Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 243 (2009) 105–120
www.elsevier.com/locate/entcs

Tool Support for Holistic Modelling of Distributed Embedded Systems in Creol

Marcel Kyas[1,2]

Department of Informatics
University of Oslo
Oslo, Norway

Abstract

A holistic approach to modelling embedded systems is advocated: Many aspects of a system should be analysed in isolation to keep the task manageable, but they often influence each other during integration in a way that the desired system becomes unrealisable. A tool-supported approach that aims at integrated models of different concerns based on formal methods is suggested to solve this problem. This approach uses Creol, which is a language designed for object-oriented modelling of distributed systems. We report on ongoing work on the design and the implementation of tools that support modelling, validation, and verification. We focus on sensor networks, which are distributed system that consists of many embedded devices with tight constraints on computational power, energy availability, and timeliness. The described tools are a compiler that performs static checks and optimisations, an interpreter that defines a formal semantics, and a prototypical LTL model checker. This supports seamless development with formal methods.

Keywords: Distributed systems, object-oriented systems, modelling, sensor networks

1 Introduction

Object-oriented programming is regarded as the leading paradigm for concurrent and distributed systems and is recommended by the RM-ODP [20]. It is popular for modelling real-time embedded systems [16]. Model-driven engineering, pushed by the increasing maturity of modelling languages and tools, is becoming more established among software designers and developers [32]. Traditional ad-hoc system models are supplemented with new formalisms like SysML [40] or other architecture description languages.

When building distributed embedded applications like sensor networks, many functional and non-functional aspects have to be considered during their design,

[1] Email: kyas@fi.uio.no

[2] This work has been supported by EU-project IST-33826 *CREDO: Modelling and analysis of evolutionary structures for distributed services* (http://credo.cwi.nl).

doi:10.1016/j.entcs.2009.07.008

because these systems are composed of many components that collaborate to perform their function. The system is expected to perform reliably and for a prolonged time, despite the resource constraints on computational power, energy, and memory. As a result, the models and the design of such a system must integrate these concerns on both the local level of every device and the global level of the whole network: functionality, timeliness, power consumption, memory use, and more.

We report on the status of our development of the modelling language Creol that allows to address all these issues in a holistic manner. It is generally accepted that separating concerns simplifies the analysis of algorithms and programs, but many concerns must be considered together in order to find a compromise.

For example, sensor networks consist of many small devices which measure some environmental data, perform some processing, and send the data to a dedicated sink using a radio. Because this sink need not be in the range of a sensor's radio, network protocols are used to forward the messages on to the sink. Sending messages and listening uses energy. Hence, it is beneficial to send messages as seldom as possible and to listen as little as possible. This decreases the likelihood that a message reaches its destination within its deadline. The goal to maximise messages delivery conflicts with the goal minimise energy use to extend the operation time of a device.

This illustrates the need for a holistic approach, which we describe in Section 2. Cross-layer design [38] for implementing communication protocols is one way to enable a holistic approach. Since standard protocols with their layered abstractions are not easily adapted to the particular needs of the application, designers will often integrate lower layers into their design in order to tune parameters and algorithms at lower levels.

Our modelling language is Creol, an object-oriented language that supports distribution by *asynchronous method calls* [26]. Creol is a statically typed language with a formal semantics defined in rewriting logic [30] and executable on Maude [25]. Creol is also designed with verification in mind and provides a proof theory that is relatively simple [11] when compared to proof theories for, e.g., Java [1]. The main feature of Creol is the concurrency model: Each object executes a multiset of activities which are scheduled cooperatively. We summarise Creol in Section 3. The tools we report on in this paper are:

Compiler A compiler that performs static checks on an input model and emits a model in a runtime syntax suitable for the interpreter or model checker described below. The compiler is described in more detail in Section 4.

Interpreter An interpreter that allows fine-grained simulation of models. The model checker is described in more detail in Section 5.

Model Checker A model checker that enables basic LTL model checking of some properties on finite-state models. The model checker is described in more detail in Section 6.

In the following sections we will highlight some design problems and their solutions and explain the different theories from which we integrated into our method.

2 A Holistic Approach

Especially when designing systems that are resource constraint it is less apparent how the different concerns of a model under development can be separated. If one concern is to be studied in isolation, we have to abstract from all other concerns. But when everything is combined into an executable, it must respect all the constraints of the target platform.

Modelling is crucial to our approach. Creol supports formal reasoning, interfaces for behavioural specifications, and classes containing the functionality of objects. For example, Creol's support for *multiple inheritance* allows to combine behaviour that has been defined in a class in isolation. For example, the Creol class displayed in Figure 2 (see p. 6) can be reused by any class that needs to route messages. Different routing algorithms can be defined in different classes and inherited by other classes that need to route messages.

Creol's simplicity forces the modeller to focus on the behaviour of the model. Memory management statements are absent. The expression language is functional and high-level. These consideration simplify Creol's proof theory considerably. Compositional reasoning is supported by our method, because Creol's proof theory is compositional [11] for functional properties.

Still, when one combines different concerns to establish properties of their composition, we prefer to stay in the same formalism. Moreover, many properties can be established compositionally with serious effort. For example, if two parts of an object consume at most x units of memory each, we can only conclude, that their composition uses only at most $2x$ units of memory. If the design goal is to limit memory consumption to x units of memory, we need to establish that both parts are never using their memory simultaneously. For this step, we would like to analyse the composition. Similarly, timeliness is equally hard to establish compositionally.

