



The United Nations
University

UNU/IIST

International Institute for
Software Technology

Exploiting Distribution and Atomic Transactions for Partial Order Reduction

Bernhard K. Aichernig, Andreas Griesmayer,
Marcel Kyas, and Rudolf Schlatte

June 17, 2009

UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macau, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the Governor of Macau and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macau authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macau or visit UNU-IIST's home page: <http://www.iist.unu.edu>, if you would like to know more about UNU-IIST and its report series.

Chris George, Acting Director



The United Nations
University

UNU/IIST

**International Institute for
Software Technology**

P.O. Box 3058
Macau

Exploiting Distribution and Atomic Transactions for Partial Order Reduction

Bernhard K. Aichernig, Andreas Griesmayer,
Marcel Kyas, and Rudolf Schlatte

Abstract

We present a method for systematic examination and reduction of the state space of distributed systems. The approach exploits properties of distributed systems to partition processes into *atomic transitions* and uses observations about partial orders among the distributed components to perform dynamic partial order reduction for state space reduction. This report concentrates on preserving reachability checks and deadlock detection for a fixed input of the system. We present a search algorithm with a succinct representation of the search stack that requires only the current state in memory. We furthermore define characteristics to determine independent blocks of processes and propose and evaluate heuristics to guide the search to further state space reduction. In combination with dynamic symbolic execution (also known as concolic execution), the approach also allows the generalization of a concrete run and gives rise to examination of the full input state space of a distributed system.

Andreas Griesmayer is post-doctoral fellow working on the project CREOL funded by the European Union (IST-33826).

Email: `agriesma@iist.unu.edu`

Bernhard Aichernig is adjunct research fellow at UNU-IIST and assistant professor at Graz University of Technology.

Email: `bka@iist.unu.edu`

Marcel Kyas is assistant professor for embedded systems in the working group Computer Systems & Telematics (CST) at the Institute of Computer Science, Freie Universität Berlin.

Email: `kyas@inf.fu-berlin.de`

Rudolf Schlatte is fellow at UNU-IIST and PhD student at Graz University of Technology, working on the project CREOL funded by the European Union (IST-33826).

Email: `rschlatte@iist.unu.edu`

Contents

1	Introduction	1
2	Preliminaries	2
2.1	Creol	2
2.2	Introduction to Partial Order Reduction	4
3	POR on Distributed Systems	5
3.1	Combination to Atomic Blocks	6
3.2	Dependency on Atomic Blocks	7
3.3	Dynamic Partial Order Reduction in Creol	8
3.3.1	Algorithm for State Space Exploration	9
3.3.2	Example	10
4	Implementation	11
5	Evaluation	13
5.1	example: Barrier in BSN	13
5.2	Heuristics for Dynamic POR	13
6	Related Work	15
7	Conclusion and Future Work	15

1 Introduction

The development of distributed systems is of increasing importance, while at the same time, the complexity makes it hard for a developer to maintain a full overview over the system.

Distributed systems are relatively loosely coupled components that communicate by exchanging messages. The components maintain local memory that is not directly accessible from outside. The local statements of a component are executed independently from each other — synchronization is performed via blocking message receptions if necessary. In this report we use Creol [8, 9] to analyze the properties of distributed systems. Creol is an executable modeling language that accurately models these properties. Creol’s semantic is defined in rewriting logic and implemented in the rewriting logic engine Maude [2], which directly gives an interpreter. Being a modeling language, Creol allows to leave local scheduling underspecified, which permits the developer to split concerns of functional properties and scheduling. Such formal models help to get a better insight over the behavior of distributed systems, but at the same time the introduced nondeterminism leads to state space explosion and thus makes automated verification hard: Previous experiments with the model checker of Maude required 10 minutes and 800 MB of RAM to check the dining philosophers example with 5 instances, 6 instances already exceeded the system’s 2GB of memory. Another obstacle in model checking Creol models, apart from the state space explosion, is the significant size of each state due to the presence of the underlying interpreter.

To cope with these problems, we analyze Creol to identify partial orders among the distributed components and develop an alternative execution model that gives rise to efficient partial order reduction techniques to reduce the state space that needs to be searched. In this report we concentrate on preserving (local) reachability properties and deadlock detection, which are of great interest in the development of distributed system. We mainly aim at systems that are cycle free, like finite systems or reactive systems that are instrumented with a use case of interest. This property allows a search algorithm that does not need to store the searched state space. A typical application for the approach is checking of all possible schedulings for test cases that up to now were only examined by a single execution of the model. For these systems, we perform concrete executions of the model with the goal to examine as few interleavings as possible while still ensuring the aforementioned properties. Because input nondeterminism is not handled, the presented methods do not perform full model checking, but an intermediate step between single executions and model checking. However, this approach can be easily combined with work on dynamic symbolic execution, an approach that generalizes a concrete run and gives rise to examination of the full input space [6].

Distributed systems and the Creol modeling language are discussed in more detail in Section 2.1, and a short overview over existing partial order reduction techniques is given in Section 2.2. Using the observations from these two sections, we introduce an alternate execution model for Creol in Section 3.1. This model significantly reduces the possible interleavings of statements of different components, while retaining the properties of interest from the original model. It also sets the stage for further optimizations that are based on observations on the behavior of

distributed systems. Section 3.2 deals with the impact of messages (the underlying technique for Creol’s method calls) on the partial orders, while Section 3.3 presents an approach that tracks and utilizes dependencies between objects during runtime.

