A Verification System for Distributed Objects with Asynchronous Method Calls *

Wolfgang Ahrendt[1] and Maximilian Dylla[1 2]

[1] Chalmers University of Technology, Gothenburg, Sweden
[2] Saarland University, Saarbrücken, Germany

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

1 Introduction

The area of object-oriented program verification made significant progress during the last decade. Systems like Boogie [6], ESC/Java2 [23], KeY [9], and Krakatoa [22] provide a high degree of automation, elaborate user interfaces, extensive tool integration, support for various specification languages, and high coverage of a real world target language (Spec# in case of Boogie, Java in case of the other mentioned tools).

However, this development mostly concerns *sequential*, *free-standing* applications. When it comes to verifying functional properties of *concurrent* and *distributed* applications, the situation is different. Even if there is a rich literature on the verification of 'distributed formalisms' (based for instance on process calculi [35, 27, 36]), 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

* This work has partially been supported by the EU-project FP7-ICT-2007-3 *HATS: Highly Adaptable and Trustworthy Software using Formal Methods.* and the EU COST action IC0701: *Formal Verification of Object-Oriented Software.*

object-oriented systems [32], and the KeY approach and system for the verification of object-oriented programs [9]. By combining KeY's proving technology with Creol's novel approach to modular modeling of components, we achieve a *system for highly modular verification of concurrent, distributed object-oriented applications*. While being a prototype system yet, past experience with the technological and conceptual basis justifies the perspective of future versions to enjoy similar features as state-of-the-art sequential verification systems already do.

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* [31], 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 suggest *distributed* scenarios, with each object being identified with a node. The second type of concurrency is object internal. Each call to a method spawns a separate thread of execution. Within one object, these threads execute *interleaved*, with only one thread running at a time. Here, the key to modularity is the cooperative nature of the scheduling: a thread is only ever interrupted when it actively releases control, at 'release points'.

Altogether, Creol allows highly *modular 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 specifications are expressed in terms of the history, and 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 [30].

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 [9]. Other target languages of KeY are C [39], ASMs [40], and hybrid systems [42]. All these have in common that they use *dynamic logic*, *explicit substitutions*, and a *sequent calculus* realized 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 [9]. 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* [38], the *Mondex* case study (the most substantial benchmark in the Grand Challenge repository) [44], the *Schoor-Waite algorithm* [12], and the electronic purse application *Demoney* [37]. The system is also used for teaching in various courses at Chalmers University and several other universities.

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

The structure of the paper is as follows. Sect. 2 introduces Creol, and gives examples of its usage. In Sect. 3, we describe the logic and calculus characteristic for KeY, insofar as they are (largely) independent of the particular target language. Thereafter, Sect. 4 presents a KeY style logic and calculus for Creol specifically. Sect. 5 discusses system oriented aspects of KeY for Creol, including a small account on taclets. Sect. 6 then demonstrates the usage of the systems in examples. In Sect. 7, we discuss related work, and draw conclusions.

2 Overview of Creol

In this section, we introduce our slightly adapted version of Creol, using an automated teller machine scenario adapted from [29]. 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.

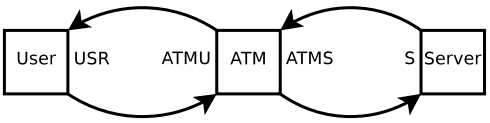

Fig. 1. Communication of the automated teller machine

The picture shows that, while User and Server implement one interface each (USR resp. S), the class ATM implements two interfaces, ATMU and ATMS,

[3] See Sect.7 for an exception.

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.)

```
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
```

Fig. 2. The interfaces of the automated 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 initialized 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 corresponding reply message has already arrived*. Otherwise, the statement blocks, without the thread releasing control. (This 'busy waiting' can be avoided by the **await** statement, see below.) The effect of li?(x) is similar to treating x as a *future variable* [15, 5] or *promise* [34]. 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

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

Fig. 3. The class implementing the teller machine

object.) In general, arbitrary code can be executed in between a call and the corresponding reply. We want to highlight that the implementation of insert extensively uses the caller reference, which is known to be of type USR, for callbacks. This style of coupling communicating objects might clarify the distribution of operations over interfaces in the teller machine scenario (cf. Fig. 2).

We discuss further features of Creol not captured by the above example. New objects are created by $x :=$ **new** $C(e^*)$, where C is a class identifier supplied with a list of class parameters. As indicated earlier, $l?(x^*)$ blocks execution, *without releasing control*, until the 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 Sect.6.1 (Fig. 7).