Here, we focus on integrating different analysis and validation methods for Creol into a development process. The modeller is assumed to be the expert in decomposing the model. We provide tools that help in analysing the model. Here, we focus on a compiler, a simulator, and a model checker. These tools take the same model as input and may be used to establish properties of the model. The compiler is used to insert statements for administrative tasks like memory management. This is necessary, because resources are constrained on the target device and the all timeliness constraints must be observed; this makes the use of a garbage collector infeasible. Other tools under development include a proof assistant and static validation tools for slicing.

We plan to add linguistic features to integrate more properties into the model. If such properties are expressed declarative at a high level of abstractions, we expect that they will not interfere too much on other aspects. We worked on integrating timing aspects into the modelling language and provide analysis for these aspects.

3 Creol in a Nutshell

Creol [26] is a modelling and programming language that aims to combine object orientation and distribution in a natural way [23]. It is object-oriented in the sense that classes are the fundamental structuring unit and that it features multiple inheritance and late binding. What sets Creol apart from other object-oriented languages is its concurrency model: In Creol, each object executes on its own virtual processor, and objects communicate only using *asynchronous method calls.* When an object O calls a method m of an object O', it sends an invocation message to O' along with arguments. Method m executes on the processor of O' and sends a reply to O once it is finished, with return values. Object O may continue executing while waiting for the reply of O'. This leads to increased parallelism. The asynchronous method call is chosen as the only inter-object communication primitive, which combines non-blocking message passing with the structure provided by the method concept.

Objects in Creol are active and reactive. Once the object is created, it executes autonomously. It also reacts to external calls. Objects may decide to become passive by terminating their activity, which is specified in a method called *run.* Any method call will supply a *handle* (a future variable [41,7]) to the activity for receiving the return value later, allowing the activity to continue with local computations.

Explicit processor release points, taking the form of **await** and **release** statements, affect the implicit internal control flow in Creol objects. Since there is only one virtual processor per object, at most one method m may execute at a given time for a given object; any other invocations must wait until m finishes or explicitly releases the processor. This *cooperative* approach to intra-object concurrency has the advantage that while a method is executing, it can assume that no other invocations are accessing the object's attributes between release points, leading to a programming and reasoning style reminiscent of monitors [19], although without explicit signalling. This also leads to increased parallelism when objects are waiting for replies and allows objects to combine active and reactive behaviour [24].

Reasoning about multi-threaded programs in a setting with synchronous method calls is highly complex [1]. Verification considerations suggest that all methods should be serialised, but synchronous communication gives rise to undesired and uncontrolled waiting, and possibly deadlock. This limitation is severe in a distributed setting this. Delays and instabilities may cause undesired waiting. A diverging method even blocks the evaluation of other method activations, which makes it hard to combine active and passive behaviour in the same object. Non-blocking (asynchronous) message passing gives better control, but does not provide the structure and discipline inherent to method calls.

Classes are the primary structuring mechanism in Creol. Classes support multiple inheritance. Classes are not types, instead classes are typed by *interfaces.* An interface is used to regulate what methods an object exports and what objects may access its methods.

Methods are only accessible to instances of its *co-interface*, giving fine-grained access control and specifying mutual dependencies: The co-interface is the type

Syntactic categories	*Definitions*
$C, I, m \in \mathsf{Names}$	$IF ::= \textbf{interface}\ I\ [\textbf{inherits}\ \{I\}]\ \textbf{begin}\ \{\textbf{with}\ I\ \{Sg\}\}\ \textbf{end}$
$t \in \mathsf{Label}$	$CL ::= \textbf{class}\ C\ [\{x : I\}]\ [\textbf{inherits}\ \{C\}\ [(\{e\})]]\ [\textbf{implements}\ \{I\}]$
$g \in \mathsf{Guard}$	$[\textbf{contracts}\ \{I\}]\ \textbf{begin}\ \{\textbf{var}\ \{x : I\ [:= e]\}\}\ \{M\}, \{\textbf{with}\ I\ \{M\}\}\ \textbf{end}$
$p \in \mathsf{MtdCall}$	$M ::= Sg == [\textbf{var}\ \{\{x\} : I\ [:= e]\};]\ s$
$s \in \mathsf{Stmt}$	$Sg ::= \textbf{op}\ m\ ([\textbf{in}\ \{x : I\}][\textbf{out}\ \{x : I\}])$
$x \in \mathsf{Var}$	$g ::= b \mid t? \mid g \wedge g \mid g \vee g$
$e \in \mathsf{Expr}$	$s ::= \textbf{begin}\ s\ \textbf{end} \mid s; s \mid s \,\square\, s \mid x := e \mid x := \textbf{new}\ C[(\{e\})]$
$o \in \mathsf{ObjExpr}$	$\mid \textbf{skip} \mid \textbf{if}\ b\ \textbf{then}\ s\ [\textbf{else}\ s]\ \textbf{end} \mid \textbf{while}\ b\ [\textbf{inv}\ b]\ \textbf{do}\ s\ \textbf{end}$
$b \in \mathsf{BoolExpr}$	$\mid [t]![o.]m(\{e\}) \mid t?(x) \mid \textbf{release} \mid \textbf{await}\ g \mid [\textbf{await}][o.]m(\{e\}; \{x\})$

Figure 1. The language syntax. Terms such as $\{e\}$, $\{x\}$, and $\{s\}$, denote lists over the corresponding syntactic categories and terms such as $[e]$ denote optional elements.

of the variable *caller*, which identifies the actual caller and provides a call-back mechanism. Method signatures specify input and output parameters, as well as pre- and postconditions. Additionally, class invariants associate protocols.

