Automating Commutativity Analysis at the Design Level

Greg Dennis, Robert Seater, Derek Rayside and Daniel Jackson
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge MA 02139, USA
{gdennis, rseater, drayside, dnj}@mit.edu

Abstract

Two operations commute if executing them serially in either order results in the same change of state. In a system in which commands may be issued simultaneously by different users, lack of commutativity can result in unpredictable behaviour, even if the commands are serialized, because one user's command may be preempted by another's, and thus executed in an unanticipated state.

This paper describes an automated approach to analyzing commutativity. The operations are expressed as constraints in a declarative modelling language such as Alloy, and a constraint solver is used to find violating scenarios. A case study application to the beam scheduling component of a proton therapy machine (originally specified in OCL) revealed several violations of commutativity in which requests from medical technicians in treatment rooms could conflict with the actions of a beam operator in a master control room. Some of the issues involved in automating the analysis for OCL itself are also discussed.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, model checking*
F.4.3 [**Mathematical Logic**]: Formal Languages—*Alloy*
H.1.2 [**Information Systems**]: User/Machine Systems

General Terms

design, human factors, reliability, verification

Keywords lightweight formal methods, model checking, testing, formal specification, concurrency, critical systems, commutativity, case study, proton therapy, radiation therapy, Alloy, OCL

1 Introduction

Two operations α and β are said to *commute* if the sequence $\alpha\beta$ leaves the system in the same state as the sequence $\beta\alpha$. Knowing whether operations commute or not can be useful in a number of contexts. In transaction systems, for example, it can be used for concurrency control; in compilers, it can be used to find opportunities for parallelization; and, as we shall see, in a multi-user system, it can be used to identify pairs of operations that may confuse human operators.

Suppose that operations α and β do not commute, and that user A commands α at (almost) the same instant as user B commands β. Depending on which operation is executed first, one or other of the users might be surprised by the result. The design of the system can mitigate this effect in various ways (that are beyond the scope of this paper), but nothing can be done until the scenarios in which it arises have been identified.

This paper introduces *automated design-level commutativity analysis* to determine which pairs of operations do or do not commute, given a design expressed in a declarative modelling formalism. This analysis can alert the designer to potential commutativity issues before the system is developed, and thereby lead to an improved design and test cases.

The problem of checking commutativity of two operations is reduced to a constraint solving problem. We express each operation as a constraint in the Alloy modelling notation [8]. The assertion that two operations commute is itself a constraint, whose negation is satisfied by scenarios in which commutativity fails. The Alloy Analyzer [12] translates constraints to boolean formulas, and solves them with the aid of off-the-shelf SAT solvers [7]. Usually, when two operations do not commute, a violating scenario is found in a few seconds.

We applied this approach to a component of a proton therapy machine that schedules radiation treatments amongst several treatment rooms. This *automatic beam scheduler* (ABS) is in essence a queuing mechanism that automatically determines which treatment room will receive the beam. The ABS itself executes in a single thread, and so from the perspective of the computer all operations occur in a serial order. However, the ABS has multiple simultaneous users, in the various treatment control rooms and in the master control room.

Most previous work on commutativity has focused on how knowledge of commutativity can be used for concurrency control, but has not provided a mechanical means for determining which operations do or do not commute [19, 3, 20]. Program analyses have been developed to determine automatically if two procedures commute, but this determination is typically based on whether the procedures may modify the same object [16]. For the ABS system, such an analysis would be far too conservative. We know that the procedures all modify the same core object: the beam request queue. We need to determine, at the design level, which pairs of operations will leave that object in the same state, regardless of the order in which the operations are executed.

We formulate our commutativity analysis as a constraint satisfaction problem in the declarative object modelling language Alloy [12]. This mode of expression is more convenient than formulation in a conventional model checker, because Alloy supports complex data structures and modular analysis, and because the temporal logic used by most conventional model checkers does not provide a means to compare the states that result from two different actions being performed.

Furthermore, the developers of the ABS originally specified its design in OCL (the Object Constraint Language [17]), which is semantically similar to Alloy. The current state of the art in OCL tools supports evaluation of constraints with respect to given instances, but cannot find instances satisfying a constraint. These tools cannot therefore perform commutativity analysis. Alloy's analyzer, in contrast, can search large spaces for counterexamples to assertions; the particular analysis problem of this study was well within Alloy's capability.

The main contributions of this paper are:

- a technique for automated design-level commutativity analysis;
- a report of a case study applying this analysis to a radiation therapy machine component;
- a discussion of our experience in translating OCL to Alloy, and the obstacles that would have to be overcome to perform the analysis on OCL directly.

2 Context: The Therapy Control System

The Northeast Proton Therapy Center (NPTC) installation has at its core a cyclotron that generates a beam of protons. The beam is multiplexed amongst several treatment rooms, each with its own gantry and nozzle for positioning the beam. Technicians in a master control room supervise the cyclotron and allocate the beam to treatment rooms. Each treatment room is paired with a treatment control room, in which clinicians enter and execute treatment prescriptions.

The patient is placed on a couch that is electromechanically positioned by staff within the treatment room. The beam emitter is also positioned, and its aim verified by staff using X-rays and lights attached to the emitter. The staff then leave the room, and the treatment is initiated from the treatment control room. Treatment consists of irradiating a specific location on the patient using a beam of protons with a defined lateral and longitudinal distribution.