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 The KeY approach: Logic, Calculus, and System

3.1 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) [25], which, like Hoare logic [26], is transparent with respect to the programs that are subject to verification. DL

is a particular kind of *modal logic*. Different parts of a formula are evaluated in different worlds (states), which vary in the interpretation of functions and predicates. The modalities are 'indexed' with pieces of program code, describing how to reach one world (state) from the other. DL extends full first-order logic with two additional (mix-fix) operators: $\langle\,.\,\rangle\,.$ (diamond) and $[\,.\,]\,.$ (box). In both cases, the first argument is a *program* (fragment), whereas the second argument is another DL formula. A formula $\langle p\rangle\phi$ is true in a state s if execution of p terminates when started in s and results in a state where ϕ is true. As for the other operator, a formula $[p]\phi$ is true in a state s if execution of p, when started in s, *either* does not terminate *or* results in a state where ϕ is true. In other words, the difference between the operators is the one between total and partial correctness.[4]

DL is closed under all logical connectives. For instance, the following formula states equivalence of p and q w.r.t. the "output", the program variable x.

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

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

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

All major program logics (Hoare logic, wp calculus, DL) have in common that the resolving of assignments requires substitutions in the formula, in one way or the other. In the KeY approach, the effect of substitutions is delayed, by having *explicit substitutions* in the logic, called *'updates'*. This allows for accumulating and simplifying the effect of a program, in a forward style. Elementary updates have the form $x := e$, where x is a location (in the case of Creol, an attribute or local variable) and e is a (side-effect free) expression. Elementary updates are combined to simultaneous updates, like in $x_1 := e_1 \,|\, x_2 := e_2$, where e_1 and e_2 are evaluated in the same state. For instance, $\mathsf{x} := \mathsf{y} \,|\, \mathsf{y} := \mathsf{x}$ stands for exchanging the values of x and y. Updates are brought into the logic via the update modality $\{\,.\,\}\,.$, connecting arbitrary updates with arbitrary formulas, like in $\mathsf{x} < \mathsf{y} \rightarrow \{\mathsf{x} := \mathsf{y} \,|\, \mathsf{y} := \mathsf{x}\}\,\mathsf{y} < \mathsf{x}$. A typical usage of updates during proving is in formulas of the form $\{\mathcal{U}\}\langle p\rangle\phi$, where $\mathcal{U}$ is an update, accumulating the effects of program execution up to a certain point, p is the remaining program yet to be executed, and ϕ a postcondition. A full account of KeY style DL is found in [11].

[4] Just as in standard modal logic, the diamond vs. box operators quantify existentially vs. universally over states (reached by the program). In case deterministic programs, however, the only difference between the two is whether termination is claimed or not.

3.2 Sequent Calculus

The heart of KeY, the prover, uses a sequent calculus for reducing proof obligations to axioms. A sequent is a pair of sets of formulas written as $\phi_1, ..., \phi_m \vdash \psi_1, ..., \psi_n$. The intuitive meaning is that, if all $\phi_1, ..., \phi_m$ hold, at least one of $\psi_1, ..., \psi_n$ must hold. 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 [24] for the full set). $\phi[v/e]$ denotes a formula resulting from replacing v with e in ϕ.

$$\mathsf{impRight}\ \frac{\Gamma, \phi \vdash \psi, \Delta}{\Gamma \vdash \phi \to \psi, \Delta} \qquad \mathsf{andRight}\ \frac{\Gamma \vdash \phi, \Delta \qquad \Gamma \vdash \psi, \Delta}{\Gamma \vdash \phi \wedge \psi, \Delta} \qquad \mathsf{allRight}\ \frac{\Gamma \vdash \phi[v/c], \Delta}{\Gamma \vdash \forall v.\phi, \Delta}$$

with c a new constant

Fig. 4. A selection of first-order rules

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

$$\frac{\lfloor\, \phi' \,\rfloor}{\lfloor\, \phi \,\rfloor}$$

for a rule stating that the premise sequent $\lfloor\, \phi' \,\rfloor$ is constructed by replacing ϕ with ϕ' *anywhere* in the conclusion sequent $\lfloor\, \phi \,\rfloor$. In Fig. 5 we present some rules dealing with statements. (assign and if are simplified, see Sect. 4.1.) The schematic modality $\langle\!\langle \cdot \rangle\!\rangle$ can be instantiated with both $[\cdot]$ and $\langle \cdot \rangle$, though consistently within a single rule application. Total correctness formulas of the form $\langle \textbf{while} \ ... \rangle \phi$ are proved by combining induction with unwind.