Creol classes are described by providing a list of formal constructor parameters, attributes, methods, and super-classes. In addition, the type of a class is specified by a list of interfaces it *implements* and a list of interfaces it *contracts*. Classes only inherit contracted interfaces, because classes usually redefine inherited behaviour. Such a separation supports code reuse while still enabling formal reasoning. The grammar of class definitions and interface declarations is shown in Figure 1. The notation $\{\ldots\}^*$ indicates a possibly empty list of elements, $\{\ldots\}^*_,$ a possibly empty, comma-separated list, and $\{\ldots\}^+$ a non-empty list. Clauses enclosed in square brackets $[\ldots]$ are optional.

A method body consists of variable declarations followed by a statement. A first-order functional expression language is assumed, whose terms are denoted by e.[3] Variable names are denoted by v. Comma-separated lists of terms are denoted by $\boldsymbol{e}$, and comma-separated lists of variables by $\boldsymbol{v}$. The **await** c statement behaves like **skip**, if c holds when the statement is executed, and otherwise the process is suspended until c holds (await synchronisation). The **release** statement will yield control to other processes unconditionally. The choice statement $S_1 \,\square\, S_2$ chooses non-deterministically between S_1 or S_2, if both do not suspend and suspends only if both S_1 and S_2 would suspend.

The communication primitive $\ell!o.m(\boldsymbol{e})$ calls a method m with arguments $\boldsymbol{e}$ to the object o and returns a *handle* ℓ with which the result of the call can be retrieved later. The statement $\ell?(\boldsymbol{v})$ is used to retrieve the return values and assign them to $\boldsymbol{v}$, potentially blocking the execution until the call returns. The expression $\ell?$ queries whether the call for ℓ has returned. Consequently, the statement **await** $\ell?$ suspends until the call returned. The expression $\ell?$ is only defined for positive occurrences, i.e., **await** $\neg\ell?$ is not a valid statement. The grammar of statements is given by production s in Figure 1.

[3] We chose a functional expression language, because evaluation of expressions should not have any side effects, as in C and Java. The language is *first-order* (i.e., without lambda abstractions), because higher-order languages allow to encode objects in the functional sub-language. This way, we have a strict separation between computations, expressed by functional expressions, states, represented by objects, and coordination, as expressed by statements.

```
class Flooding (network: Network, size: Int) contracts Node
begin
  var log: List[[Node,Int]]

  op prune == await length(log) = size;
     log := after(log, length(log) - (size / 2)) ;
     prune(;)

  op receive ==
    var sender: Node; var htl: Int;
    var id: Int; var msg: Data;
    network.receive(; sender, htl, id, msg);
    if sender /= this && htl > 0 && ~((sender, id) in log) then
      await length(log) < size ;
      log := log |- (sender, id) ;
      network.broadcast(sender, htl - 1, id, msg;)
    end
end
```

Figure 2. Creol model of a flooding routing algorithm. The function call `after(l,n)` returns the elements of the list l after the nth element. The function call `length(l)` returns the length of the list l.

An operational semantics is sketched in Section 5 and we refer the reader to [26] for a more detailed definition of the operational semantics.

Figure 2 shows a small example for a Creol model. It represents the implementation of routing by flooding. The `receive` method is called periodically to listen to the network and receive some data from it. The variable `htl` counts the maximum number of times this message will be forwarded. If the message has not been seen yet, i.e., it is not yet in `log`, it is broadcast on the network with a decreased `htl`. Interleaved with that activity a second task called `prune` removes the oldest messages from the log to avoid memory overflow.

The example shows that concurrent activities can be easily expressed in Creol. Moreover, the `await` statements coordinate both activities in a very high-level way. The log is only appended to if there is room and it is pruned when it is full.

4 Creol Compiler

Developing Creol models is supported by a compiler that translates a model into a run-time syntax used by an interpreter formulated in Maude [25] (see Section 5). The result can be simulated or analysed in Maude. The design and implementation of the compiler is standard [3]: The compiler front-end contains the parser of models written in Creol. The result of the front end is an *abstract syntax tree* representing the input. Analysis steps and transformations steps identify errors and insert auxiliary statements needed for resource management in the interpreter.

Creol's type system supports multiple types for each object, overloading of methods and functions, and universal polymorphism for data types, e.g., lists. As such, the type theory can be formulated in $F^{\omega}_{\wedge}$ [35]. This implies that type checking may be undecidable [36]. However, for most applications one can use a semi-decision procedure based on greedy unification for type reconstruction [6].

This type reconstruction algorithm may reject some well-typed models, because it may guess the wrong type or when the solution is ambiguous due to overloading:

For example, for an interface that declares two methods `op m(x:List[Bool])` and `op m(x:List[Int])`, the type checker cannot decide which method is meant in

the call $o.m(nil)$. Such a call might have unexpected behaviour, since one method is supposed to be called while the other one will be executed.

Even though the implemented type checker may fail to establish the type correctness of some model, we believe that a solution based on type reconstruction is more popular with modellers, since the types of terms need not be given, as it is done in type checking algorithms for $F^{\omega}_{\wedge}$ [9]. Writing just `nil` for the empty list and have the type reconstruction figure out that `nil` is, e.g., an empty list of integers is much nicer than writing `nil: List[Int]` in every place where this list is used.

Standard data flow analysis is used to insert *free* statements, which are used to release resources that are no longer needed. Especially in a distributed system such annotations greatly reduce the overhead of distributed garbage collectors and the amount of allocated resources. For many examples this analysis allows us to disable garbage collection altogether.