The machine is considered safety critical primarily due to the potential for overdose—treating the patient with radiation of excessive strength or duration. The International Atomic Energy Agency lists 80 separate accidents involving radiation therapy in the United States over the past fifty years [15]. Infamous amongst these accidents are those involving the Therac-25 machine [10], in whose failures faulty software was a primary cause. More recently, software appears to have been the main factor in similar accidents in Panama in 2001 [4].

The NPTC system was developed in the context of a sophisticated safety program. Unlike the Therac-25, the NPTC system makes extensive use of hardware interlocks, and has a redundant PLC-based system running in parallel with the software control system. Video cameras inside the control room allow the technicians to view internal mechanisms, including a lead beam stop that can be inserted to isolate the treatment room from the cyclotron. The software itself is instrumented with abundant runtime checks, including a heartbeat monitor to ensure continued operation of critical processes. A detailed system-level risk analysis was performed. The software implementation was heavily tested, and manually reviewed against rigorous coding standards.

2.1 Automatic Beam Scheduler

Currently, Treatment Control Room (TCR) operators contact the Master Control Room (MCR) by telephone to request the beam. The MCR operator allocates the beam to one of the treatment rooms. When the beam is allocated to a room, the operator of that room (and that room alone) can instruct the beam to fire, and, if the beam fires, it will be directed to that room (and that room alone). Allocating the beam does not turn it on, although deallocating the beam will turn it off.

The NPTC proposes to automate this procedure by adding an Automatic Beam Scheduler (ABS) component, with the intent of lightening the burden of the MCR operator, and improving throughput of the machine, thus serving more patients. The ABS will automate the task of deciding when and to which room the beam should be allocated. Under the new procedure, TCR operators will submit beam requests to the ABS, indicating one of three priority levels:

1. **Service Priority** (lowest): for testing the machine, calibrating it, and performing research;
2. **Normal Priority**: for typical requests;
3. **High Priority**: in cases where the patient is young, restless, or anesthetized. These are not emergencies, but should be handled before less urgent cases.

The ABS maintains a data structure with a slot for the currently allocated request, and a queue of pending requests. Once the TCR of the allocated room releases the beam, the ABS automatically allocates the beam to the request at the head of the queue. The queue is by default ordered first-in first-out, but the ordering is complicated by priority levels and the possibility of manual intervention.

The following commands are issued from the TCRs:

- RequestBeam issues a normal or service priority request for a given room.
- RequestBeamHighPriority issues a high priority request for a given room. If the currently allocated request is not of high priority, RequestBeamHighPriority moves the currently allocated request back to the pending queue and allocates the new high priority request. Otherwise it will insert the high priority request in the queue.
- CancelBeamRequest deletes a pending request from the queue.
- ReleaseBeam deletes an allocated request, if one is currently allocated. If the beam is firing in the allocated room, this operation turns it off.

The following commands are issued from the MCR:

- StepUp moves the selected request one step closer to the front of the queue, but never moves a lower priority request in front of a higher priority request;
- StepDown moves the selected request one step closer to the back of the queue, but never moves a higher priority request behind a lower priority request;
- Flush deletes all requests of a given priority from the queue;
- FlushAll deletes all requests from the queue, regardless of priority.

2.2 Potential Risks of the ABS

The Master Control Room (MCR) operator and the Treatment Control Room (TCR) operators issue commands asynchronously to the Beam Scheduler. If two non-commuting commands, α and β, are issued by different operators at almost the same time, processing α before β could yield a different state than processing β before α. Consequently, operators would not be able to predict the results of their actions. In contrast, if α and β commute, then neither operator is surprised, regardless of their order.

Prior to the analysis, we realized that certain pairs of commands issued to the Automatic Beam Scheduler (ABS) clearly did not commute. Consider the case in which the MCR operator issues a FlushAll (emptying the queue of pending requests) at approximately the same time as the TCR operator issues a RequestBeam (adding a request to the pending queue); depending on the order of their execution, the resulting queue could either have a single request or be completely empty. In the latter case, a new request would have been inadvertently flushed.

Though inadvertent flushing may be relatively benign, the possibility of other unexpected actions being taken by the ABS is more worrisome. Since the ABS cannot actually turn the beam on, a reordering cannot cause the beam to be fired unintentionally. However, scenarios in which the beam is wrongly allocated or deallocated are still problematic.

For example, if a beam is unintentionally allocated to a low-priority request, it could cause a higher priority request to languish in the queue, waiting for the beam to be free again. On the flip side, an unintentional deallocation of the beam can turn off the beam prematurely. Worse, a deallocation of the beam is often followed by an automatic allocation of another request in the queue, thereby producing potentially long delays before treatment can continue in the interrupted room. Preparing and angling the beam and positioning and anesthetising the patient can take an hour, so a premature termination of the treatment would be at best inconvenient. At worst, it will result in an uneven or unknown dose being delivered, complicating future treatments.

3 Commutativity Analysis

Commutativity analysis is applicable to any system in which commands are executed atomically but asynchronously, and have state effects that are not always orthogonal to each other. In short, it checks whether executing a command α before a command β has the same effect as executing β before α. If not, unexpected and possibly undesirable results can occur when both are issued simultaneously.

A word of clarification: Commutativity analysis does not consider now non-atomic operations performed at the same time can interleave to yield unacceptable results (such as read/write race conditions); it is an analysis of how atomic operations issued at roughly the same time can be ordered in such a way as to yield unwanted results. Lack of commutativity is not necessarily an error, but should be addressed either internally or in the user interface.