$$\mathsf{assign}\ \frac{\lfloor\, \{x := e\}\langle\!\langle \omega \rangle\!\rangle \phi \,\rfloor}{\lfloor\, \langle\!\langle x := e; \omega \rangle\!\rangle \phi \,\rfloor} \qquad \mathsf{if}\ \frac{\lfloor\, (b \to \langle\!\langle s_1; \omega \rangle\!\rangle \phi) \wedge (\neg b \to \langle\!\langle s_2; \omega \rangle\!\rangle \phi) \,\rfloor}{\lfloor\, \langle\!\langle \textbf{if}\ b\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{end}; \omega \rangle\!\rangle \phi \,\rfloor}$$

$$\mathsf{unwind}\ \frac{\lfloor\, \langle\!\langle \textbf{if}\ b\ \textbf{then}\ s; \textbf{while}\ b\ \textbf{do}\ s\ \textbf{end end}; \omega \rangle\!\rangle \phi \,\rfloor}{\lfloor\, \langle\!\langle \textbf{while}\ b\ \textbf{do}\ s\ \textbf{end}; \omega \rangle\!\rangle \phi \,\rfloor}$$

Fig. 5. Dynamic logic rules

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

in the next step merged into a single simultaneous update. Once the box or diamond modality is completely resolved, the entire update is applied to the postcondition.

4 A Calculus for Creol Dynamic Logic

Building on the logic and the calculus presented in the previous section, we proceed with the sequent rules handling Creol statements. For the full set of rules, see [20].

4.1 Sequential Constructs

We start with assignments. As soon as the right side is simply a variable or literal (summarized 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.

$$\mathsf{assign}\ \frac{\lfloor\, \{x := te\}\langle\!\langle \omega \rangle\!\rangle \phi \,\rfloor}{\lfloor\, \langle\!\langle x := te;\ \omega \rangle\!\rangle \phi \,\rfloor} \qquad \mathsf{intDecl}\ \frac{\lfloor\, \{i := 0\}\langle\!\langle \omega \rangle\!\rangle \phi \,\rfloor}{\lfloor\, \langle\!\langle \textbf{var}\ i : \textbf{Int};\ \omega \rangle\!\rangle \phi \,\rfloor}$$

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

$$\mathsf{DivTerminal}\ \frac{\lfloor\, (\neg te_2 \doteq 0 \rightarrow \{x := te_1/te_2\}\langle\!\langle \omega \rangle\!\rangle \phi) \wedge (te_2 \doteq 0 \rightarrow \langle\!\langle \textbf{block};\ \omega \rangle\!\rangle \phi) \,\rfloor}{\lfloor\, \langle\!\langle x := te_1/te_2;\ \omega \rangle\!\rangle \phi \,\rfloor}$$

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

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

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

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

Note that application of this rule may lead to proof branching in subsequent steps. As for **while**, the unwind rule was presented in Sect. 3.2. An alternative

rule using a loop invariant is discussed in section 4.3. 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:

$$\mathsf{blockBox}\ \frac{\lfloor\, \mathit{true}\,\rfloor}{\lfloor\, [\mathbf{block};\, \omega]\phi\,\rfloor}\ \mathsf{blockDia}\ \frac{\lfloor\, \mathit{false}\,\rfloor}{\lfloor\, \langle\mathbf{block};\, \omega\rangle\phi\,\rfloor}$$

4.2 Interface and Class Invariants

The verification process of Creol programs is completely modular. This means we verify only one method (of one class) at the time and do not consider any other code during this process. Instead, we take into account the other threads of the object by guaranteeing the class invariant at release points and assuming it again when execution proceeds. As for the behavior of other objects, that is represented by using specification 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 [17, 19].