Dead value elimination resets the value of a local variable that will not be read on future computation paths, because the reset will not affect the semantics of the program. This may help to reduce the state space during model checking (see Section 6), because it identifies states that differ only by garbage in these variables [22].

Tail-call optimisations are used to further reduce memory use: When the result of a call at the tail of a method are only returned as the method's result, the compiler uses a form of *promises* [17], which are essentially a reference to the calling object and the handle of the reply it expects, to allow the caller to terminate immediately and to allow the callee to return the result directly. This has the side-effect of making model checking feasible for some models.

Note that standard *tail-call elimination* does not apply to object-oriented programs, because the called method is bound late, i.e., it is unknown at compile time.

The result of the compilation is a term suitable for the execution environment in Maude, which can also be model checked, as described in the following two sections.

5 Execution with Maude

Johnsen et. al. present an operational semantics that defines the behaviour of Creol programs in [26] as a rewrite theory in rewriting logic (RL) [30]. A *rewrite theory* consists of a signature that defines the function symbols of the language, equations between terms, and a set of rewrite rules. The (membership) equational logic is the functional sublanguage of RL and supports algebraic specification.

RL extends algebraic specification techniques with transition rules: The dynamic behavior of a system is captured by rewrite rules, supplementing the equations that define the term language. Assuming that all terms can be reduced to normal form, rewrite rules transform terms modulo the equations. A rewrite rule $t \longrightarrow t'$ **if** c may be seen as a *local transition rule* allowing an instance of the pattern t to evolve into the corresponding instance of the pattern t', where the optional condition c must hold for the main rule to apply. Several rules that can be applied to distinct sub-configurations can be executed in a *concurrent rewrite step*. Consequently, concurrency is implicit in rewriting logic semantics. Rules may be formulated at a

$$\langle O : C \mid \mathsf{Att} : A, \mathsf{Pr} : \langle L, S \,\square\, S'; S'' \rangle, \mathsf{Q} : M \rangle \longrightarrow \langle O : C \mid \mathsf{Att} : A, \mathsf{Pr} : \langle L, S; S'' \rangle, \mathsf{Q} : M \rangle \textbf{ if } \mathrm{enabled}_{A \circ L, M}(S) \tag{1}$$

$$\langle O : C \mid \mathsf{Att} : A, \mathsf{Pr} : \langle L, \mathit{await}\ b; S \rangle, \mathsf{Q} : M \rangle \longrightarrow \langle O : C \mid \mathsf{Att} : A, \mathsf{Pr} : \langle L, S \rangle, \mathsf{Q} : M \rangle \textbf{ if } [\![b]\!]_{A \circ L, M} \tag{2}$$

$$\langle O : C \mid \mathsf{Att} : A, \mathsf{Pr} : \langle L, \mathit{release}; S \rangle, \mathsf{PrQ} : W \rangle \longrightarrow \langle O : C \mid \mathsf{Att} : A, \mathsf{Pr} : \mathsf{idle}, \mathsf{PrQ} : W\ \langle L, S \rangle \rangle \tag{3}$$

$$\langle O : C \mid \mathsf{Att} : A, \mathsf{Pr} : \langle L, S \rangle, \mathsf{PrQ} : W, \mathsf{Q} : M \rangle \longrightarrow \langle O : C \mid \mathsf{Att} : A, \mathsf{Pr} : \mathsf{idle}, \mathsf{PrQ} : W\ \langle L, S \rangle, \mathsf{Q} : M \rangle \textbf{ if } \neg \mathrm{enabled}_{A \circ L, M}(S) \tag{4}$$

$$\langle O : C \mid \mathsf{Att} : A, \mathsf{Pr} : \mathsf{idle}, \mathsf{PrQ} : \langle L, S \rangle\ W, \mathsf{Q} : M \rangle \longrightarrow \langle O : C \mid \mathsf{Att} : A, \mathsf{Pr} : \langle L, S \rangle, \mathsf{PrQ} : W, \mathsf{Q} : M \rangle \textbf{ if } \mathrm{ready}_{A \circ L, M}(S) \tag{5}$$

Figure 3. Formal Semantics of Core Creol Statements

high-level of abstraction, similar to structural operational semantics. In fact, RL provides a framework unifying equational and operational semantics [31].

A state configuration is modeled as a multiset of terms representing local object states. Object states are commonly represented by terms $\langle o : C \,|\, a_1 : v_1, \ldots, a_n : v_n \rangle$, where o is the identity, C is its class, the a_i's are the names of the object's fields, and the v_i's are the corresponding values [8]. We adopt this form of presentation. Objects have attributes like Att, which holds the valuation of all the object's member variables, Pr, which holds the current activity, PrQ which holds the suspended activities, and Q, which holds the object's message queue. An activity is represented by a pair $\langle L, S \rangle$ of a valuation of the local variables L and the statements S that are to be executed. Needed auxiliary functions are defined in equational logic and evaluated in between transitions [30]. Multisets with identity element empty are constructed by juxtaposition, whereas semicolon is used as the associative constructor of lists, also with identity element empty. Variables of the operational semantics are written in upper case letters, whereas variables of the modelling language (as well as auxiliary functions) are written in lower case letters.

A state *configuration* is a multiset of objects, classes, interfaces, queues, and messages. There are three different kinds of rewrite rules:

(i) *Code execution rules* corresponding to the different program statements. Some of these rules, which are concerned with scheduling of local activities are displayed in Figure 3. Rule (4) expresses that a process without any enabled statements may be suspended. Rule (5) expresses that any ready process may be activated if the object is idle.