3.1 Commutativity Properties

Given any two atomic operations α and β, commutativity comprises two properties: *diamond equivalence* and *diamond connectivity* [Figure 1].

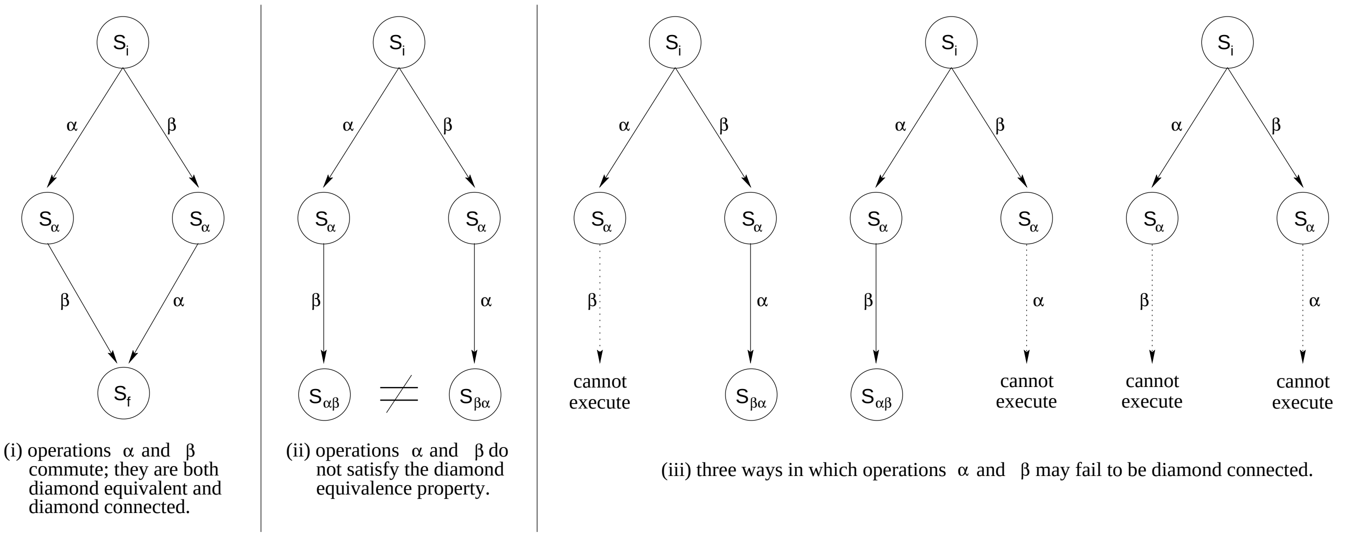
1. **diamond equivalence:** If $\alpha\beta$ and $\beta\alpha$ can both occur from the same initial state — their preconditions are met — then the states that result from applying $\alpha\beta$ or $\beta\alpha$ to the initial state should be observably equivalent (although not necessarily strictly identical).

   Therefore, if two diamond-equivalent operations are issued at the same time by different users, it does not matter which is executed first. If two commands fail diamond equivalence, then human operators cannot predict the state of the system after executing one of the commands. The commands might be accepted, but due to an unexpected ordering, result in surprising and undesirably results.

2. **diamond connectivity:** If $\alpha\beta$ can occur and β can occur from the initial state — their preconditions are met — then $\beta\alpha$ can also occur in the initial state.

   Therefore, if two diamond-connected operations are issued at the same time, then the execution of one cannot preclude the execution of the other. If two commands fail diamond connectivity, then human operators cannot predict whether or not those commands will be accepted by the system. Even if the command is legal to execute when the operator decides to execute it, an unfortunate reordering may result in it being rejected anyway.

Figure 1 Commutativity properties



(i) operations α and β commute; they are both diamond equivalent and diamond connected.

(ii) operations α and β do not satisfy the diamond equivalence property.

(iii) three ways in which operations α and β may fail to be diamond connected.

We will express an operation as a constraint parameterized by its pre-state S and its post-state S', explicitly separating the pre- and post-conditions (which are likewise represented as parameterized constraints):

$$Op(S, S') = \quad Pre(S) \wedge Post(S, S') \tag{1}$$

Note that our pre-conditions represent guards and not disclaimers, so that an operation invoked in a state that violates a pre-condition will be explicitly rejected. For two operations α and β, we therefore have:

$$\begin{aligned} Op_\alpha(S, S') = &\quad Pre_\alpha(S) \wedge Post_\alpha(S, S') \\ Op_\beta(S, S') = &\quad Pre_\beta(S) \wedge Post_\beta(S, S') \end{aligned} \tag{2}$$

The operations Op_α and Op_β are diamond-equivalent if for all states S_i, S_α, S_β, $S_{\alpha\beta}$, $S_{\beta\alpha}$:

$$\begin{array}{l} Op_\alpha(S_i, S_\alpha) \wedge Op_\beta(S_\alpha, S_{\alpha\beta}) \wedge \\ Op_\beta(S_i, S_\beta) \wedge Op_\alpha(S_\beta, S_{\beta\alpha}) \end{array} \Rightarrow S_{\alpha\beta} = S_{\beta\alpha} \tag{3}$$

And the operations Op_α and Op_β are diamond-connected if for all states S_i, S_α, S_β:

$$Op_\alpha(S_i, S_\alpha) \wedge Op_\beta(S_i, S_\beta) \Rightarrow Pre_\alpha(S_\beta) \wedge Pre_\beta(S_\alpha) \tag{4}$$

Note that for any pair of operations, there are three ways in which the diamond connectivity property can be violated: (1) when the first operation blocks the second, (2) when the second blocks the first, and (3) when they mutually block each other. All three scenarios are depicted in Figure 1.