The communication history can be viewed as a list of messages of method invocations, method completions, and object creations. For modular reasoning we always consider projections of the system wide history $\mathcal{H}$. Every interface is specified by an interface invariant $\mathit{inv}_I(\mathcal{H}/o/I)$, with o ranging over objects of type I. The system wide history $\mathcal{H}$ is projected $(\mathcal{H}/o/I)$ to messages concerning o and talking about methods declared in I. During verification at method calls and replies, $\mathcal{H}/\mathsf{this}/I$ is checked against the specification. Continuing the previous example of Fig. 2 the interface USR is equipped with the following invariant:

$$\mathcal{H}/o/\mathsf{USR} \leq (\rightarrow \mathsf{giveCode}[\cdot \rightarrow \mathsf{withdraw}[\cdot \rightarrow \mathsf{dispense}]]\cdot \rightarrow \mathsf{returnCard})^*$$

where $\cdot$ is appending, $\rightarrow$ are invocation messages, $\leftarrow$ are completion messages, brackets are used for optional occurrence, and * is the Kleene star. The parameters and communication partners are omitted for brevity. The invariant expresses that the history of the 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:

$$\mathcal{H}/o/\mathsf{S} \leq \left(\rightarrow \mathsf{authorize}(\mathit{cid}, .)\cdot \begin{pmatrix} \leftarrow \mathsf{authorize}(\mathit{false})| \\ \leftarrow \mathsf{authorize}(\mathit{true})\cdot \rightarrow \mathsf{debit}(\mathit{cid}, .)\cdot \leftarrow \mathsf{debit}(.) \end{pmatrix}\right)^*$$

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

We turn to the class invariant $\mathit{inv}_C(\mathcal{H}/\mathsf{this}, \overline{\mathcal{W}})$, which forms a contract between all threads of the object. $\overline{\mathcal{W}}$ is the vector of class attributes. Those might get overwritten by other threads during suspension of this thread, but the invariant expresses properties of $\overline{\mathcal{W}}$ every thread is respecting. The class invariant

is parametrized by $\mathcal{H}/\mathsf{this}$ which is the projection of the system wide history to the object the invariant belongs to. It contains all messages sent to or by the object this. A class invariant consists of several parts:

$$inv_C(\mathcal{H}/\mathsf{this}, \overline{\mathcal{W}}) \triangleq F(\mathcal{H}/\mathsf{this}, \overline{\mathcal{W}}) \wedge Wf(\mathcal{H}/\mathsf{this}) \wedge \forall obj \bigwedge_I inv_I(\mathcal{H}/\mathsf{this}/obj/I)$$

$F(\mathcal{H}/\mathsf{this}, \overline{\mathcal{W}})$ relates the state of the ordinary class attributes $\overline{\mathcal{W}}$ with the history, reflecting the refinement of the fully abstract interface specification to the local state. $Wf(\mathcal{H}/\mathsf{this})$ is a predicate being interpreted to true for well-formed histories. A well-formed history starts with the creation message of this, contains invocation messages for all completion messages, and does not include any object references being null. Then, all invariants of all interfaces I invoked or implemented by the class of this put in a conjunction to ensure that all methods respect them. *obj* are the objects known by this. 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 its completion message.

$$\vdash inv_C(\mathcal{H}/\mathsf{this}, \overline{\mathcal{W}}) \rightarrow [body] inv_C(\mathcal{H}/\mathsf{this}, \overline{\mathcal{W}}) \tag{1}$$

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

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

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

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

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

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

In the system such equations are realized as taclets (see Sect. 5).

4.3 Concurrent Constructs

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

concerning the first and focus on the latter further below. In this section we abbreviate $\mathcal{H}/\mathsf{this}$ by $\mathcal{H}$.

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

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

Here, $U_{\mathcal{H},\overline{\mathcal{W}}}$ is the update $\mathcal{H}, \overline{\mathcal{W}} := \mathsf{some}\ H, \overline{W}.(inv_C(H, \overline{W}) \wedge \mathcal{H} \leq H)$. The update $U_{\mathcal{H},\overline{\mathcal{W}}}$ represents an arbitrary but fixed system state satisfying the class invariant in which execution continues. By $\mathcal{H} \leq H$ we denote that the old history $\mathcal{H}$ is a prefix of the new one H. The update is necessary because values of the class attributes could have been overwritten by other threads, and because $\mathcal{H}$ might have grown meanwhile.

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 [11]).

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

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

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

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

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

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

$$\mathcal{H}, \overline{mod} := \mathsf{some}\ H, \overline{m}.(Wf(H) \wedge \mathcal{H} \leq H \wedge inv_{loop}(H, \overline{m}))$$

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

Analogous to $Comp(\mathcal{H}, l)$ there are predicates $Invoc(\mathcal{H}, l)$ and $New(\mathcal{H}, o)$ which guarantee the existence of an invocation message with label l and an object creation message with reference o in the history $\mathcal{H}$, respectively. During a proof, uncertainty is inherent in the projection of the history to this, as there could be incoming method invocations at any time. When dealing with method calls we only state the existence of a corresponding message in the history. We do not append it to the history. In general all rules of Sect. 4.1 would need to cover potential extensions, using the prefix predicate $\leq$. It is however equivalent to extend the history on access (release points, method calls, etc.).

