vdash \text{while } b \text{ do } p \text{ end; } \omega[\varphi, \Delta]
\]

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

Partial correctness of a loop can also be shown with the help of a loop invariant \( \text{inv}_{\text{loop}}(\mathcal{H}, \overline{mod}) \), where \( \overline{mod} \) is the modifier set of the loop (all variables assigned in the loop). To be most general, all object 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.

\[
\Gamma \vdash \text{loopInv} \quad \Gamma \vdash \text{init.valid} \quad \Gamma \vdash \text{preserving} \quad \Gamma \vdash \text{use-case}
\]
The update $U^\text{loop}_{\mathcal{H}, \mathsf{mod}}$ is defined as:

$$\mathcal{H}, \mathsf{mod} := \text{some } H, \mathsf{m} (\mathcal{H} \leq H \land \text{inv}_{\text{loop}}(H, \mathsf{m}))$$

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

Analogous to $\text{Comp} (\mathcal{H}, l, y)$ there are predicates $\text{Invoc} (\mathcal{H}, l, \bar{x})$ and $\text{New} (\mathcal{H}, o)$ 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:

$$\text{Comp} (\mathcal{H}_0, l, \bar{y}) \land \mathcal{H}_0 \leq \mathcal{H}_1 \rightarrow \text{Comp} (\mathcal{H}_1, l, \bar{y})$$

Moreover $\text{Comp}$, $\text{New}$, as well as $\text{Invoc}$ are monotonic w.r.t. $\leq$. Additionally, the contraposition is used in our proof system.

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

$$\begin{array}{l}
\Gamma \vdash o \equiv \text{null} \rightarrow [\text{block}; \omega \phi], \Delta \\
\Gamma \vdash -o \equiv \text{null} \rightarrow [l := ((\text{this}, \text{me}), (o, i))][U^{\text{invoc}}_{\mathcal{H}, l, \bar{y}}](\text{Invoc}(l)(\mathcal{H}, \text{this}, \bar{x}) \land \{\omega \phi\}, \Delta)
\end{array}$$

Here, $i$ is a new constant symbol representing the thread ID. If $o$ is null, execution blocks. In the first branch, the invariant of the remote interface $l$ must be shown to fulfil the assumption (16) where $l$ is the type of $o$. The abbreviation $U^{\text{invoc}}_{\mathcal{H}, l, \bar{y}}$ for the update in its full form is:

$$\mathcal{H} := \text{some } H (\mathcal{H} \leq H \land \text{Invoc}(H, l, \bar{x}))$$

The new history contains the invocation message $\text{invoc}(l, \bar{x})$.

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

$$\begin{array}{l}
\Gamma \vdash l \equiv \text{null} \rightarrow [\text{block}; \omega \phi], \Delta \\
\Gamma \vdash -l \equiv \text{null} \rightarrow [U^{\text{comp}}_{\mathcal{H}, l, \bar{y}}](\omega \phi), \Delta \\
\Gamma \vdash [\text{comp}(l)(\omega \phi), \Delta)
\end{array}$$

The update $U^{\text{comp}}_{\mathcal{H}, l, \bar{y}}$ is analogous to $U^{\text{invoc}}_{\mathcal{H}, l, \bar{y}}$. It overwrites the return parameters $\bar{y}$, uses $\text{Comp}$ to denote that the completion messages occurs in the new history, and assumes the interface invariant.

$$\mathcal{H}, \bar{y} := \text{some } H, \bar{p} (\mathcal{H} \leq H \land \text{Invoc}(l)(\mathcal{H}, \text{this}, \bar{x}, \text{callee}) \land \text{Comp}(H, l, \bar{p}))$$

$I$ 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, $i$), where $i$ is an 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 $\text{return}$ 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 $\phi$ (see previous section).

$$\begin{array}{l}
\Gamma \vdash \{U^{\text{return}}_{\mathcal{H}, l, \bar{y}}\} \phi, \Delta \\
\Gamma \vdash \{\text{return}(l)\} \phi, \Delta
\end{array}$$

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

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

### 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. 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.

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, tooltips 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].