Two additional comments on these properties are in order. First, the initial state S_i should be restricted to a legal states satisfying the invariants; clearly it is unproductive to find an instance of non-commutativity that begins from an illegal state.

Second, it is important to note that if an operation were non-deterministic, it would fail the diamond equivalence test even when combined with a no-op (since two different outcomes can be produced from a single state). A more sophisticated notion of commutativity could be developed to identify the additional non-determinism that arises from asynchronous interleavings. However, in our case – and we suspect in most contexts in which the notion will be useful – the operations are deterministic, so a simpler definition will do. Determinism of an operation is an easy property to check, as explained below (in Section 5.2).

That said, there is a harmless form of non-determinism arising from the treatment of states as atoms in their own right. Just like the objects in an object-oriented program, two distinct states can be observationally equivalent. This can be handled either by 'canonicalizing' the state (with a constraint that requires distinct states to be observationally distinct), or by replacing equality tests on states with equivalence tests (using an auxiliary predicate that defines equivalence of states in terms of their components). We chose the second approach, which is why the Alloy formulation of the commutativity test (in Figure 4) has `equivQueues (e1, e2)` in place of `e1 = e2`.

3.2 Commutativity Analysis as Constraint Solving

To analyze the commutativity of the Beam Scheduler operations, we used the Alloy Analyzer, a constraint solver for Alloy, a first-order relational logic similar to Z.

We began by translating the existing OCL model into Alloy. We modelled the basic entities in the Beam Scheduler system and environment, the commutativity properties, and the operations we wished to check for commutativity in Alloy. Next, we divided the operations into those performed by the MCR operator and those performed by the TCR operator, and we formulated claims in Alloy that each MCR-TCR operation pair satisfied both the diamond equivalence property and the diamond connectivity property. Details of this model can be found in Section 4.

To verify a claim, the Alloy Analyzer first negates the claim and then proceeds to search for a scenario in which the negated claim is true. If the model is a correct representation of the system, then such a scenario corresponds to a counterexample to the claim. In our model, a counterexample is an instance in which an MCR-TCR operation pair yields different states depending on the order in which the operations are applied.

3.3 On Finite Scope

The Alloy Analyzer does not symbolically prove whether or not the instances of non-commutativity exist. Instead, it exhaustively searches the entire state space of scenarios within user-defined bounds. It is able to analyze millions of scenarios in a matter of seconds. Consequently, failure to find a counterexample does not constitute proof that the claim is valid, but the reporting of a counterexample does imply that it is invalid.

Each analysis in Alloy is parameterized by a *scope*, which assigns a bound to each basic top-level type. The scope is provided by the user as part of a command to check an assertion or simulate a predicate. Since the search for a solution is conducted only within the scope, the absence of a counterexample does not necessarily indicate that a conjecture is valid (nor does absence of an instance indicate that a predicate is inconsistent). In practice, however, small scopes suffice to find most flaws, and the user's confidence increases as the scope is increased.

4 Analysis of the Beam Scheduler

To perform the commutativity analysis, we began by translating the OCL model into the Alloy modelling language.

4.1 Sets and Relations

The first step in building an Alloy model is to declare the basic sets corresponding to the entities of the problem domain, and the relations amongst them (Figure 2).

From the OCL, we identified the following basic sets: Request, Room, Priority, OrderID, and Queue. Here, the entity Queue represents the entire state of the ABS, i.e., both the currently allocated request and the actual queue of pending requests.

The fields within signatures define relations over the signature types. In Figure 2 the room and priority fields are binary relations, both of which map Requests to exactly one room and exactly one Priority, respectively. The fields requests, alloc, and pending are also binary relations, where the set keyword allows these relations to map each Queue to zero or more Requests. The last relation, order, is a ternary relation that maps each Queue to a relation that maps requests in that Queue to exactly one OrderID. In sum, a request has a room and priority, and a queue has a set of allocated and pending requests, and a mapping of those requests to their order ids.

The effect of line 4 is to partition the set Priority into three singleton subsets named Service, Normal, and High, representing the three possible priority levels of requests. The constraint on line 18 makes the set of requests in the queue equal to the union of the set of allocated requests and the set of those that are pending (note that every request is either allocated or pending).

Figure 2 Signatures

```
1  sig Room {}
2
3  abstract sig Priority {}
4  one sig Service, Normal, High extends Priority {}
5
6  sig Request {
7      room: Room,
8      priority: Priority
9  }
10
11 sig OrderID {}
12
13 sig Queue {
14     requests: set Request,
15     alloc, pending: set Request,
16     order: requests → one OrderID
17 }{
18     requests = alloc + pending
19 }
```

Figure 3 Some operations

```
1  pred preRequestBeam(q: Queue, r: Room, p: Priority) {
2      p in Normal + Service
3      r not in q.(pending + alloc).room
4  }
5
6  pred RequestBeam (q, q': Queue, r: Room, p: Priority) {
7      preRequestBeam(q, r, p)
8      some req : q'.requests {
9          req.priority = p
10         req.room = r
11         p in Normal ⇒ NextOrderID(q, q'.order[req], Normal)
12         p in Service ⇒ NextOrderID(q, q'.order[req], Service)
13         no q.alloc ⇒ (q'.alloc = req) && (q'.pending = q.pending)
14             else (q'.alloc = q.alloc) && (q'.pending = q.pending + req)
15     }
16     q.order in q'.order
17 }
```

4.2 Operations

Having modelled the basic entities of our system, we proceeded to translate the eight OCL operations on these entities into Alloy.