(ii) *Transport rules* move messages between objects.

(iii) *System rules* manage low-level activities such as table look-up for classes.

Remark that code execution and transport rules apply to local configurations and allow concurrent execution, whereas system rules also apply to the whole system.

The semantics of Creol is relatively simple, where the core of the interpreter is defined in just 13 execution rules. There is one additional transport rule and about 50 system level rules, of which 11 are concerned with scheduling.

6 Model Checking in Maude

Maude provides a model checker [12] that enables the verification of LTL properties of rewriting systems. A natural idea is to use state-space exploration to verify properties of Creol models. This allows rapid prototyping of transformations and semantic optimisations: how can the rewriting system be optimised to use less resources and how can the input program be adapted to make model checking feasible. Our current approach is not efficient, because we do not analyse a program represented in Maude but an interpreter executing that program.

On the other hand, direct model checking requires the translation of the model and its specification into the input language of a model checker, a proof of correctness of the translation, and a translation from the output of the model checker back to the input model to interpret counter examples. Furthermore, most model checkers do not support object creation during run-time directly.

Our model checking experiments are based on the interpreter described in Section 5 and executing the actual program on top of that interpreter. This is akin to a *deep embedding* into a theorem prover [28], with similar advantages and disadvantages. The most important advantage is that it is simpler to check whether the representation of the program is correct, whereas the most severe disadvantage is an increase in resource usage. Since Creol is not yet stable enough for a translation into the language of another model checker, we prefer the slower solution, where we can prototype new methods when needed.

Another advantage is that counterexamples need not be translated back to Creol models. Instead, Maude presents them in terms of Creol, showing program states and indicating the sequence of rules, which corresponds to the sequence of statements executed. Other model checkers present counterexamples in terms of their representation, displaying many details about the program which usually are invisible to the modeller.

Maude also provides a *profiler* which helps in identifying how often a rule is executed and what the overall cost of executing a rule is. The profiler was instrumental in identifying performance bottlenecks and in identifying where adaptations of the program can have the largest benefit.

We need to define abstractions which help to reduce the state space of the system we want to verify. Some abstractions can be formalised in the RL specification. These abstractions will be applied to all programs. Other abstractions and optimisations are computed by the compiler (see Section 4).

Models in Creol describe infinite state systems, because the interpreter identifies calls uniquely. Even finite state systems exhibit a huge state space, because communication is asynchronous and unordered and scheduling is non-deterministic. Therefore, the rewriting system has to be adapted for model checking.

The interpreter of Section 5 defines a rewrite theory that defines the behaviour of Creol models. This theory gives rise to a Kripke structure, where the relation is defined by the semantics of statements. The worlds are sets of configurations of the Creol model. Maude allows us to define assertions for use by its satisfaction

function $\models : \mathit{State} \to \mathit{Prop} \to \mathit{Bool}$.

When writing specifications we need to know the *identity* of objects beforehand, because Maude does not support quantification. It allows us to implement such quantification by iteration, but this approach does not allow us to define predicates which use the same object identity in different states unless that object occurs in both states. As a consequence, all objects must be created at initialisation time and naming of objects must be deterministic, because initialisation is performed by equations, which have to define a confluent and terminating rewriting system.

Given a set Ψ of assertions, the set of LTL formulas is defined by the grammar $\varphi ::= \psi \mid \varphi \wedge \varphi \mid \neg\varphi \mid \varphi \mathcal{U} \varphi \mid \Box\varphi \mid \Diamond\varphi$, where the temporal modalities have their usual meaning [37]. The next modality is not used, because they are not compatible with transactions (see below).

An assertion *must* hold in every processor release point, e.g., whenever an **await** or **release** statement is executed. However, the model checker checks states between these release points, in which an assertion need not hold. For example, executing choice statements is represented by a rule, because it is not deterministic, and gives rise for a new world in the Kripke structure. Therefore, the program is augmented with a Boolean variable that indicates when the assertion has to hold and the assertion is conditional on the value of that variable.

We introduce *transactions*, which subsume many independent steps into one atomic step [13]. Transactions are obtained by turning rules into equations and adding a *commit* statement that cause a rule application. These work, because scheduling is explicit in Creol and data of objects is strictly encapsulated. Evaluation of a statement in one object does not depend on the state of any other object. Overtaking of messages is allowed, which is necessary to guarantee confluence of the resulting equational theory.

Specifications involving the next modality $\bigcirc$ are not compatible with transactions: $y := 1; x := 2 \models \bigcirc x = 0$ holds if $x = 0$ holds in the initial state, but if the statements are executed in a transaction, i.e., $y, x := 1, 2$, it does not hold anymore. On the other hand, $x := 1; x := 2$ must be executed in sequence, since the assignments to x are not independent. Since the eventually modality $\Diamond$ does not refer to a next state and includes the current state, it is still compatible with transactions.

Observe that transactions complement partial order reduction techniques [15,27]. Partial order reduction techniques effectively eliminate choices by choosing one representative computation. Transactions are usually applied to *linearly* ordered sequences of statements and partial order reduction techniques do not apply in such situations. We believe that partial order reduction is a useful technique which applies to a much larger range of models and should therefore be implemented as part of the model checking algorithm and not as a transformation of the Creol model. We expect the largest benefit of partial order reductions in a reduction of the nondeterminism between *different* objects. This cannot be expressed on the level of classes.