To exemplify some properties of the predicates dealing with the history we give the following formula which is a tautology.

$$Comp(\mathcal{H}_0, l) \wedge \mathcal{H}_0 \leq \mathcal{H}_1 \rightarrow Comp(\mathcal{H}_1, l) \tag{2}$$

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

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

$$\mathsf{invoc}\ \frac{\begin{array}{l}\Gamma \vdash Wf(\mathcal{H}) \wedge inv_I(\mathcal{H}/o/I), \Delta \\ \Gamma \vdash o \doteq \mathsf{null} \rightarrow \langle\!\langle \mathbf{block}; \omega \rangle\!\rangle \phi, \Delta \\ \Gamma \vdash \neg o \doteq \mathsf{null} \rightarrow \{l := (\mathsf{this}, o, mtd, \overline{p_{in}}, i)\}\{U^{invoc}_{\mathcal{H}}\}\langle\!\langle \omega \rangle\!\rangle \phi, \Delta\end{array}}{\Gamma \vdash \langle\!\langle l!o.mtd(\overline{p_{in}}); \omega \rangle\!\rangle \phi, \Delta}$$

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

$$\mathcal{H} := \mathsf{some}\ H.(\, Wf(H) \wedge \mathcal{H} \leq H \wedge inv_I(H/o/I, \overline{p_{in}}) \wedge Invoc(H, l) \wedge \neg Invoc(\mathcal{H}, l))$$

The new history contains the invocation message $Invoc(H, l)$. As the label l is unique the invocation message must not be included in the previous history ($\neg Invoc(\mathcal{H}, l)$), which prefixes the new one ($\mathcal{H} \leq H$). The new history H is well-formed ($Wf(H)$) and it respects the interface invariant $inv_I(H/o/I, \overline{p_{in}})$ where the in-parameters $\overline{p_{in}}$ are added as they occur in the appended invocation message.

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

$$\mathsf{comp}\ \frac{\begin{array}{l}\Gamma \vdash Invoc(\mathcal{H}, l) \wedge Wf(\mathcal{H}) \wedge inv_I(\mathcal{H}/l.callee/I), \Delta \\ \Gamma \vdash l \doteq \mathsf{null} \rightarrow [\mathbf{block}; \omega] \phi, \Delta \\ \Gamma \vdash \neg l \doteq \mathsf{null} \rightarrow \{U^{comp}_{\mathcal{H}, \overline{p_{out}}}\}[\omega] \phi, \Delta\end{array}}{\Gamma \vdash [l?(\overline{p_{out}}); \omega] \phi, \Delta}$$

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

$$\mathcal{H}, \overline{p_{out}} := \mathsf{some}\ H, \bar{p}.\ \begin{pmatrix} Wf(H) \wedge \mathcal{H} \leq H \wedge inv_I(H/l.callee/I, \bar{p}) \\ \wedge Comp(H, l) \wedge \neg Comp(\mathcal{H}, l) \end{pmatrix}$$

We omit the rule for object creation, mentioning only that the new reference is constructed by the pair (this, i), here 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 **return** statement. It sends the completion message belonging to the method call of the verification process and the thread terminates afterwards. The class invariant is not explicitly mentioned in the following rule as it is contained in ϕ (see previous section).

$$\mathsf{return}\ \frac{\Gamma \vdash Invoc(\mathcal{H}, l) \wedge Wf(\mathcal{H}) \wedge inv_I(\mathcal{H}/\mathsf{caller}/I), \Delta \qquad \Gamma \vdash \{U^{return}_{\mathcal{H}}\}\phi, \Delta}{\Gamma \vdash \langle\!\langle \mathbf{return}(\overline{p_{out}}) \rangle\!\rangle \phi, \Delta}$$

Here, l is the label of the message which created the thread subject to verification, I the corresponding interface, and caller the corresponding caller. The update $U^{return}_{\mathcal{H}}$ adds the completion message to the history which must not occur in the previous history.

$$\mathcal{H} := \mathsf{some}\ H.(Wf(H) \wedge \mathcal{H} \leq H \wedge inv_I(H/\mathsf{caller}/I) \wedge Comp(H, l) \wedge \neg Comp(\mathcal{H}, l))$$