We modelled all these operations as Alloy *predicates*. To check for diamond connectivity, we factored out the preconditions of each operation into separate predicates so they could be independently invoked.

To illustrate, an Alloy formulation of the RequestBeam operation is shown in Figure 3. Recall that RequestBeam accepts a room and a priority as arguments and adds a request with that room and priority to the list of pending requests in the queue, providing there is no allocated or pending request for that room and the priority level is Service or Normal.

The precondition of RequestBeam is given in the Alloy predicate preRequestBeam. To avoid code duplication, RequestBeam begins by 'invoking' preRequestBeam; this simply has the effect of including the constraints of the latter in the former (appropriately instantiated). It also stipulates that there be a request in the post state with the given room and priority in the queue.

4.3 Checking Commutativity

To check the diamond equivalence and diamond connectivity properties, we created Alloy *assertions*, which are formulas the Analyzer can check for validity. We wrote two assertions for each MCR-TCR operation pair: one to encode the diamond equivalence formula (Formula 3), and one to encode the diamond connectivity formula (Formula 4). For each assertion, we added a *check command* to verify the assertion within a scope of 6. The assertions and commands to check the diamond equivalence of CancelBeamRequest and StepUp and the diamond connectivity of RequestBeam and ReleaseBeam are shown in Figure 4. (The Invariants predicate in the assertions constrains the initial state to be a legal state; see section 5.3.)

Figure 4 Commutativity property assertions

```
assert Cancel_StepUp_Equiv {
    all s, m1, m2, e1, e2: Queue, rq1, rq2: Request {
        (Invariants(s) && CancelBeamRequest(s, m1, rq1) &&
        StepUp(m1, e1, rq2) && StepUp(s, m2, rq2) &&
        CancelBeamRequest(m2, e2, rq1))
        ⇒ equivQueues(e1, e2)
    }
}
assert Request_Flush_Connect {
    all s, m1, m2: Queue, r: Room, p1, p2: Priority {
        (Invariants(s) && RequestBeam(s, m1, r, p1) &&
        Flush(s, m2, p2))
        ⇒ (preFlush(m1, p2) && preRequestBeam(m2, r, p1))
    }
}
check Cancel_StepUp_Equiv for 6
check Request_Flush_Connect for 6
```

4.4 Results

The commutativity analysis was performed on a Windows XP machine with an Intel Pentium III 597 MHz processor with 192 MB of RAM. Each of the thirty two commands were executed in a scope of six, and ranged in execution time between three and one hundred seconds. The following operation pairs were found not to commute:

- RequestBeam/RequestBeamHighPriority and Flush/FlushAll violate diamond equivalence. If a RequestBeam and a FlushAll happen at the same time, the result is either an empty queue (if the flush is last) or a single request allocated (if the request is last). In the former case, the last request is mistakenly flushed.

- CancelBeamRequest and StepUp/StepDown violate diamond connectivity and diamond equivalence. If the cancel happens first, then StepUp/StepDown fails because the request is no longer pending – a violation of diamond connectivity. A second, more subtle case violates diamond equivalence: When the request above the request being stepped up is cancelled, the stepped-up request might remain at its current position in the queue (if the StepUp comes first) or move up past the request ahead of the canceled request (if the CancelBeamRequest comes first). An analogous scenario applies to StepDown.

- CancelBeamRequest and Flush/FlushAll violate diamond connectivity. If a CancelBeamRequest is issued at the same time as a Flush or a FlushAll command and the request to be cancelled is flushed before the cancel is performed, then the CancelBeamRequest will fail.

  We also applied the analysis to some TCR/TCR operation pairs and found some additional cases of non-commutativity:

- RequestBeamHighPriority and ReleaseBeam violate diamond connectivity. Consider the scenario in which a Normal or Service priority request R is allocated. And suppose a command to release the beam from request R is issued at the same time as a RequestBeamHighPriority. If the request is executed before the release, R will be returned to the queue to make way for the high priority request, and the subsequent ReleaseBeam will fail because its precondition that the request be allocated is now false.

- ReleaseBeam and CancelBeamRequest violate diamond connectivity. When a ReleaseBeam occurs in automatic mode, the scheduler automatically allocates the next request. If the next request to be allocated is simultaneously cancelled, the CancelBeamRequest may succeed (if it comes first) or fail (if it comes second).

- RequestBeam and ReleaseBeam violate diamond equivalence. Consider an initial state where a Service priority request is pending. If the beam is released and then a Normal priority request is issued, the Service priority request will now be allocated and the Normal priority request will be pending. However, if a Normal priority request is issued and then the beam is released, the Normal priority request will be allocated and the Service priority request will be pending.

5 Other Analyses

We also applied a variety of more conventional analyses to the ABS case study, in addition to commutativity analysis.