The approach implements model checking techniques in the rewriting logic of Maude [2] without using Maude’s built-in search command or model checker. In this way the language semantics of Creol can be exploited better than with Maude’s generic tools. A state of Creol includes representation of all variables in the objects and processes and is therefore complex. Storing the full state for each execution step, which Maude’s built-in tools have to do, is therefore not feasible. By explicitly resolving all nondeterminism, and introducing a unique naming of objects and processes, keeping the full state space in memory is avoided. Instead, only the current state, and a trace of executed process IDs is recorded. Backtracking can be performed by replaying the execution log. We discuss these implementation details in Section 4.

2 Preliminaries

2.1 Creol

In this section we summarize the syntax and operational semantics of Creol. Creol’s semantics has been formalized in rewriting logic [10] and forms the basis of a Creol interpreter. A more complete description of the language can be found in [8] and [9]. We refer the reader to these papers for details.

Conceptually, an object in Creol is active and executes on its own (virtual) processor. Object activity results from method calls. Active behavior is initiated by the special **run** method, which is called immediately after object creation, and interleaved with reactive behavior by means of *processor release points*. Reactive behavior is caused by external method calls. Method calls are *asynchronous*, where the result may be obtained with the help of a *future variable* [15]. Calls can always be emitted, because objects cannot block communication. Overtaking of calls is permitted: if methods offered by an object are invoked in one order, the object may start execution of the resulting processes in a different order.

An object may contain several processes, of which at most one (the “current process”) can be *active*; the other processes are either *waiting* or *suspended*. A Creol process always executes within its object; method calls to another object cause the creation of another process within the called object. The concrete syntax of Creol programs is displayed in Fig. 1.

We assume a *functional first-order language* of expressions e without side effects, which includes an equality function on object identities. Object-oriented features extend the functional language. Class definitions include declarations of persistent state variables and method definitions. Calls are *bound late*.

$$\begin{aligned}
If &::= \text{interface } I [\text{inherits } I \{, I\}] \text{ begin } \{MDecl\} \text{ end} \\
Cl &::= \text{class } C[(Vdecl \{, Vdecl\})] [\text{inherits } C[(\vec{e})] \{, C[(\vec{e})\}]] \\
&\quad [\text{implements } I \{, I\}] \text{ begin } \{\text{var } Vdecl\} \{Meth\} \text{ end} \\
MDecl &::= [\text{with } Type] \text{op } m [[(\text{in } Vdecl \{, Vdecl\})[[;]] \\
&\quad \text{out } Vdecl \{, Vdecl\}]] \\
Vdecl &::= v : Type \\
Meth &::= MDecl == \{\text{var } Vdecl; \} Stmt \\
Stmt &::= \text{skip} \mid v := e \mid \ell.o.m(\vec{e}) \mid \ell?(\vec{v}) \mid \text{await } c \mid \text{release} \mid \\
&\quad \text{if } b \text{ then } Stmt \text{ else } Stmt \text{ fi} \mid Stmt; Stmt \mid Stmt \square Stmt
\end{aligned}$$

Figure 1: The syntax of the Creol kernel language. The symbol $\{\dots\}$ represent repetition of the enclosed production. The symbol $[\dots]$ represents that the enclosed production is optional. The symbol \vec{e} represents a comma-separated list of expressions. The symbol \vec{v} represents a comma-separated list of variable names.

The semantics of the Creol language are defined by rewriting logic. A rewriting logic consists of an equational theory, that is described by a confluent and terminating term substitution system, and rewrite rules, that again define term substitutions. A conditional rewrite matches a sub-term in a term that has been normalized using the equational theory, replaces the sub-term by a new sub-term and computes the normal form of the result wrt. the equational theory. The rewrite rules neither need to be confluent nor terminating.

A Creol system state is described by a *configuration*, which is a multi-set of objects, classes, and messages. As customary in rewriting logic, multi-sets are constructed by juxtaposition. Concurrency of objects is modeled by concurrent rewrites of non-overlapping left hand sides. A rule of the form

$$subconfiguration \rightarrow subconfiguration' \text{ if } condition$$

expresses a conditional one-step rewrite. Whenever *subconfiguration*, i.e. some multi-set of objects, classes, or messages that is a part of the configuration, matches and *condition* holds for the match, then *subconfiguration* is replaced by *subconfiguration'*. Each rule contains at most one object in the left hand side, ensuring that the objects are running independently.

A Creol object representation in Maude has the general form

$$\langle O : C \mid Att : \bar{a}, Pr : p, PrQ : q \rangle$$

where O is the identity, C the name of its class, a the state (valuation) of its attributes, p the current process (which may be the *idle* process) and q the multi-set of waiting or suspended processes. A process has the form $\{\bar{l} \mid \bar{s}\}$ where \bar{l} is the state (valuation) of the local variables and \bar{s} is a statement list.

Processes are executed in *run-to-completion* steps, i.e., until the process has terminated *or* control

is given up *explicitly*. Hence, there is no preemptive scheduling. The statement **release** releases control unconditionally, whereas the statement **await** b releases control until the Boolean expression b evaluates to true. The current process is only suspended if b evaluates to false. A process may wait for a labeled completion messages with, e.g., **await** $\ell?$. Again, control is released only if the completion message has not yet arrived.

Assignment statements allow multiple assignments. Assignments are evaluated in two stages: first all the values to be assigned are computed, and then the actual change to the state is committed. Additionally, Creol provides sequential composition of statements, a conditional statement, and a nondeterministic choice operator **[]**. Program execution can continue with any enabled branch of the choice statement.

We assume functions *enabled* and *ready* that define whether a statement in a process is enabled or whether a suspended process is ready for execution.