---

6 [www.key-project.org](http://www.key-project.org).
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 the problem files of the verified methods were each 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 [54] 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 a 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 a 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 adaptations 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 [56]. In Fig. 7 the rule \texttt{impRight} from Fig. 4 and the Eq. (18) with $\bar{y}$ consisting of only one variable $Y$ are defined in the taclet language. A \texttt{find} describes the formula the rule is applicable to, \texttt{replacewith} specifies the replacement for the \texttt{find} formula, \texttt{assumes} characterises further assumptions not subject to replacements, and \texttt{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 \texttt{replacewith} or \texttt{add} causes a branching. Taclets omitting the sequence arrow $=>$ are rewriting rules applicable in all contexts.

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

\[
\forall H_0. \ (\mathcal{H} \leq H_0 \rightarrow \forall H_1. \ [\mathcal{H} := H_1]. \ ((\texttt{Wf}(H_1) \land H_0 \leq H_1) \rightarrow \phi))
\]  

(19)

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 $\phi$.

Secondly, there are different prefix predicates $\leq_l$ where $l$ is an interface. Thereby the interface invariant for $l'$ is monotonic on $\leq_l$ if $l' \neq l$. The rules \texttt{invoc}, \texttt{comp}, and \texttt{return} use $\leq_l$ where $l$ is the interface the message added by the rule corresponds to. Release points and the loop invariant use a prefix predicate $\leq_{all}$ which is not monotonic 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 a class invariant holds after the symbolic execution of the method, or that 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 are induced by the implementation of the non-deterministic updates (see Eq. (19)). 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-aid proof tree visualisation following the lines of [19, 11] could greatly simplify this process.
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

Fig. 8. The class implementing the buffer.

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 [25]. 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 the interface invariant $IInv(FifoBuffer)(\mathcal{H} / \{\text{caller, callee}\})$ (we write $\mathcal{H}$ instead of $\mathcal{H} / \{\text{caller, callee}\}$) as:

$$
\text{out}(\mathcal{H}, \text{callee}) \leq \text{in}(\mathcal{H}, \text{callee}) \land \forall x. (x \in \text{in}(\mathcal{H}, \text{callee}) \rightarrow \neg x \neq \text{null})
$$

where $\text{in}$, $\text{out}$ are defined as:

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

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 $\text{cell}$ 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 $\text{cnt}$ holds the number of elements stored in $\text{cell}$ and all following objects. Calls of get on an empty buffer are suspended until there are elements in the buffer.

Let us proceed with the class invariant. The attribute $\text{cnt}$ equals the number of elements in $\text{cell}$ 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 $\mathcal{H} / \{\text{caller, callee}\}$.)

$$
|\text{cell} \cdot \text{buf}(\mathcal{H}, \text{this}, \text{next})| \equiv \text{cnt}
\land (\neg \text{next} \equiv \text{null} \rightarrow \text{IInv(FifoBuffer)}(\mathcal{H}, \text{next}) \land \text{IInv(FifoBuffer)}(\mathcal{H}, \text{this})
\land \text{int}(\mathcal{H}, \text{this}) \equiv \text{out}(\mathcal{H}, \text{this}) \land \text{cell} \cdot \text{buf}(\mathcal{H}, \text{this}, \text{next})
$$

If $\text{cell}$ is null it is omitted. The term $\text{buf}(o_1, o_2, h)$ in the above formula reconstructs for an object $o_1$ and its next object $o_2$ from the history $h$ the elements in $\text{cell}$ and all following objects.

$$
\text{buf}(o_1, o_2, h) = \begin{cases} 
\epsilon & \text{if } h \equiv \epsilon \lor o_1 \equiv \text{null} \lor o_2 \equiv \text{null} \\
\text{buf}(o_1, o_2, h') \cdot x & \text{if } h \equiv h' \lor o_1 \equiv o_2.\text{put}(x) \\
\text{rest}(\text{buf}(o_1, o_2, h')) & \text{if } h \equiv h' \lor o_1 \equiv o_2.\text{get}(\cdot) \\
\text{buf}(o_1, o_2, h') & \text{otherwise } h \equiv h' \cdot \text{msg}
\end{cases}
$$
rest removes the first element of a sequence. 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 proving 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 the 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 [35], 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, (co)interfaces). 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' [53]. 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." [22]. Our definition of the validity of formulas $\varphi$, 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. [36].