Not all rules may be changed to equations. The Maude model checker considers states that result from rule applications only. A processor release point is reached

whenever an activity executes a **await** b or **release** statement, or when the execution of a method terminates. Properties have to hold at processor release points. Furthermore, the next activity may be chosen non-deterministically, hence we cannot use equations for scheduling. It is important to observe that the suspension itself is deterministic and is defined using an equation.

The changes to the interpreter are not specific to a model. All models benefit from these changes. And since the tight link between the adapted interpreter and Creol is preserved, model checking still provides counterexamples in terms of statements and states of Creol itself. Note that although transactions abstract from intermediate states, these intermediate states can be recovered by using the *search* command and *the original interpreter.*

Uniqueness of message identifiers makes a model infinite state: a model with an infinite number of method invocations will have infinitely many states. This problem is overcome by identifying messages with terms of the form $\mathrm{label}(\mathrm{caller}, \mathrm{callee}, m, \boldsymbol{e})$. This implies that if the parameters are identical, then two invocation of m from O to O' will have identical labels. This abstraction is sound, because Creol's communication model allows message overtaking. In the original interpreter matching of invocation and completion message was unique. The model checker only guarantees to get a completion message to an invocation with the same parameters.

Synchronous calls to other objects may be represented by asynchronous calls and blocking on the return value. For synchronous self-calls additional changes have been implemented. Essentially, we introduce an *auxiliary parameter* that records the depth of its current call chain. This allows to maintain the stack discipline for self-calls. However, an ambiguity remains, because calls with the same label may still occur by releasing control using **await** and **release** statements. In that case, however, the technical reason allowing such re-ordering is in the non-deterministic selection of the next enabled activity of an object's scheduler.

Although these changes are sufficient to ensure the intended behaviour of synchronous calls, a rapid growth of the process queue can be observed for recursive calls. We address this problem by introducing compiler optimisations, e.g. tail-calls are optimised. This technique does not cover all (recursive) synchronous calls and is thereby more a program optimisation than an interpreter enhancement, although the interpreter has to be adapted to provide facilities to cope with tail-calls.

To prevent objects from sending an arbitrary number of invocations, process queues are bounded in their size. The number of invocation and completion messages is limited to n. While limiting the number of invocation messages is harmless, since these can always be received, limiting the number of activations in the process queue of an object may introduce deadlocks, and will generally lead to an under-approximation of the behaviour. We do not have a general method which will select a suitable queue size for a program. The modeller has to define this number large enough to still observe all behaviour he is interested in.

Alternatively, bounded fair schedulers generate queue bounds in a more flexible manner. *Bounded fair scheduling* specifications [39] are sets of constraints of the form (E, F, L), where $L \in \mathbb{N}$ is a bound, and E and F are sets of method names.

The intuitive meaning of a constraint is to accept at most L calls to a method of F before any occurrence of a call to a method in E. Such specifications can be translated into a Streett automaton, where the number of states are an upper bound to the queue size. In addition, they define a fair scheduling, which further reduces the number of states analysed, since not all possible orderings of events are considered.

7 Related work

Asynchronous message passing is well-known from, e.g., Actors [2]. Languages which support *future variables* are usually based on asynchronous messages; the caller's activity is synchronised with the arrival of the completion message rather than with the emission of the call, and the activities of the caller and the callee need not directly synchronise [41,7,4,21]. This approach seems well-suited for distributed environments, reflecting the fact that communication in a network takes time. However, method calls imply an ordering on communication not easily captured in the Actor model. Actors do not distinguish completion messages from invocations, so capturing method calls with Actors quickly becomes unwieldy [2].

Formal automata models have been used to analyse protocols and channels. The properties of communication media are usually modelled as automata, too. For example, Nancy Lynch models communication media by processes in [29]. A lossy channel is modeled by a process that randomly drops messages. In contrast to these approaches, which apply ad-hoc techniques to model various kinds of links and networks, our modelling language fully integrates a set of primitives to describe dynamically evolving network topologies.

TinyOS [10] is a popular operating system for wireless sensor nodes. The TinyOS' programming language nesC [14] takes a similar approach as we do: Programs in nesC are structured in components. However, these components do not correspond to classes but rather to objects. In nesC tasks correspond to our processes and are cooperatively scheduled, because sensor nodes usually do not permit dynamic scheduling. In contrast, our approach abstracts from particular scheduling schemes; in fact, our models could be refined with specific schedulers. This may be a starting point for a development technique for applications which target TinyOS. We are currently investigating the relationship between our models and nesC programs in detail.

Ölveczky and Thorvaldsen [34] have shown how Real-Time Maude [33] can be applied to model and analyse advanced wireless sensor network algorithms, using, e.g., Monte Carlo simulations for performance evaluation for networks with up to 800 nodes. Our work complements this approach by emphasising sensor functionality and behavior. However, we intend to investigate how their techniques for simulation may apply in our setting.

KOOL [18] is another object-oriented language embedded into Maude. In [18] Hills and Rosu give a perspective of verification of KOOL programs in Maude. They introduce auto-boxing and a separation of local and global memory to facilitate

model checking. Their main motivation is to utilise Maude's verification facilities for KOOL.

KOOL features single inheritance, object creation, encapsulation, communication via message sends, and concurrency. KOOL objects communicate by means of their class signature whereas Creol objects communicate by means of designated interfaces. Compared to Creol KOOL is missing the concept of a cointerface and multiple inheritance.

The concurrency model of KOOL is based on threads introducing the need for mutual exclusion solved by locks. The concurrency model of Creol is based on active objects guaranteeing mutual exclusion by exclusive access to the processor.