The primitives for asynchronous method calls have the form $\ell!o.m(\bar{e})$ and $\ell?(v)$. The handle ℓ of a call plays the role of the future variable: It allows the caller to receive the return values to a call $\ell!o.m(\bar{e})$ later in the computation using the statement $\ell?(v)$. The effect of the later statement is to store the result of the call in the variables \bar{v} .

2.2 Introduction to Partial Order Reduction

The execution of parallel programs involves concurrent execution of statements in separate processes. In the absence of synchronization, the actual order of execution is not defined by the program and can be performed arbitrarily, which is one of the major reasons for state space explosion in model checking concurrent software. Consider the process P_0 with $p_{01} : x = 1; p_{02} : z = 5$; and P_1 with $p_{11} : y = 3; p_{12} : z = 4$. Concurrent execution of this two processes gives rise to a number of interleavings that may lead to different result states. For instance $p_{01}; p_{11}; p_{02}; p_{12}$ results in a state with $x = 1, y = 3$, and $z = 4$. Not all possible sequences lead to a different final state though. E.g., in the previous run, it does not matter whether p_{01} or p_{11} is executed first, but the order of execution of p_{02} and p_{12} *does* change the result. The result of the execution therefore depends on the relative order of some of the statements, a *partial order*. We call pairs of statements whose order influences the result of an execution *dependent*; conversely, reordering *independent* statements has no influence on the result. The technique of partial order reduction aims to search only relevant interleavings by identifying and exploiting the knowledge about independent statements.

Partial order reduction (POR) is a vivid area of research with a number of applications adapted to different purposes [12]. There are basically three different groups of approaches: *sleep set* POR [5] uses the enabled statements of the current state and observations from previous search to avoid transitions to states that have been visited before. This limits the transitions that have to be searched, but still the full state space is explored. A second group of approaches (*persistent set* [5], *stubborn set* [13] and *ample set* [1] POR) delay independent transitions to reduce the

number of states, but need information about the static structure of the program to ensure that no states are skipped that would spawn further processes. A third approach is *dynamic* partial order reduction [3], which is similar to *persistent set* POR, but gathers information about transitions to search on the fly. These different groups of POR are independent from each other and can be combined. In the following we will use the experiences from those methods to make POR suitable for the concurrency model of Creol, which is considerably different from other languages and thus requires some adaptations.

We use standard terms to define our POR method. A *state* s is defined by the active objects, the values of their object variables and the processes in their process queues, a *transition* is the atomic change from one state to the other. The set of transitions that are ready for execution in a state is given by $enabled(s)$. The state that is reached by executing transition α in state s is denoted by $\alpha(s)$.

Definition 1 (independent). *two enabled transitions α and β are independent, if $\alpha \in enabled(\beta(s))$ and $\beta \in enabled(\alpha(s))$ and $\alpha(\beta(s)) = \beta(\alpha(s))$. That is, if the execution of one of the transitions does not disable the other one, and execution of the transitions in any order results in the same state. For transitions that are never enabled at the same time, the question of dependency does not arise. We therefore treat them as trivially independent.*

Independent transitions can be executed in any order without changing the result of the execution [1].

3 POR on Distributed Systems

Distributed systems are a special form of concurrent systems and therefore inherently affected by state space explosion due to interleaving. This is reflected by the standard semantic rules of Creol that directly act as interpreter of the language and allow any order of execution of statements in concurrently active objects. This correctly models distributed systems, which do not allow any synchronization between statements from different objects (objects can be synchronized indirectly via messages), but makes it hard to perform automated verification on such systems. Because statements from different objects cannot interfere with each other, the results from POR suggest that the state space can be greatly reduced by removing equivalent runs. Reducing the interleavings on the statement-level is therefore the first step of the POR approach for distributed systems, and is done by partitioning processes into *atomic blocks* (AB).

This reduction also gives rise to further considerations on which interleavings of atomic blocks to examine. As distributed systems usually are only loosely coupled, it seems likely that not all interleavings of atomic blocks from different objects have to be considered. In the remainder of this section, we will introduce an alternative execution model of Creol that is based on atomic blocks, and show methods to further reduce the interleavings by considering the dependencies between distributed objects. We concentrate on deadlock detection and object-local assertions

in finite traces to present an efficient algorithm to search the state space and show for both cases that the presented reductions preserve those properties.

3.1 Combination to Atomic Blocks

We change the execution model from switching between the execution of objects after each statement, to allow switching only when a process gives up control. The sequence of statements between process switches is called an *atomic block* (AB). Such an AB is identified dynamically during execution. It starts when a process is activated and contains all statements until it terminates or gives up control explicitly as explained in Section 2.1.¹ Only when a process gives up control (an AB finished execution), a new AB is started. A method call corresponds to creating an AB in the callee-object, message reception ($\ell?(\bar{v})$) releases the processor and *disables* all ABs in the caller-object until the callee-AB returns. We use this model to reduce the object switches and as simplified view of the execution. The resulting local states of the objects, and therefore the enabledness of the processes, remain unchanged as shown by the following theorem.

Theorem 1 (Preservation of local state). *Any run in Creol has an equivalent run in the AB based execution model that leads to the same local state.*

Proof. Because the statements from different objects operate on separated memory, they are independent and therefore can be switched. Together with the observation that an object executes only one process at a time, it is easy to see that every run of non-communicating ABs in Creol can be reordered such that all statements of an AB are executed in sequence without interleaving by other statements.