5 A System for Creol Verification

The verification system for Creol is based on KeY[9]. Written in Java and published under the GNU general public license, it is available from the project's website[5]. The current version is a prototype which provides the functionalities presented in this paper. It has a graphical user interface where the proof tree and open proof goals are displayed. Other features are pretty-printing and syntax-highlighting of the subformula/subterm currently pointed at with the mouse pointer. This enables a context sensitive menu offering only the rules applicable to the highlighted subformula/subterm. Apart from the rule name, tool-tips describe the effect of a rule. Besides interactive application of rules, automatic strategies can be configured. A more detailed description of the KeY interface is available in [3].

Problem files, logical rules, and axiomatizations of data types are written in the *taclet* language [43]. In Fig. 6 the rule impRight from Fig. 4 and the equation Eq. (2) are defined in the taclet language. A `find` describes the formula the rule

[5] `www.key-project.org`

is applicable to, `replacewith` specifies the replacement for the `find` formula, `assumes` characterizes further assumptions not subject to replacements, and `add` causes its argument to be added. The arrow `==>` indicates on which side of the sequent the formulas are found, replaced or added. Writing a semicolon between two occurrences of `replacewith` or `add` causes a branching. Taclets omitting the sequence arrow `==>` are rewriting rules applicable in all contexts.

```
impRight {\find(==> phi -> psi)      compMon {\find(Comp(H1,L) ==>)
          \replacewith(==> psi)               \assumes(Prefix(H1,H2) ==>)
          \add(phi ==>) }                     \add(Comp(H2,L) ==>) }
```

Fig. 6. Rules in the taclet language

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

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

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

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

The Creol parser is written in about 3900 lines of code using ANTLR as parser generator. The adaptions in the KeY-system took another 5000 lines. Finally, the rules written in the taclet language are about 1700 lines long.

6 Verification Examples

6.1 Unbounded buffer

We give an implementation for an unbounded first-in-first-out (FIFO) buffer. This example is adapted from [18]. 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, no element must equal null. We define $inv_I(\mathcal{H}, callee)$ (slightly simplified) as:

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

where I is FifoBuffer and in, out are defined as:

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

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

The implementation of the buffer, given in Fig. 7, uses a chain of objects where each of them can store one element. The attribute cell is null if the object does not store an element. In next the reference to the following chain of objects is stored. Requests are forwarded to it if the object cannot serve them alone. The variable cnt holds the number of elements stored in cell and all following objects. Calls of get on an empty buffer are suspended until there are elements in the buffer.

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

Fig. 7. The class implementing the buffer

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

cell and all following objects.

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

rest removes the first element of a sequence. Let us proceed with the class invariant. The attribute cnt equals the number of elements in cell and all following buffer cells. The interface invariant of FifoBuffer has to hold for both the interface called and implemented by the class. Additionally, we state that the sequence of values put into the current cell equals the sequence of values obtained from the buffer with the cell and the content of the following buffer appended.

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

In the above formula I, is instantiated by FifoBuffer and $\mathcal{H}$ is an abbreviation for $\mathcal{H}/\mathsf{this}$. If cell is null it is omitted. The example with the given specifications was proved interactively by the system. The method put was verified in 1024 proof steps and 80 branches, whereas get needed 587 proof steps and 43 branches. Great parts of the proof were transformations of the sequences the buffer was specified with. However they went rather smoothly as the problem of the equality of two sequences is human-readable even if the automated strategy gets stuck. It seems that a logical toolbox expressing sets, relations and other well-understood mathematical notions would simplify the process of specifying and verifying other case studies.

6.2 Automated teller machine

The example of the automated teller machine distributed throughout the paper was successfully verified in 2495 steps (27 branches) by the system. As the implementation of the class makes heavy use of asynchronous method calls and (co)interfaces, it has been shown that our system can deal with them. The amount of method calls produces a chain of prefixed histories where the monotonicity of properties has to be used often. This leads to a number of predicates expressing properties of histories on the left-hand-side of the sequent. Hence, the automated strategy must use the monotonicity with care to improve readability if a branch cannot be closed by it. The experiences with specifications in form of regular expressions were promising. They are easy to write down and a automated strategy can deal with them as the number of successor states is usually limited which narrows the search space of the proof.