5.1 Simulation and Sanity Checks

Simulation involves running test executions of operations and subsequently examining the executed traces. Clearly, simulation is far from a comprehensive analysis of a system; but it can lend insight into the general operation of the system, reveal egregious errors, and if the simulation scenarios are chosen properly (or luckily), stumble upon more subtle deficiencies.

The Alloy Analyzer will simulate any predicate, finding instances that satisfy it (if they exist). We simulated several of the operations in their original form, and also in a modified form in which we added various additional pre- and post-conditions to explore executions of particular interest (such as those that leave the queue empty, or change the allocated request).

A sanity check is a non-trivial verification that the system is not flawed in some fundamental way. For example, we checked that StepUp (which moves a request one step closer to the front of the queue) does not alter the set of requests pending in the queue. Like simulation, sanity checks are useful for uncovering basic faults in the system; but unlike simulation, sanity checks are more comprehensive because they can examine all traces within a given scope.

5.2 Determinism Checks

Often, an operation is assumed to be deterministic. As explained above, our formulation of commutativity requires this. An operation $Op(S, S')$ is deterministic if, for every pre-state S and possible post-states S' and S'',

$$Op(S, S') \wedge Op(S, S'') \Rightarrow S' = S'' \qquad (5)$$

This conjecture can be written directly in this form in Alloy, and checked automatically.

5.3 Invariant Preservation

An invariant is a property that should be preserved by every operation; if it holds before the operation, then it must still hold afterwards. For example, one invariant is that if the pending queue is non-empty, then the beam is allocated to some room. Any command that deallocates the beam must ensure that another request is allocated if one is pending.

An invariant takes the form of a predicate on the state of the system that is true if and only if the system is in a correct state. Let $Inv(S)$ represent our invariant. An operation $Op(S, S')$, a predicate on a pre-state and a post-state, preserves the invariant if for all pre-states S and all post-states S',

$$Inv(S) \wedge Op(S, S') \Rightarrow Inv(S') \qquad (6)$$

A solution to the negation of this formula corresponds to a situation where the operation does not preserve the invariant.

We expressed the invariants of the Beam Scheduler in a predicate called Invariants shown in Figure 5. The predicate consists of the following constraints:

- At most one request is allocated at any point in time;
- If no requests are allocated, then no requests may be pending;
- If a high priority request is pending, then a high priority request must be allocated;
- No request is both allocated and pending;
- No two requests in the queue (either allocated or pending) are for the same room;
- Any two pending requests of the same priority must have different order identifiers.

We used the Alloy Analyzer to check that each operation preserves the invariant listed in Figure 5. For each of the eight operations in our model, we wrote an assertion encoding the invariant preservation formula (Formula 6) in Alloy. For example, the assertion that the ReleaseBeam operation preserves the invariant is shown in Figure 6.

Figure 5 Queue invariants

```
pred Invariants (q:Queue) {
    lone q.alloc
    no q.alloc ⇒ no q.pending
    High in q.pending.priority ⇒ High in q.alloc.priority
    no q.alloc & q.pending
    no disj r,r':q.requests | r.room = r'.room
    all disj r,r': q.pending |
        (r.priority = r'.priority) ⇒ (q.order[r] != q.order[r'])
    all disj r,r':q.alloc |
        (r.priority = r'.priority) ⇒ (q.order[r] != q.order[r'])
}
```

Figure 6 ReleaseBeam preserves invariant check

```
assert ReleaseBeamPreservesInv {
    all q, q': Queue, req: Request |
        Invariants(q) && ReleaseBeam(q, q', req) ⇒ Invariants(q')
}
```

6 OCL Experience

The original design of the beam scheduler was expressed in OCL [17], the constraint language of UML. The OCL model was not mechanically checked, although a number of analyzers for OCL have recently become available. In this section, we review the errors we found by translation to Alloy and subsequent analysis, and we discuss opportunities for direct analysis of OCL itself.

6.1 Translation to Alloy

The translation from OCL to Alloy was straightforward. OCL *contexts* were readily converted into Alloy *signatures*, and OCL *operations* into Alloy *predicates*. The OCL specification omitted frame conditions, relying on the reader's intuitions to provide them. They were therefore added to the Alloy specification. The lack of a frame condition is readily exposed by invariant checking, so, with the Analyzer's help, it is easy to determine where frame conditions are required.

To illustrate the directness of the translation, OCL formulations of FlushAll and CancelBeamRequest are shown in Figure 7 and their corresponding Alloy versions in Figure 8. As shown in the figures, it is necessary to add frame conditions to the Alloy specification that FlushAll does not change the allocated request and that CancelBeamRequest does not change the allocated request nor the order of the pending requests.

Figure 7 OCL Specification of FlushAll and CancelBeamRequest

```
context BeamScheduler::flushAll()
post:
    self.pendingRequests→size == 0

context BeamScheduler::
        cancelBeamRequest(req: BeamRequest)
pre:
    self.pendingRequests@pre→exists(r | r == req)
post:
    not self.pendingRequests→exists(r | r == req)
```

Figure 8 Alloy Specification of FlushAll and CancelBeamRequest

```
pred FlushAll(q, q': Queue) {
    no q'.pending
    q'.alloc = q.alloc
}
pred CancelBeamRequest(req: BeamRequest) {
    req in q.pending
    q'.pending = q.pending − req
    q'.alloc = q.alloc
    q'.order = q.order − req→OrderID
}
```

6.2 Errors Found in OCL Model

The ready availability of automatic analysis results in higher quality models, with fewer errors. Neither the commutativity analysis nor the more conventional analyses of Section 5 can be applied directly to the original OCL model, due to the limitations of the available OCL tools.