It remains to be shown that the execution model is fine enough, i.e., that all necessary interleavings of ABs are considered. The only possibility for another object to change the local state of a running process is by message reception $\ell?(\bar{v})$, possibly with a preceding **await** $\ell?$. In case the current AB started the method call related to ℓ , the respective message can not have been executed yet, and either the **await** releases the processor or $\ell?(\bar{v})$ blocks. Both cases mark the end of an AB and allow for scheduling of other objects. In case the respective method call was issued before execution of AB, the respective method can be executed before AB. Both cases allow for an arbitrary interleaving of enabled atomic blocks as in standard Creol. \square

Note that the previous theorem also implies that **assert** statements and deadlock properties remain unchanged.

For easier representation of the system, we define a mapping from the Creol program to a system of atomic blocks. A state s contains a set of atomic blocks $\mathcal{AB}(s)$ and a valuation of the object variables σ , where σ is partitioned between the active objects. The objects are not explicitly

¹Note that a **[]** operator creates two new AB and selects one of them for execution. This is done to allow the implementation to “replay” the trace instead of explicitly storing the state trace.

mentioned in the state, but accessible by $ab.o$ for any $ab \in \mathcal{AB}(s)$. For brevity, we will use capital letters to denote atomic blocks from a certain object and indices to give a unique name, e.g., A_1, A_2, B_1 for atomic blocks with $A_1.o = A_2.o \neq B_1.o$. We will also write \mathcal{AB} for the set of (possibly infinitely many) atomic blocks in a program.

For an $ab \in \mathcal{AB}$ we write $ab.enabled$ to check if an atomic block is ready to execute, $s.enabled$ is the set of all ABs that are enabled in state s . Executing ab on valuation σ with the resulting valuation σ_1 is denoted as $\sigma_1 = ab(\sigma)$. The set $ab.post$ contains the ABs that are created as a consequence of the execution of ab . For $ab_i \in ab_j.post$ we also write $ab_j \hookrightarrow ab_i$. Execution of an atomic block causes a transition $s \xrightarrow{ab} s'$ in the system with

$$\begin{aligned} ab \in s.enabled \wedge ab \notin \mathcal{AB}(s') \\ \mathcal{AB}(s') \setminus \mathcal{AB}(s) = ab.post \\ s'.\sigma = ab(s.\sigma) \end{aligned}$$

Intuitively, the executed atomic block must be ready for execution in s and is replaced by the atomic blocks it creates. The variable assignment of the object is changed by execution of ab .

Because ABs of the same object work on the same state, they are possibly dependent on each other. Although atomic blocks on different objects are independent, the execution can create new ABs that are dependent. Thus, executions of ABs can not per se be exchanged. In the following, we will show approaches how to find out which interleavings of AB executions have to be checked.

3.2 Dependency on Atomic Blocks

The execution model as introduced above avoids the interleaving of statements of different processes, but still has to examine all permutations of atomic blocks. For further reduction of the state space, we define a dependency relation among atomic blocks. The dependency relation has to consider two aspects: state change, and creation of further blocks. If we only consider access to the state, two ABs from different objects can always be executed in any order. Because of method calls, however, this is not the case in general. Consider two atomic blocks A_1 and B_1 with $B_1.post = \{A_2\}$. Although A_1 and B_1 are independent, executing B_1 first allows a trace $s_0 \xrightarrow{B_1} s_1 \xrightarrow{A_2} s_2 \xrightarrow{A_1} s_3$, which is not equivalent to $s_0 \xrightarrow{A_1} s'_1 \xrightarrow{B_2} s'_2 \xrightarrow{A_2} s'_3$. Thus, both interleavings have to be checked. For state space reduction we identify cases where atomic blocks can be reordered although messages are sent:

Theorem 2 (push up AB creation). *For each run $\pi = s_0 \dots s_i \dots \xrightarrow{ab_j} s_j \dots \xrightarrow{ab_k} s_k$ with $ab_j \in s_i.enabled$, $ab_j \hookrightarrow ab_k$ and ab_j does not access the state, there is an equivalent run $\pi' = s_0 \dots s_i \xrightarrow{ab_j} \dots \xrightarrow{ab_{j-1}} s_j \xrightarrow{ab_{j+1}} \dots \xrightarrow{ab_k} s_k$.*

Proof. Because ab_j does not access (neither by reading nor by writing) the local object state, we have $s_{j-1}.\sigma = s_j.\sigma$ in π , and removing ab_j from the trace does not change the enabledness of the ABs up to ab_k . Similarly, executing the transition at an earlier state of the run creates ab_k earlier, but does not change the valuation of the variables and thus the enabledness of further ABs. \square

Such atomic blocks that do not access the local state are used in distributed systems for synchronization code like *barriers*. The previous theorem shows that in such cases the AB can be executed immediately when they are enabled the first time without having to examine another interleaving. Note that, if several such ABs are enabled in a state, the order among them can be chosen arbitrarily. Note furthermore that such blocks, although not accessing the object state, are not trivial as their enabledness can depend on the state, and the behavior can depend on call arguments.

Theorem 3 (push up AB with exclusive control). *For each run $\pi = s_0 \dots \xrightarrow{ab_i} s_i \dots \xrightarrow{ab_j} s_j \dots s_k$ with $ab_i \hookrightarrow ab_j$ and ab_j has exclusive control on the object, there is an equivalent run $\pi' = s_0 \dots \xrightarrow{ab_i} s_i \xrightarrow{ab_j} \dots \xrightarrow{ab_{j-1}} s'_j \xrightarrow{ab_{j+1}} \dots s_k$.*

Proof. If ab_j has exclusive control on the processor, there is no atomic block ab_l with $i < l < j$ and $ab_j.o = ab_l.o$, thus, all ab_l and ab_j are independent and for each $s_{j-2} \xrightarrow{ab_{j-1}} s_{j-1} \xrightarrow{ab_j} s_j$ there is $s_{j-2} \xrightarrow{ab_j} s'_{j-1} \xrightarrow{ab_{j-1}} s_j$. (Also if ab_j is enabled in s_{j-1} it is also enabled in s_l with $i < l < j$ as only dependent ABs could disable it.) Subsequent exchanges of ab_j with its predecessor will therefore create π' . \square

Atomic blocks that exclusively hold the processor of an object, like object initialization and blocking message reception, can therefore be executed immediately when they are enabled.