7 Discussion and Conclusion

Creol's notion of inter-object communication is inspired by notions from process algebras (CSP [27], CCS [35], π-calculus [36]), which however model syn-

chronous communication mostly. Moreover, Creol differs from those in integrating the notion of processes in the object-oriented setting, using named objects and methods rather than named channels. This also introduces more structure to the message passing (calls, replies, caller references, cointerfaces). The message passing paradigm on the inter-object level is combined with the shared memory paradigm on the local inter-thread level. Early approaches to the verification of shared memory concurrency are interference freedom based on proof outlines [41] and the rely/guarantee method [33]. Other approaches use object invariants as a combined assumption/guarantee, both in the sequential setting to achieve modularity [7, 8], and in the concurrent setting [28]. 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 [14, 27], and were used for specification and verification for instance in [45, 16].

KeY is among the state-of-the-art approaches to the verification of (at first) sequential object-oriented programs, together with systems like Boogie [6], ESC/Java(2) [23], and Krakatoa [22]. 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 generalized substitutions in formalisms such as the B method [2]. Updates are, however, tailored to symbolic execution rather than modeling (for instance, conflicts are resolved via right-win). The KeY tool uses these updates not only for verification, but also for test case generation with high code based coverage [21] 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 [10]. Here, the main idea is to prove the correctness of all permutations of schedulings at once. In [1], concurrent correctness of Java threads is addressed by combining sequential correctness with interference freedom tests and cooperation tests.

Very related to our work is the extension of the Boogie methodology to concurrent programs [28], targeting concurrent Spec#. From the beginning, this work is deeply integrated into an elaborate formal development environment, with all the features mentioned the first paragraph of this paper. The methodology requires users to annotate code with commands in between which an object

is allowed to violate its invariant. This is combined with ownership of objects by threads. Just as in our system, invariants have to be established at specific points, and can be assumed at others. Also similar is the erasing of knowledge, there with the havoc statement, here with the *some* operator. Differences (apart from the asynchronous method calls) are the purely cooperative nature of our threads, and that our shared memory is object local, which makes ownership trivial. Connected to this is the inherently fully abstract specification of remote object interfaces, employing histories. The Boogie approach can simulate histories as well (see Fig. 1 in [28]), 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. Also, the automated strategies are very rudimentary yet. We currently achieve an automation of 90% (automatic per total proof steps), which is very low by our standards. As we are only at the beginning of the work on automated strategies tailored to Creol, there is great potential here. The true challenge has been the omnipresence of the history, and it is here that future research on verification in this domain will focus on. This concerns various levels: better support for history based specifications, like a library of frequently used queries on histories, or the usage of specification patterns [13], extended and configurable proof support for history based reasoning, and improved presentation on the syntax level and in the user interface.

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

Acknowledgments

The authors would like to thank Frank de Boer, Einar Broch Johnsen, Olaf Owe, and Martin Steffen for fruitful discussions on the subject, Richard Bubel and Markus Drescher for their comments on drafts of this paper, Richard Bubel moreover for his guidance concerning implementation issues, and the anonymous reviewers for detailed comments.

References

1. E. Ábrahám, F. S. de Boer, W.-P. de Roever, and M. Steffen. An assertion-based proof system for multithreaded Java. *Theor. Comput. Sci.*, 331(2-3):251–290, 2005.
2. J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge Universtity Press, 1996.
3. W. Ahrendt. Using KeY. In Beckert et al. [9], pages 409–451.