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 [37]. 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 [18,35]. A sound and complete compositional proof system based on history invariants and history projections was presented by Job Zwiers [59]. For other usages of communication histories in specification and verification, see for instance 58,23.

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 [31], and Krakatoa [30]. 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 last-win). The KeY tool uses these updates not only for verification, but also for test case generation with high code-based coverage [28] 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.

Related closely to our work is the extension of the Boogie methodology to concurrent programs [37], 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 same 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 [37]), 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.

Among the next issues on the agenda of this line of research are soundness, completeness, and compositionality. We believe that the calculus presented in Section 6 is ‘nearly’ sound wrt. the semantics from Section 3. We write ‘nearly’ because one should never claim soundness of a calculus for concurrency without having proven it. W.-P. de Roever reported that “every alleged proof method for concurrency, which reached his desk and […] had not been proven sound, turned out to be unsound” [22]. Therefore, a few fixes in details of the calculus or/and the semantics are to be expected. A related item of future work is the investigation of the degree of completeness. Finally, an important step to be taken is the precise formulation of the property of parallel compositionality, together with a proof of it. Compositionality can be formulated as a ‘parallel composition rule’ on the level of the calculus, or alternatively as a meta-level property. The latter could be more appropriate in a setting where the parallel composition (of dynamically created threads in dynamically created objects) is not explicit as a language construct, but rather implicit in the semantics.

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 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.

Acknowledgements

The authors would like to thank Richard Bubel, Jasmin Christian Blanchette, Frank de Boer, Einar Broch Johnsen, Olaf Owe, Martin Steffen, and Ilham Kurnia 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. Finally, we thank the anonymous reviewers for many insightful comments which led to many improvements in the final version of the paper.

The work of first author has partially been supported by the EU-project FP7-ICT-2007-3 HATS: Highly Adaptable and Trustworthy Software using Formal Methods. The work of second author 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.

Appendix A. Syntax

This part of the appendix contains the grammars describing the syntax of Creol code used throughout this article. In comparison to other publications about Creol, some features which we do not cover are excluded from the syntax definition.

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 Kleene star, + stands for multiple occurrences but at least once and ^ is used for optional occurrence.

\[
\begin{align*}
f & ::= \{ c | i \}^* & \text{file} \\
c & ::= \text{class } C \begin{align*} & \begin{cases} \text{implements } (l)^+ \end{cases} \end{align*} \begin{align*} & \begin{cases} \text{var } x : T; \end{cases} \end{align*} \begin{align*} & \begin{cases} \text{with } l \begin{cases} \text{op mtd(in } d; \text{ out } d)^+ \end{cases} \end{cases} \begin{cases} \text{return(e^*)} \end{cases} \end{align*} \text{ methods} \\
i & ::= \text{interface } I \begin{align*} & \begin{cases} \text{with } l \begin{cases} \text{op mtd(in } d; \text{ out } d)^+ \end{cases} \end{cases} \text{ methods} \\
d & ::= x : T \begin{cases} \text{, x : T} \end{cases}^+ \\
T & ::= \text{Int | Bool | } \begin{cases} f \end{cases}^+ 
\end{align*}
\]

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 object attributes indicated by variable declarations \( \text{var } x : T \) as in the previous grammar.
Those statements are defined by the following grammar defining statements.