3.3 Dynamic Partial Order Reduction in Creol

The previous sections show how to reduce the interleavings by identifying and fixing an ordering for independent statements and atomic blocks in the system. Although the introduction of atomic blocks considerably reduces the number of interleavings, the system remains highly nondeterministic. Checking dependency among atomic blocks for the general case is in practice infeasible as, to find out if two atomic blocks A_1 and B_1 are dependent, one has to check if B_1 will in the future run create an atomic block in A or vice versa. This corresponds to a reachability problem that has to be solved for each step in the execution. To solve this problem, partial order reduction techniques like in [1] exploit static information about the model to generate heuristics like marking all statement from two processes dependent that during their run have any dependent statements. In our setting this seems too restrictive though.

```

1  exectrace : Stack          18  search()
2  backtrack : Stack        19    current = s0
3  select : AB              20    backtrack.push(s0, selectNext())
4  current, dependent : State 21    while backtrack not empty
5                               22      (current, select) = backtrack.pop(),
6  explore():                23      adjust exectrace
7    dependent =              24      explore()
8      selectLastDependent() 25
9    if dependent != null    26  addBacktrack()
10     addBacktrack()        27    ancestor =  $\mathcal{AB}(\text{dependent}) \cap \text{select.pre}$ 
11    if select.executable   28    if ancestor in dependent.done
12     current = select(current) 29      return
13     exectrace.push(current) 30    if ancestor.enabled
14    add select to current.done 31      backtrack.push(dependent, ancestor)
15    select = selectNext()  32    else
16    if select                33       $\forall ab \in (\mathcal{AB}(\text{dependent}) - \text{dependent.done})$ 
17     explore()              34      push(dependent, ab)

```

Figure 2: Algorithm for dynamic partial order reduction

Flanagan et al. show in [3] a method of *dynamic* partial order reduction with the idea of checking the inter-dependencies between processes at runtime. The approach maintains an execution stack and after each transition, the dependencies of currently enabled transitions and the execution stack is checked. If a dependent transition is found on the stack, the respective state is marked for backtracking to examine a different scheduling later on. This can be seen as performing the reachability check to find out if two ABs are dependent, and use the check already for examination of the model.

To adopt the approach to the Creol setting, we introduce a unique, ordered identifier to each object in order of their creation. Atomic blocks have a two-part identifier consisting of the object ID and an incrementing number within the objects. Thus, we write A_1, A_2, \dots, A_n for the ABs of the first created object and B_1, B_2, \dots, B_m for the second object and so on. We build the transitive closure of the “created by” relation $A \hookrightarrow B$ and denote with the set $pre(A) = \{B \mid B \hookrightarrow A\} \cup \{A\}$ the antecedents of A . As discussed above, A is independent from B for $A.o \neq B.o$. To simplify the discussion of the approach, we assume that two ABs of the same object are dependent.

3.3.1 Algorithm for State Space Exploration

Dynamic partial order reduction examines the state space in a depth first search manner, with the modification that the states to backtrack to are determined dynamically during the search. The algorithm is outlined in Fig. 2. The `search()` routine in Line 18 sets up the search by pushing the initial state and the first AB to execute on the `backtrack` stack and starts `explore()` on all states until all significant interleavings are searched. The function `selectNext()` selects an

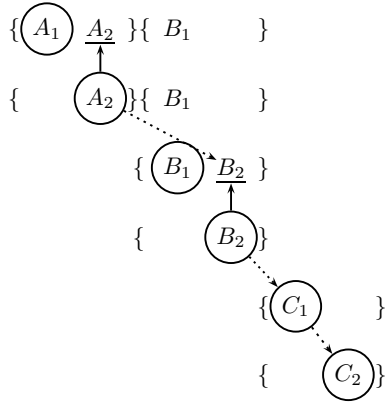


Figure 3: Initial run

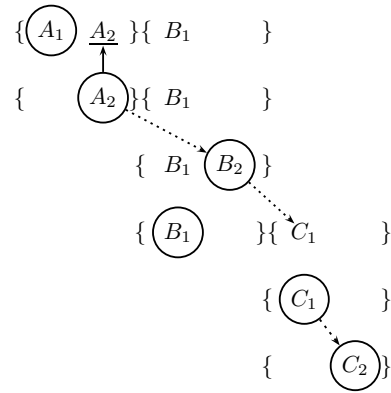


Figure 4: First Backtracking

AB among the ABs of the state that were not examined yet. For now, we select the “smallest” one, we will show heuristics to further reduce the state space in Section 5.2.

Function `explore()` in Line 6 performs the actual execution of a run and adds backtracking points to `backtrack`. Such a point is added if there is a state where an dependent AB was executed in the current run (a dependent AB is from the same object and not an ancestor of the selected AB). If such a state exists, `addBacktrack()` selects the *closest* (in terms of the search tree) state-AB combination that can lead to a reordering of the dependent ABs.