8 Conclusion

The holistic approach advocated in this paper expresses itself in the use of one formalism for modelling all aspects. Multiple inheritance is used as a composition mechanism that integrates different functional aspects. Non-functional aspects are not yet represented in our approach, but especially memory aspects are addressed in part with the model checking experiments. Especially, the tail-call optimisation and the use of static analysis to simplify memory management are important aspects.

The tools described in this paper are currently used in the Credo project. Initial experience points to some short-comings: Interpretation and model checking in Maude exposes the runtime syntax with the program annotations to the modeller, which makes it hard to relate the executed model to the original model. This problem is aggravated by the lack of proper output routines in Maude, which makes it difficult to present the program state in a more Creol-like manner to the modeller.

The setup helps us to separate many concerns: The compiler implements all static checks and the resulting model is type safe. There is no need in the interpreter and model checker to validate the type correctness of the executed model. This simplifies the formulation of the operational semantics considerably.

Using Maude for model checking and adapting the interpreter for model checking has a certain overhead during model checking. We can only check quite small models with currently available computational resources. This makes the need for a direct instantiation of the model into the interpreter apparent, which allows faster and more efficient model checking. This may make some state space reduction techniques available, e.g., partial order reductions. Others benefit the concrete application, too, and will be provided as model transformations.

However, at the current stage of the development, our approach has two crucial advantages:

(i) The model checker is very close to the interpreter, which defines the operational semantics in Creol. It is therefore trivial to relate the counter examples found by the model checker to executions of the interpreter (the counter example trace is a sub-sequence to a trace of the interpreter) and also to the input program.

(ii) The similarity of the interpreter to the model checker makes it trivial to prove that the semantics encoded by both tools are very closely related. We need not concern ourselves with potential errors introduced in a separate compilation step.

Finally, we summarise the results of this research:

(i) Separating concerns into different tools simplifies the complexity of each individual tool considerably, which reduces the development time and increases the confidence into each tool.

(ii) Using rewriting logic and Maude enables us to prototype the modelling language at a very fast pace, allowing us to experiment with language features and analyse special-purpose extensions.

(iii) Maude provides powerful analysis methods with its model checking and search facilities. The close correspondence between the interpreter and the model checker reduces the development time and helps again in prototyping.

(iv) Using Maude for model checking allows us to add many extensions to the interpreter that help in reducing the state space with very little development cost. This enables us to evaluate each idea for its potential in a full-fledged implementation.

A lot of work is left for the future. Embedded applications are inherently timed applications, so a timing model is necessary. We hope that once a suitable model and language for time has been developed, the tools can be adapted.

An explicit memory model is needed. Memory models are not trivial, since they depend on the target platform. It is hopeless to estimate the memory usage without a precise description of the intended target platform and its properties. The "pointer size", the different sizes of integer data types, or the number of registers available on the CPU all influence the amount of memory needed by the application. This is usually not of concern for modern desktop computers, but for sensor devices, which are often equipped with only 10 KiB of memory (less than one fifth the size of this paper's LaTeX source), this decides on whether the model can be realised.

We envisage a tool chain that allows to develop very abstract models of a sensor network, perform analysis of these models and develop these with the help of tools that support formal methods into implementations that run on a real device. The relatively small memory of a device makes model checking of these devices feasible. Still, compositional methods are needed to reason about the global method, which also need to be developed.

Acknowledgement

The following people contributed to this research by their feedback, discussion, and contributions: Jasmin Blanchette, Immo Grabe, Einar Broch Johnsen, Wolfgang Leister, Andries Stam, and Bjarte Østvold.

References

[1] Ábrahám-Mumm, E., F. S. de Boer, W.-P. de Roever and M. Steffen, *Verification for Java's reentrant multithreading concept*, in: M. Nielsen and U. H. Engberg, editors, *FoSSaCS*, LNCS **2303** (2002), pp. 4–20.

[2] Agha, G. A., *Abstracting interaction patterns: A programming paradigm for open distributed systems*, in: E. Najm and J.-B. Stefani, editors, *FMOODS* (1996), pp. 135–153.

[3] Aho, A. V., R. Sethi and J. D. Ullman, "Compilers: Principles, Techniques, and Tools," Addison Wesley Publishing Company, 1986.

[4] Benton, N., L. Cardelli and C. Fournet, *Modern concurrency abstractions for C♯*, ACM Transactions on Programming Languages and Systems **26** (2004), pp. 769–804.

[5] Bonsangue, M. M. and E. B. Johnsen, editors, "Formal Methods for Open Object-Based Distributed Systems, 9th IFIP WG 6.1 International Conference, FMOODS 2007, Paphos, Cyprus, June 6–8, 2007, Proceedings," LNCS **4468**, Springer, Heidelberg, 2007.

[6] Cardelli, L., *An implementation of $F_{<:}$*, Technical Report 97, Digital Equipment Corporation, System Research Center (1993).

[7] Caromel, D., L. Henrio and B. Serpette, *Asynchronous and deterministic objects*, in: *POPL* (2004), pp. 123–134.

[8] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J. F. Quesada, *Maude: Specification and programming in rewriting logic*, Theoretical Computer Science **285** (2002), pp. 187–243.

[9] Compagnoni, A. B., *Higher-order subtyping and its decidability*, Information and Computation **191** (2004), pp. 41–113.