\[
\begin{align*}
s & ::= \text{skip} \mid \text{var} \mid \text{if} \ b \ \text{then} \ s \ \text{else} \ s \ \text{end} \mid \text{while} \ b \ \text{do} \ s \ \text{end} \\
& \mid \text{release} \mid \text{await} \ b \mid \text{await} \ l \mid \text{block} \\
e & ::= b \mid i \mid e = e \mid \text{null} \mid \text{this} \mid \text{me} \mid \text{caller} \\
b & ::= x \mid \text{true} \mid \text{false} \mid b \land b \mid b \lor b \mid \neg b \mid i < i \mid i > i \mid e = e \\
i & ::= x \mid i + i \mid i - i \mid i \ast i \mid i / i \mid \ldots \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots \\
D & ::= T \mid \text{Label}([T, T^*]) \\
T & ::= \text{Bool} \mid \text{Int} \mid l
\end{align*}
\]

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 b 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 l or a Label which contains a list of types not containing Label in the brackets.

Appendix B. Semantics

In this section we repeat all formulae of the semantics presented in Section 3 as they are distributed over the text. Two additional formulae omitted in the main part of the paper concerning variable declarations and the if-statement are given. The function defining the semantics of programs:

\[M : \text{PROG} \to (\Sigma \to 2^{\Sigma \times H})\]

The block statement:

\[M(\text{block})(\sigma) = \{\}\]

The skip statement:

\[M(\text{skip})(\sigma) = \{(\sigma, \langle\rangle)\}\]

The declaration of a variable i adds a new mapping to the partial function \(\sigma\) representing the state:

\[M(\text{var} \ i : \text{Int})(\sigma) = \{(\sigma', \langle\rangle) \mid \exists v. \sigma' = (\sigma : i \mapsto v)\}\]

We note that i is a local variable such that it holds \(\sigma|_u = \sigma'|_o\).

The assignment statement:

\[M(x := e)(\sigma) = \{(\sigma', \langle\rangle) \mid \exists v. v = e(\sigma)\quad\sigma' = (\sigma : x \mapsto v)\}\]

Sequential composition of statements:

\[M(S_1; S_2)(\sigma) = \{(\sigma_2, \theta_2) \mid \exists \sigma_1. (\sigma_1, \theta_1) \in M(S_1)(\sigma), (\sigma_2, \theta_2) \in M(S_2)(\sigma_1)\}\]

The branching if-statement not given in the main part of the paper is presented here:

\[M(\text{if} \ b \ \text{then} \ S_1 \ \text{else} \ S_2 \ \text{end})(\sigma) = \{(\sigma_1, \theta) \mid B(b)\sigma_1. (\sigma_1, \theta) \in M(S_1)(\sigma)\} \cup \{(\sigma_1, \theta) \mid B(\neg b)\sigma, (\sigma_1, \theta) \in M(S_2)(\sigma)\}\]

Depending on the evaluation of b either the first or the second set of the union is selected which delivers the semantics of the corresponding branch.

The semantics of a while-loop:

\[M(\text{while} \ b \ \text{do} \ S \ \text{end})(\sigma_0) = \\{ (\sigma, \theta) \mid \exists k \in \mathbb{N}, (\sigma_1, \theta_1), \ldots, (\sigma_k, \theta_k) \text{ such that } \}
\begin{align*}
\sigma &= \sigma_k, \theta = \theta_k \ldots \theta_1, B(\neg b)\sigma_k, \\
&\text{for } i = 0, \ldots, k - 1 : B(b)\sigma_i, (\sigma_{i+1}, \theta_{i+1}) \in M(S)(\sigma_i) \}\}
\]

Having dealt with all sequential statements we turn to concurrent statements communicating via shared memory. The first statement is the release-statement:

$$M(\text{release})(\sigma) = \{(\sigma', (\text{yield}(\sigma_{|a})) \land (\text{resume}(\sigma'_{|a}))), | \sigma'| = |\sigma||\}
$$

The conditional await-statement occurs in two different versions, where the first deals with a Boolean condition \(b\) whereas the second refers to a label \(l\):

$$M(\text{await }b)(\sigma) = \{(\sigma', (\text{yield}(\sigma_{|a})) \land (\text{resume}(\sigma'_{|a}))), | \sigma'| = |\sigma|, \mathcal{B}(b)\sigma'\}$$

$$M(\text{await }l)(\sigma) = \{(\sigma', (\text{yield}(\sigma_{|a})) \land (\text{resume}(\sigma'_{|a}))) \land (\text{comp}(E(l)\sigma')) | \sigma'| = |\sigma||\}
$$