Correctness (outline). If enabledness is not an issue, it is easy to see that the algorithm is correct: Backtracking is only necessary when the execution contains a transition that depends on the selected transition. The way to reorder the ABs is to execute the ancestor (or the selected AB itself) instead of the dependent AB. Because the last backtracking points are revisited first, it is ensured that if an AB is in done, all of the relevant interleavings with the prefix up to that AB are searched. (If all ABs are dependent, a full state search is performed)

If the ancestor is not enabled, the algorithm ensures that all interleavings that could enable the ancestor are searched by pushing all ABs that are not searched before in Line 34. Note that this step also pushes the disabled ancestor to `backtrack`, which ensures that interleavings that enable ancestor in a state before the state dependent are searched too. This is because an AB can only be disabled by an dependent AB, which will be put into `backtrack` when the execution step with (dependent, ancestor) is executed.

3.3.2 Example

Fig. 3 to 6 show the search of the state space of three objects A , B , and C with $A_2 \leftrightarrow B_2 \leftrightarrow C_1 \leftrightarrow C_2$, which allows 30 different interleavings. Initially, only ABs A_1 , A_2 and B_1 are present and A_1 is selected for execution (marked by the circle). Its execution does not create any new ABs, so A_2 is selected for execution next. The execution stack is searched for dependent ABs

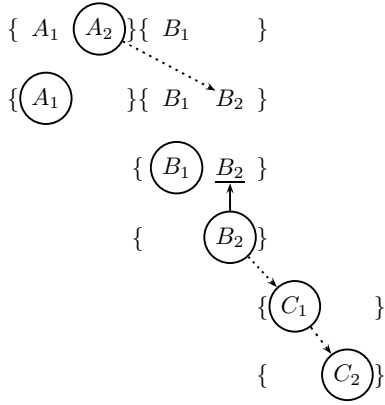


Figure 5: Second Backtracking

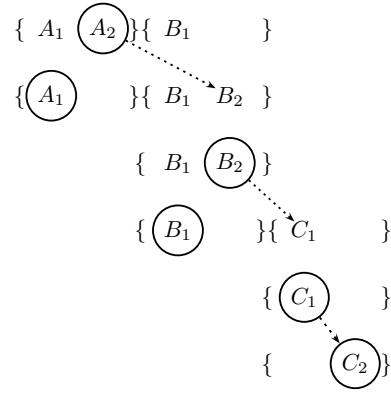


Figure 6: Third Backtracking

and A_1 is found in state 1. To search all necessary interleavings, A_2 is marked in the same state (marked by the underline and solid arrow). A_2 also creates the AB B_2 (dotted line), so the resulting state contains the active states B_1 and B_2 . After executing B_1 , the next execution step for B_2 finds a dependent AB B_1 in the execution trace and marks the corresponding state as another backtracking point. ABs C_1 and C_2 are executed without adding new backtrack states. Note that, although C_2 and C_1 depend on each other, no backtracking point is added because C_1 creates C_2 and therefore has to be executed first. The resulting first execution results in $s_0 \xrightarrow{A_1} s_1 \xrightarrow{A_2} s_2 \xrightarrow{B_1} s_3 \xrightarrow{B_2} s_4 \xrightarrow{C_1} s_5 \xrightarrow{C_2} s_6$. We return to the last backtracking point and execute B_2 in state s_2 . The corresponding execution is shown in Fig. 4 and results in the second interleaving $s_0 \xrightarrow{A_1} s_1 \xrightarrow{A_2} s_2 \xrightarrow{B_2} s'_3 \xrightarrow{B_1} s'_4 \xrightarrow{C_1} s'_5 \xrightarrow{C_2} s'_6$. The execution does not create a new backtrack point (executing B_1 in state 4 would create a backtracking point in state 3, but this was already executed before.) After backtracking to the first state, we get two further interleavings: $s_0 \xrightarrow{A_2} s'_1 \xrightarrow{A_1} s'_2 \xrightarrow{B_1} s''_3 \xrightarrow{B_2} s''_4 \xrightarrow{C_1} s''_5 \xrightarrow{C_2} s''_6$ and $s_0 \xrightarrow{A_2} s'_1 \xrightarrow{A_1} s'_2 \xrightarrow{B_2} s'''_3 \xrightarrow{B_1} s'''_4 \xrightarrow{C_1} s'''_5 \xrightarrow{C_2} s'''_6$. Note that all of these interleavings are necessary as they result in different states if the ABs in the object are truly dependent. There is no execution that starts with B_1 , as the ABs in object A are not dependent on it and safely can be executed first. Also, ABs like A_1 in Fig. 5 and 6 (and B_1 in Fig. 6) could be postponed to any point after those traces.

4 Implementation

Implementation of model checking of Creol programs faces a number of problems. Both the number of states and the size of a single state are considerable. We describe an implementation of the approach that builds upon the current implementation of the runtime interpreter and therefore adheres to the semantic of Creol, while making adaptations to further development of Creol as easy as possible. Furthermore, the code for searching the state space is fit into the semantics, while only the current state is required to be held in memory.