[10] Culler, D. E., J. L. Hill, P. Buonadonna, R. Szewczyk and A. Woo, *A network-centric approach to embedded software for tiny devices*, in: T. A. Henzinger and C. M. Kirsch, editors, *EMSOFT*, LNCS **2211** (2001), pp. 114–130.

[11] Dovland, J., E. B. Johnsen and O. Owe, *Verification of concurrent objects with asynchronous method calls*, in: *SwSTE* (2005), pp. 141–150.

[12] Eker, S., J. Meseguer and A. Sridharanarayanan, *The Maude LTL model checker*, in: F. Gadducci and U. Montanari, editors, *WRLA*, ENTCS **71** (2004), pp. 162–187.

[13] Flanagan, C. and S. Qadeer, *Transactions for software model checking*, ENTCS **89** (2003), pp. 518–539.

[14] Gay, D., P. Levis, J. R. von Behren, M. Welsh, E. A. Brewer and D. E. Culler, *The nesC language: A holistic approach to networked embedded systems*, in: *PLDI* (2003), pp. 1–11.

[15] Godefroid, P. and P. Wolper, *A partial approach to model checking*, Information and Computation **110** (1994), pp. 305–326.

[16] Graf, S. and J. Hooman, *Correct development of embedded systems*, in: F. Oquendo, B. Warboys and R. Morrison, editors, *EWSA*, LNCS **3047** (2004), pp. 241–249.

[17] Halstead, R. H., *MULTILISP: A language for concurrent symbolic computation*, ACM Transactions on Programming Languages and Systems **7** (1985), pp. 501–538.

[18] Hills, M. and G. Rosu, *On formal analysis of OO languages using rewriting logic: Designing for performance*, in: Bonsangue and Johnsen [5], pp. 107–121.

[19] Hoare, C. A. R., *Monitors: An operating system structuring concept*, CACM **17** (1974), pp. 549–557.

[20] International Telecommunication Union, Geneva, "Open Distributed Processing - Reference Model parts 1–4," (1995).

[21] Itzstein, G. S. and M. Jasiunas, *On implementing high level concurrency in Java*, in: A. Omondi and S. Sedukhin, editors, *ACSAC*, LNCS **2823** (2003), pp. 151–165.

[22] Jia, G. and S. Graf, *Verification experiments on the MASCARA protocol*, in: M. B. Dwyer, editor, *SPIN*, LNCS **2057** (2001), pp. 123–142.

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

[24] Johnsen, E. B., O. Owe and M. Arnestad, *Combining active and reactive behavior in concurrent objects*, in: D. Langmyhr, editor, *NIK* (2003), pp. 193–204.

[25] Johnsen, E. B., O. Owe and E. W. Axelsen, *A run-time environment for concurrent objects with asynchronous method calls*, in: N. Martí-Oliet, editor, *WRLA*, ENTCS **117** (2005), pp. 375–392.

[26] Johnsen, E. B., O. Owe and I. C. Yu, *Creol: A type-safe object-oriented model for distributed concurrent systems*, Theoretical Computer Science **365** (2006), pp. 23–66.

[27] Kurshan, R., V. Levin, M. Minea, D. Peled and H. Yenigün, *Static partial order reduction*, in: B. Steffen, editor, *TACAS*, LNCS **1384** (1998), pp. 345–357.

[28] Liu, H. and J. S. Moore, *Java program verification via a JVM deep embedding in ACL2*, in: K. Slind, A. Bunker and G. C. Gopalakrishnan, editors, *TP-HOLs*, LNCS **3223** (2004), pp. 184–200.

[29] Lynch, N. A., "Distributed Algorithms," The Morgan Kaufmann Series in Data Management Systems, Morgan Kaufmann Publishers, Inc., San Francisco, 1996.

[30] Meseguer, J., *Conditional rewriting logic as a unified model of concurrency*, Theoretical Computer Science **96** (1992), pp. 73–155.

[31] Meseguer, J. and G. Rosu, *Rewriting logic semantics: From language specifications to formal analysis tools*, in: D. A. Basin and M. Rusinowitch, editors, *IJCAR 2004*, LNCS **3097** (2004), pp. 1–44.

[32] Oliver, I., *Model driven embedded systems*, in: *ACSD* (2003), p. 5.

[33] Ölveczky, P. C. and J. Meseguer, *Specification of real-time and hybrid systems in rewriting logic*, Theoretical Computer Science **285** (2002), pp. 359–405.

[34] Ölveczky, P. C. and S. Thorvaldsen, *Formal modeling and analysis of the OGDC wireless sensor network algorithm in Real-Time Maude*, in: Bonsangue and Johnsen [5], pp. 122–140.

[35] Pierce, B. C., *Intersection types and bounded polymorphism*, Mathematical Structures in Computer Science **7** (1997), pp. 129–193.

[36] Pierce, B. C., "Types and Programming Languages," MIT Press, 2002.

[37] Pnueli, A., *The temporal logic of programs*, in: *FOCS* (1977), pp. 46–57.

[38] Raisinghani, V. T. and S. Iyer, *Cross-layer design optimizations in wireless protocol stacks*, Computer Communications **27** (2004), pp. 720–724.

[39] Schönborn, J. and M. Kyas, *A theory of bounded fair scheduling*, in: J. Fitzgerald, A. Haxthausen and H. Yenigün, editors, *ICTAC*, LNCS **5160** (2008), pp. 334–348.

[40] SysML Partners, "OMG SysML Specification," (2007), available for download at `http://www.sysml.org/specs.htm`.

[41] Yonezawa, A., "ABCL: An Object-Oriented Concurrent System," Series in Computer Systems, MIT, 1990.