4. W. Ahrendt, F. S. de Boer, and I. Grabe. Abstract object creation in dynamic logic. In A. Cavalcanti and D. Dams, editors, *Proc. 16th International Symposium on Formal Methods (FM'09)*, LNCS. Springer, 2009. to appear.
5. H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. *SIGPLAN Not.*, 12(8):55–59, 1977.
6. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *4th International Symposium on Formal Methods for Components and Objects, Amsterdam, The Netherlands*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.
7. M. Barnett, R. DeLine, M. Fändrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
8. M. Barnett and D. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *Mathematics of Program Construction, 7th International Conference, Stirling, Scotland*, volume 3125 of *LNCS*, pages 54–84. Springer, 2004.
9. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.
10. B. Beckert and V. Klebanov. A dynamic logic for deductive verification of concurrent programs. In M. Hinchey and T. Margaria, editors, *Conference on Software Engineering and Formal Methods (SEFM)*. IEEE Press, 2007.
11. B. Beckert, V. Klebanov, and S. Schlager. Dynamic logic. In Beckert et al. [9], pages 69–177.
12. R. Bubel. The Schorr-Waite-Algorithm. In Beckert et al. [9], pages 569–587.
13. R. Bubel and R. Hähnle. Pattern-driven formal specification. In Beckert et al. [9], pages 295–315.
14. O.-J. Dahl. Can program proving be made practical? *Les Fondements de la Programmaion*, pages 57–114, Dec. 1977.
15. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 316–330. Springer, Mar. 2007.
16. F. S. deBoer. A Hoare logic for dynamic networks of asychronously communicating deterministic processes. *Theor. Comput. Sci.*, 274(1-2):3–41, 2002.
17. J. Dovland, E. B. Johnsen, and O. Owe. A Hoare logic for concurrent objects with asynchronous method calls. Technical Report 315, Department of Informatics, University of Oslo, 2006.
18. J. Dovland, E. B. Johnsen, and O. Owe. A compositional proof system for dynamic object systems. Technical Report 351, Department of Informatics, University of Oslo, 2008.
19. J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of dynamic systems: Component reasoning for concurrent objects. *Electron. Notes Theor. Comput. Sci.*, 203(3):19–34, 2008.
20. M. Dylla. A verification system for the distributed object-oriented language Creol. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, June 2009.
21. C. Engel and R. Hähnle. Generating unit tests from formal proofs. In Y. Gurevich and B. Meyer, editors, *Proceedings, 1st International Conference on Tests And Proofs (TAP), Zurich, Switzerland*, volume 4454 of *LNCS*. Springer, 2007.
22. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *Conference on Computer Aided Verification*, volume 4590 of *LNCS*. Springer, 2007.

23. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Conference on Programming Language Design and Implementation, Berlin*, pages 234–245. ACM Press, 2002.
24. M. Giese. First-order logic. In Beckert et al. [9], pages 21–68.
25. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. MIT Press, 2000.
26. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 583, Oct. 1969.
27. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
28. B. Jacobs, K. R. M. Leino, F. Piessens, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *Conference on Software Engineering and Formal Methods*, pages 137–147. IEEE Computer Society, 2005.
29. E. B. Johnsen and O. Owe. A compositional formalism for object viewpoints. In B. Jacobs and A. Rensink, editors, *Proceedings of the 5th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002)*, pages 45–60. Kluwer Academic Publishers, Mar. 2002.
30. E. B. Johnsen and O. Owe. Object-oriented specification and open distributed systems. In O. Owe, S. Krogdahl, and T. Lyche, editors, *From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl*, volume 2635 of *LNCS*, pages 137–164. Springer, 2004.
31. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
32. E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2), 2006.
33. C. B. Jones. *Development Methods for Computer Programs Including a Notion of Interference*. PhD thesis, Oxford University, UK, 1981.
34. B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Conference on Programming Language design and Implementation*, pages 260–267, New York, NY, USA, 1988. ACM.
35. R. Milner. *A Calculus for Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.
36. R. Milner. *Communicating and Mobile Systems: the Pi Calculus*. Cambridge Universtity Press, 1999.
37. W. Mostowski. The demoney case study. In Beckert et al. [9], pages 533–568.
38. W. Mostowski. Fully verified Java Card API reference implementation. In B. Beckert, editor, *Verify'07*, volume 259 of *CEUR WS*, July 2007.
39. O. Mürk, D. Larsson, and R. Hähnle. KeY-C: A tool for verification of C programs. In F. Pfenning, editor, *Proc. 21st Conference on Automated Deduction (CADE), Bremen, Germany*, volume 4603 of *LNCS*, pages 385–390. Springer-Verlag, 2007.
40. S. Nanchen, H. Schmid, P. H. Schmitt, and R. Stärk. The ASMKeY theorem prover. Technical Report 436, ETH Zürich, 2004.
41. S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.
42. A. Platzer and J.-D. Quesel. KeYmaera: A hybrid theorem prover for hybrid systems. In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR*, volume 5195 of *LNCS*, pages 171–178. Springer, 2008.
43. P. Rümmer. Construction of proofs. In Beckert et al. [9], pages 179–242.
44. P. H. Schmitt and I. Tonin. Verifying the Mondex case study. In M. Hinchey and T. Margaria, editors, *Proc. 5.IEEE Int.Conf. on Software Engineeging and Formal Methods (SEFM)*, pages 47–56. IEEE Press, 2007.
45. N. Soundararajan. Axiomatic semantics of communicating sequential processes. *ACM Trans. Program. Lang. Syst.*, 6(4):647–662, 1984.