The Maude representation of an object itself and the rewrite rules governing statement execution

```

1 class OnetimeBarrier(total : Int)
2   contracts Barrier
3 begin
4   var arrived : Int := 0
5   with Any
6     op wait ==
7     arrived := arrived + 1;
8     await arrived = total
9 end

```

Figure 7: Simple *barrier* for synchronization of concurrent objects

are left unchanged from the original interpreter. The augmented interpreter implementing the AB execution model adds rules for selecting the next process to execute, which is non-deterministic in the original interpreter. In the augmented interpreter, a process only starts to execute if all objects in the system are idle. In the following, we will express this as selecting an enabled AB and executing its transition. To enforce this behavior in the augmented interpreter, a control configuration is added to the rewrite rules that select a next process

```
< Control | Mode: M Trace: TR>
```

where M is either `select` or `execute`. We start with `select` and switch to `execute` when an AB to execute was identified. After finishing execution of the AB and generating the followup ABs, we switch to `select` again. The view of executing Creol based on atomic blocks is compatible with the original interpreter as ABs are only a different representation for the processes in the process queue of a Creol object. Thus, this execution model reduces the state space without changing the language semantics of Creol.

Besides the vast number of interleavings, model checking Creol so far also suffers from the size of a state that has to be kept in memory by the built in model checker of Maude (resp. the search command). By introducing a unique naming of objects and atomic blocks, and restriction to finite traces, we can circumvent that problem by storing only a sequence of names instead of the full states. This naming is done by adding a unique identifier for each process in the process queue of an object. The execution trace TR is used for determining the dynamic POR as well as “replaying” the trace up to a selected state for backtracking. The field TR is a list of tuples of the form `< Sel: AB EnAB: EAB DiAB: DIAB Done: DAB Backtrack: BAB >` with AB the identifier of the executed AB, EAB (DIAB) the set of enabled (disabled) processes, DAB the set of ABs that already where examined by the POR search algorithm, and BAB the ABs that are selected for future backtracking.

The full implementation is still work in progress. To show the potential of the presented approach, a program was implemented to model a AB structure and perform the POR approach. As the enabledness of an AB depends on the underlying Creol program, the approaches from subsection 3.2 where not taken into account by the program, but are shown in an example that was evaluated manually.

5 Evaluation

To evaluate the approach, we first show an example for the reduction of interleavings by determining independent atomic blocks. Then, in Section 5.2 we show the reduction of interleavings for AB models and heuristics that were developed while examining the approach.

5.1 example: Barrier in BSN

Barriers are used to avoid unwanted interference between parallel objects, an effect that is not recognized by the model checker who in general has to check all possible interleavings of acquire and release.

An example is taken from a model of sensor networks from the CREDO project. This model describes a network between sensor nodes that generate ad-hoc connections to distribute data among multiple nodes towards a sink node. In the model, five sensors are initialized and the network is set up. A barrier is used to keep the sensors from sending before the network is ready.

The barrier (Fig. 7) is initialized with a threshold of 6, its `wait` method is called by the 5 sensor nodes and the network as the last step of their `init` methods. After all objects call the function, the block is released for all of them at the same time. The atomic block that controls the synchronization starts at the `await` statement in line 8 and consists of only that line. For 5 sensors, this leads to 120 different permutations, which can be reduced to only one because the `init` method has exclusive control over the processor in its object. The barrier receives 5 method calls that increment the `arrived` field. Because this AB accesses the local state, these 120 permutations can not be reduced. Finally, after setting up the network, the main method calls the 6th and final `wait`, which leads to the barrier releasing all six blocked ABs, and thus to 720 possible interleavings of the message returns. Because the ABs only create further atomic blocks (the processes waiting for the message returns in sensor and main method), they are identified as independent and only one of the interleavings is selected. The same is true for execution of the newly created ABs on the receiving end, which again can be reduced from 720 interleavings to one. The total number of different execution traces for this simple example is therefore $120 * 720 * 720 * 720 = 44789760000$, which can be reduced to only 120. Barrier-type synchronization mechanisms are not uncommon for modeling distributed systems, which shows the necessity of partial order reduction for checking distributed systems.

5.2 Heuristics for Dynamic POR

The algorithm as explained above chooses always the least AB in a self defined ordering, this leads to good results, but ignores cases in which whole execution traces should be reordered before a dependent AB. E.g. for two ABs A_1 and B_1 and $B_1 \hookrightarrow B_2 \hookrightarrow B_3 \hookrightarrow A_2$, the standard algorithm first creates $s_0 \xrightarrow{A_1} s_1 \xrightarrow{B_1} s_2 \xrightarrow{B_2} s_3 \xrightarrow{B_3} s_4 \xrightarrow{A_2} s_5$ and then backtracks to s_0 to create

	full	DPOR	nHeur	onHeur
creation(4)	70	1	1	1
reportback(8)	48620	10	3	3
reportback(15)	-	17	3	3
concur&reportback(3)	72072	60381	207	49
concur&reportback(4)	-	-	417	84

Table 1: Results from Partial order reduction giving the full number of paths and DPOR (and heuristics)

$s_0 \xrightarrow{B_1} s'_1 \xrightarrow{A_1} s_2 \xrightarrow{B_2} s_3 \xrightarrow{B_3} s_4 \xrightarrow{A_2} s_5$, leading to the same final state (and backtracks to s'_1). Only after several iterations a trace with A_2 before A_1 is reached. In this case, DPOR does not lead a reduction of interleavings. Therefore we use a heuristic that chooses ABs to execute in order of their creation, with the newest first (nHeur). This connects atomic blocks of the same execution trace. Using nHeur in the above example leads to only two interleavings, which is the optimum for that case. In general, we expect loosely connected systems to profit most from the POR algorithm. That is, objects that spawn further objects that after some time report back as in A_1 with $A_1 \hookrightarrow A_2 \hookrightarrow A_3$, $A_1 \hookrightarrow B_1 \hookrightarrow B_2 \hookrightarrow B_3 \hookrightarrow A_4$, and $A_1 \hookrightarrow C_1 \hookrightarrow C_2 \hookrightarrow C_3 \hookrightarrow A_5$. Intuitively, that corresponds to an object A that creates two objects B, C and executes some local ABs. The generated objects at some point report back. In total, this system allows 3150 interleavings, that are reduced to 2555 by the standard dynamic POR without any heuristic and to 87 with nHeur. Analyzing the resulting interleavings leads to a refinement of the heuristic: Because B and C report backs to A and A executes some local ABs, it is desirable to create these A_5 and A_6 as soon as possible to allow for the respective interleavings.