We proceed with concurrence by means of message passing using the invocation-statement and the completion-statement:

$$M(\text{!o.m}(\bar{x}))(\sigma) = \{(\sigma_1, \theta) |\exists oid, \exists i. \text{ oid} = E(\alpha)\sigma, v = E(\bar{x})\sigma, $$

$$\theta = (\text{invoc}(E(\bar{i})\sigma_1, v))\}
$$

$$M(\text{i?(y)})(\sigma) = \{(\sigma_1, \theta) |\exists \bar{y}. \text{ lv} = E(l)\sigma, \sigma_1 = (\sigma : \bar{y} \rightarrow \bar{v}), \theta = \text{comp}(\text{lv}, \bar{v})\}
$$

Now, the only statement missing is the object creation:

$$M(o := \text{new } C)(\sigma) = \{(\sigma_1, \theta) |\exists i. \text{ oid} = E(\alpha)\sigma, \theta = \text{new}(E(\text{this})\sigma, E(\alpha)\sigma_1)\}
$$

Given the semantics of all statements the semantics of a method looks as follows:

$$M(\text{op m}(\text{in } \bar{x}; \text{ out } \bar{y}) := \text{body})(\sigma) = \{(\sigma, \theta, b) \exists \bar{v}. \exists \sigma_1, \exists v. \theta = \text{comp}(\text{lv}(\bar{v}), \theta)\}
$$

For \(i\) threads of method \(m\) the semantics are:

$$M(m, i)(\sigma) = \begin{cases} \theta \downarrow \{\text{m}, j\} \mid j \in \{1, \ldots, i\} = \theta, \\ \forall j \in \{1, \ldots, i\}. \exists \theta. \exists \sigma_1, \exists \sigma_2, \exists \sigma_3. \\ t = \{m, j\}, \sigma_t = (\sigma : \text{me} \rightarrow t, \text{caller} \rightarrow o, \text{call} \rightarrow \bar{v}), \\ (\theta_t, \sigma_t) \in M(\text{op m}(\text{in } \bar{x}; \text{ out } \bar{y}) := \text{body})(\sigma_t), \\ \theta \downarrow \{t\} = \theta_t, \text{cond}_m(\theta) \end{cases}
$$

By the union over all numbers of threads we obtain the semantics of a method:

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

The semantics of an object combine the semantics of all methods of its class:

$$M(o) = \begin{cases} \theta' \exists \theta. \theta \downarrow \{m\} \mid m \text{ is method of } C = \theta, \\ \forall \text{ methods } m \text{ of } C. \exists \theta_m. \exists \theta'. \\ \theta_m \in M(m)(\text{this} \rightarrow o), \theta \downarrow \{m\} = \theta_m, \\ \theta' = (\text{yield}(M(\text{constr}_C))) \land (\text{cond}_m(\theta')), \\ \theta'' = (\text{created}(\theta', o)) \downarrow \text{MP} \end{cases}
$$

As for methods we first give the semantics of a class having \(i\) instances (objects):

$$M(C, i) = \begin{cases} \theta \downarrow \{(C, j) \mid j \in \{1, \ldots, i\} = \theta, \\ \forall j \in \{1, \ldots, i\}. \exists \theta_0. \theta = (C, j), \theta_0 \in M(o), \theta \downarrow \{o\} = \theta_0 \end{cases}
$$

The semantics of class \(C\) consist of the semantics of all possible numbers of objects:

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

Finally, the semantics of a program \(P\) are:

$$M(P) = \begin{cases} \exists \theta. \theta \downarrow \{C \mid C \text{ is class of } P\} = \theta, \\ \forall \text{ classes } C \text{ of } P. \exists \theta_C. \exists i. \\ \theta_C \in M(C), \theta \downarrow \{C\} = \theta_C, \\ o.\text{class} = \text{Main}, \theta' = (\text{new}(.., o)) \land (\text{invoc}(.., (\text{main}, i), \bar{v})) \land (\text{type}(\theta_1) \land \text{type}(\theta_2) = \emptyset) \\ \forall \text{ type } \theta_1, \forall \theta_2. \theta' = \theta' \Rightarrow \text{get}_{\text{type}}(\theta_1) \cap \text{get}_{\text{type}}(\theta_2) = \emptyset \\ \text{exists bijection } \text{f} : \text{get}_{\text{new}}(\theta') \rightarrow \text{get}_{\text{created}}(\theta'), \text{cond}_f(\text{f}) \\ \text{exists bijection } \text{g} : \text{get}_{\text{invoc}}(\theta') \rightarrow \text{get}_{\text{begin}}(\theta'), \text{cond}_g(\text{g}) \\ \text{exists function } \text{h} : \text{get}_{\text{comp}}(\theta') \rightarrow \text{get}_{\text{end}}(\theta'), \text{cond}_h(\text{h}) \end{cases}
$$
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.

\[
\begin{align*}
\text{skip} & \vdash \{ \omega \} \phi \\
\{ \text{skip; } \omega \} \phi & \quad \text{assign} \quad \{ x := te \} \{ \omega \} \phi
\end{align*}
\]

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.

\[
\begin{align*}
\text{intDecl} & \quad \{ i := 0 \} \{ \omega \} \phi \\
\{ \text{var } i : \text{Int; } \omega \} \phi & \\
\text{boolDecl} & \quad \{ b := \text{false} \} \{ \omega \} \phi \\
\{ \text{var } b : \text{Bool; } \omega \} \phi
\end{align*}
\]

\[
\begin{align*}
\text{objDecl} & \quad \{ o := \text{null} \} \{ \omega \} \phi \\
\{ \text{var } o : \text{I; } \omega \} \phi & \\
\text{labelDecl} & \quad \{ l := \text{null} \} \{ \omega \} \phi \\
\{ \text{var } l : \text{Label}; \omega \} \phi
\end{align*}
\]

A division by zero could occur arbitrarily 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.

\[
\begin{align*}
\{ x' := e_2; x := te_1 + x'; \omega \} \phi & \quad \{ x' := e_1; x := x' + te_2; \omega \} \phi \\
\{ x := e_1 + e_2; \omega \} \phi & \\
\{ x := e_1 + te_2; \omega \} \phi
\end{align*}
\]

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

\[
\begin{align*}
\text{DivTerminal} & \quad (\neg te_2 \equiv 0 \rightarrow \{ x := te_1/te_2 \} \{ \omega \} \phi) \land (te_2 \equiv 0 \rightarrow \{ \text{block; } \omega \} \phi) \\
\{ x := te_1/te_2; \omega \} \phi
\end{align*}
\]

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

\[
\begin{align*}
\text{ifUnfold} & \quad \{ x := b; \text{if } x \text{ then } p \text{ else } q \text{ end; } \omega \} \phi \\
& \quad \{ \text{if } b \text{ then } p \text{ else } q \text{ end; } \omega \} \phi
\end{align*}
\]

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 \( tb \).

\[
\begin{align*}
\text{if} & \quad (tb \equiv \text{true} \rightarrow \{ p; \omega \} \phi) \land (tb \equiv \text{false} \rightarrow \{ q; \omega \} \phi) \\
\{ \text{if } tb \text{ then } p \text{ else } q \text{ end; } \omega \} \phi
\end{align*}
\]

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

\[
\begin{align*}
\text{new} & \quad \{ o := \text{(this, (c, i))}; U_{\mathcal{H}}^{\text{new}} \} \{ \omega \} \phi, \Delta \\
\Gamma & \vdash \{ \Delta \}
\end{align*}
\]

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 \( U_{\mathcal{H}}^{\text{new}} = \mathcal{H} := \text{some } H. (\mathcal{H} \leq H \land \text{New}(\mathcal{H}, o)) \).

References