Not surprisingly, subjecting the OCL model to analysis (indirectly, via translation to Alloy) exposed a variety of small flaws. Most of these were minor and were easy to correct. Some, such as the misspelling of association names, would have been caught by syntax and type checking. Others required simulation to expose logical inconsistency. Operations differed, for example, about whether order ids of pending requests increased or decreased towards the front of the queue, and whether order ids were unique (some operations testing this explicitly, and others assuming it).

6.3 Solving Versus Evaluation

The analysis offered by OCL tools differs fundamentally from Alloy's. The primary analysis they offer, in addition to syntax and type checking, is *evaluation*, in which the user provides an instance, and the tool checks whether it satisfies a constraint by evaluating each subexpression and assigning true or false to the constraint as a whole. Alloy, in contrast, offers *solving*, which involves searching to find an instance satisfying a given constraint.

Evaluation helps, but can be tedious to use, since a value must be given for every single set and relation. An earlier version of Alloy offered evaluation also, but it was rarely used, since it is usually easier to specify a particular instance as a partial constraint, letting the solver fill in the details, and specifying more if the expected instance is not obtained.

Analyses that reveal subtle bugs in a model – such as checking preservation of invariants, or the commutativity analysis of this paper – require solving. It is therefore worth considering what might be done to bring such analysis to OCL.

6.4 Syntactic Issues

OCL and Alloy are, at heart, very similar languages. Both are designed for lightweight modelling via constraints. Alloy's syntax is based on the traditional syntax of first order logic, and the relational operators of Z; OCL's syntax is influenced by Smalltalk, and has a more operational flavour. For example, the set comprehension defining the pending requests that have service priority is written in Alloy as

```
{r: Request | r in q.pending and r.priority = Service}
```

and in OCL as

```
self.pending→select(r | r.priority == #SERVICE)
```

(where q and self are the respective names for the current queue state). Alloy's relational operators also admit more succinct forms that are favored by experienced users (but less readable by novices), such as

```
q.pending & priority.Service
```

(in which priority.Service navigates backwards from the service priority to a set of requests). For analyzability, however, these syntactic differences are immaterial.

6.5 Semantic Issues

OCL has a more complicated semantics than Alloy for two reasons. First, its type system is based on object-oriented programming languages, so typecasts are required (and may fail). Second, whereas Alloy treats associations uniformly as relations, and navigation as relational image, OCL has a variety of special cases. An association of zero/one multiplicity is treated as a function whose navigation may result in an undefined value. An association of zero or more multiplicity is treated as a relation whose navigation may result in the empty set. Navigating from a set gives a bag, but navigating from a bag gives another bag, so that a navigation via a single relation is different from a navigation via more than one. There are also implicit flattenings that are applied.

These complications should not affect the fundamental analyzability of the language, although they would make implementation much more challenging. Alloy reduces to a small relational kernel, which makes it possible to confine the analysis proper to a much simpler backend. The first task in developing an analysis for OCL would be to design such a kernel for it.

In terms of expressive power, Alloy and OCL are incomparable. Alloy includes transitive closure as an operator, so it can express reachability properties that cannot be expressed in OCL. But OCL allows sets of sets to be created, and nested to arbitrary depth (although its quantifiers are first order). Alloy's translation from relational logic to SAT [7] will therefore not be applicable to OCL directly. There are two causes for optimism, however.

First, the higher-order features of OCL are not often used. OCL uses UML diagrams for declaring sets and relations, so nested sets cannot be declared, and only arise when constructed as an expression. Associations can yield ordered sets or sequences, but these are rarely used, and when they are used, could often be eliminated. In 'Royal and Loyal', the running example in the OCL textbook [18], only one association yields an ordered set, for representing the service levels of a loyalty program. It would not affect the model as a whole to order the service levels globally with a homogeneous association on the service level class. The only other examples of potentially higher-order features in the book are in stand-alone expressions and not in model constraints.

Second, even when higher-order features are used, it may be possible to render them first-order by a subtle change to the semantics. This is what Alloy does. In Alloy, higher-order objects (such as sequences and sets of sets) are represented as atoms. For example, the declaration

```
sig MySet {elements: set X}
```

declares a set of atoms MySet and a relation elements mapping each to a set of atoms drawn from X. Analysis involves finding values for these sets and the relation. In almost all cases, this allows MySet to be viewed as a set of composite values, each holding a set of elements. But occasionally the encoding is exposed. For example, the assertion

```
assert {all s: set X | some ms: MySet | s in ms.elements}
```

admits a (counterintuitive!) counterexample in which MySet is simply made empty. There are ways to mitigate this problem when it does arise, but there is no free lunch: higher order logic cannot be reduced to first order logic. Since OCL, like Alloy, does not allow higher-order objects to be declared, this approach should work well for it too.

OCL has the standard built-in types of a programming language. Some of these are either already in Alloy (integers with addition and subtraction, for example) or could be provided in a library module (strings, for example). Real numbers with multiplication and division are a serious obstacle however, since they are not supported even by specialized decision procedures.

7 Related Work

Work related to our analysis fall roughly into four categories: conventional model checking, concurrency control, parallelizing compilers, and OCL analysis.