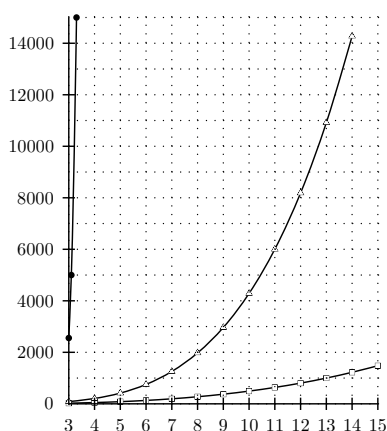


Figure 8: DPOR for increasing length of call sequences

A revised heuristic onHeur therefore also considers the object when selecting an AB to execute. This leads to a reduction to 25 interleavings in the example above. Table 1 shows some results of our experiments, where creation corresponds to creating two new objects without reporting back, reportback creates new objects with return and concur&reportback creates new objects and executes local ABs. The number in braces gives the number of ABs executed per object. Fig. 8 gives the numbers of examined interleavings for concur&callback and up to 15 ABs per object for nHeur (\square) and onHeur (\triangle). This example represents an worst case for the DPOR without heuristic, which cannot handle that number of ABs, which shows the value of the heuristics.

6 Related Work

Two of the most successful model checking tools that allow checking concurrent systems are the Java PathFinder (JPF, [14]²) and SPIN [7]. While JPF uses virtual machine to execute and model check Java byte code, SPIN uses Promela, a modeling language that closely resembles automata that communicate via channels. Both tools use partial order reduction similar to *ample set* construction [1] using static information about the program. This approach marks all statements from processes dependent, that can execute dependent statements in the further run. Besides the effort to compute these dependencies, in our setting, this would mean a much worse reduction as can be seen in the examples. In contrast to these approaches, [3] computes these dependencies during runtime and avoids to store the visited states. Our dynamic POR is motivated by this work and extends it by adaption to the distributed setting and heuristics for further reduction of the state space to search. Another example for an implementation of DPOR is [11] in a very similar setting to ours: The authors work on programs with separated memory that communicate via the *message passing interface* (MPI). Instead of initializing processes in distributed objects like in Creol, the MPI interface is used to pass messages between existing processes.

In [4], Flanagan et al. show a method how to combine statements in C-like concurrent programs to transactions, which is done by our approach dynamically. The properties of Creol make this process considerably easier.

7 Conclusion and Future Work

We presented an approach to significantly reduce the state space that has to be searched in order to prove deadlock and safety properties in distributed systems modeled in Creol. Based on the determination of *atomic blocks*, the approach defines independence criteria and a dynamic partial order algorithm to reduce the interleavings of concurrent processes. The approach can be implemented using the Maude rewriting system and therefore strictly adheres to the semantics of Creol, while storing the full state space is avoided.

The current approach checks the different interleavings caused by local scheduling and inter-process communication, but does not yet handle input-nondeterminism. The approach was developed with combination with previous work on dynamic symbolic execution [6] in mind. Combination of the two approaches gives rise to full (bounded) model checking of Creol.

²we refer to the current version of JPF. Initially from NASA Ames and made open source in 2005 (<http://javapathfinder.sourceforge.net/>). The first version of JPF performed model checking by translation to Promela and used SPIN.

References

- [1] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. Springer, 1999.
- [2] M. Clavel, 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(2):187–243, Aug. 2002.
- [3] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 40(1):110–121, 2005.
- [4] C. Flanagan and S. Qadeer. Transactions for Software Model Checking. *Electronic Notes in Theoretical Computer Science*, 89(3), 2003.
- [5] P. Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos, and P. Wolper. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Springer New York, 1996.
- [6] A. Griesmayer, B. Aichernig, E. Johnsen, and R. Schlatte. Dynamic symbolic execution for testing distributed objects. In *Second International Conference on Tests and Proofs (TAP)*, LNCS. Springer-Verlag, 2009. to be published.
- [7] G. Holzmann. *The SPIN model checker: Primer and reference manual*. Addison-Wesley Professional, 2004.
- [8] E. B. Johnsen, O. Owe, and E. W. Axelsen. A run-time environment for concurrent objects with asynchronous method calls. In N. Martí-Oliet, editor, *WRLA*, volume 117 of *ENTCS*, pages 375–392, Amsterdam, Jan. 2005. Elsevier.
- [9] 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):23–66, Nov. 2006.
- [10] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
- [11] R. Palmer, G. Gopalakrishnan, and R. M. Kirby. Semantics driven dynamic partial-order reduction of mpi-based parallel programs. In *PADTAD '07: Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging*, pages 43–53, New York, NY, USA, 2007. ACM.
- [12] D. Peled. Ten years of partial order reduction. In A. J. Hu and M. Y. Vardi, editors, *CAV*, volume 1427 of *Lecture Notes in Computer Science*, pages 17–28. Springer, 1998.
- [13] A. Valmari. A stubborn attack on state explosion. In *Computer-Aided Verification'90: Proceedings of a DIMACS Workshop, June 18-21, 1990*, page 25. American Mathematical Society, 1991.
- [14] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

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