7.1 Conventional Model Checking

There are three obstacles to performing commutativity analysis with conventional model checking [13, 9, 5], as opposed to using a declarative modeling language such as OCL or Alloy.

Limitations of temporal logic The most fundamental obstacle is that conventional model checkers require properties to be expressed in temporal logic. However, temporal logic does not provide a means to compare the states resulting from two different action sequences. Nor is it possible to determine if those two action sequences originate from the same state. These two tasks are fundamental to determining if two commands commute; one must compare the states resulting from the possible orderings of the commands when executed from a common pre-state.

Non-modularity Another drawback of conventional model checking is that it is not modular with respect to operations. A model checker typically checks that a given property holds not for all executions of an operation but for all reachable states. An operation cannot therefore be checked in isolation, and an analysis like our commutativity analysis, which requires considering all possible pre-states, cannot be done. Generating a set of pre-states is not possible for an explicit model checker, such as SPIN [5], but might be done in a symbolic model checker such as SMV [11] (using the *trans* facility), although it is still unclear how one would encode the commutativity checks.

Lack of complex data structures Conventional model checking does not support complex data structures as part of the representation of state, and thus does not lend itself to encoding the ABS priority queue in a natural fashion. Java Path Finder [1] can handle complex data structures, but would require formulating the model more concretely in Java. SPIN/Promela [5] offers records and arrays, which would probably suffice for this model, but would again require a less abstract and less succinct description.

7.2 Commutativity-Based Concurrency Control

Commutativity has been exploited for the purposes of concurrency control in a number of different ways. Weihl used the commutativity of operations to develop novel concurrency control algorithms [19]. Fekete et al expanded upon this work to nested transaction systems [3]. More recently, Fekete and Wu showed how identifying commutative operations could yield greater concurrency in the context of application code [20]. Unlike our work, these techniques do not automatically determine the commutativity or non-commutativity of operations, but presume it has been determined in advance.

7.3 Parallelizing Compilers

Conservative commutativity analysis has been used in parallelizing compilers. Rinard and Diniz developed a technique for automated commutativity analysis to parallelize computations that manipulate dynamic, pointer-based data structures [16]. In their method, symbolic execution is used to determine whether reordering two operations yields the same final result. If the symbolic execution fails, the operations are conservatively assumed to not commute. If applied to the case study presented in this paper, that analysis would conservatively report that no operations commute because they all operate on a common queue.

7.4 OCL

Alloy and OCL are superficially quite similar: both provide a means to express declarative constraints about data and transitions. However, one crucial difference is that Alloy was designed to be an automatically analyzable language, so that instances of models, and counterexamples to claims can be machine-generated.

There has been recent work to make OCL machine-analyzable (e.g. [2]), although the current state of tool support seems to be limited to syntax checking, type-checking, and evaluation of constraints with respect to given instances [14, 6]. Our experience reflects the observation of Hussmann et al [6] that automated tool support greatly increases both the correctness and utility of declarative object models.

8 Lessons Learnt

Analysis of the ABS case study suggested some lessons of potential interest to researchers and practitioners.

8.1 Commutativity Analysis in General

We have demonstrated that commutativity analysis can be useful in contexts where there are multiple operators, even when the commands they execute are atomic and single threaded. There is still a potential for "human concurrency" to effectively reorder pairs of commands; if the two commands do not commute, potentially problematic non-determinism is introduced. If the two possible orderings of the commands have different effects on the system (i.e. if the commands do not commute), then the operators cannot predict the effects of their commands.

We analyzed the Automatic Beam Scheduler component of a Proton Therapy Machine which was developed with rigorous coding standards and preceded by a thorough OCL specification. Despite these precautions, our analysis revealed several non-commuting pairs of operations with the potential to produce problematic situations.

A pair of non-commuting operations is not in itself an error, but rather it is an indication of a concern that must be addressed. Indeed, some operations fundamentally do not commute (such requesting a beam request and flushing the queue of requests), in which case the designers might respond by adding blocks, warnings, or time delays to prevent users from unwittingly issuing non-commuting commands in rapid succession. Such solutions are not in the scope of this paper, but are a topic for future work.

8.2 Using a Constraint Solver for the Analysis

We have also demonstrated the feasibility and benefits of performing the analysis with a declarative constraint solver (such as Alloy). Commutativity properties can be expressed in a straightforward manner and automatically analyzed in this context.

We translated existing OCL to Alloy, to be sure that our model accurately reflected the actual design of the code, and so that we could take advantage of Alloy's automatic analysis. However, such a translation is not necessary. Were we starting from scratch, we could have modeled the ABS directly in Alloy. Alternatively, a tool for automatic analysis of OCL would have allowed us to work entirely in OCL.

That said, the translation from OCL to Alloy was straightforward. Four researchers spent roughly 20 person-hours on the modeling process: around 4 person-hours writing the actual model, around 4 person-hours understanding the general documentation and background, and the remaining 12 person-hours understanding the details of the OCL specification. The first version of the Alloy model was written from scratch in about an hour by a researcher with a solid background in discrete mathematics, no background in formal methods or OCL, moderate programming experience, and limited prior experience with Alloy. As mentioned in Section 4.4, the aggregate machine time for the entire analysis was around 20 minutes.

8.3 The Role of Analysis

Commutativity analysis aside, the lightweight analyses we applied to the Alloy model revealed a number of small errors in the original OCL model. This is neither surprising nor an indication of sloppiness on the part of the designers, given the currently non-analyzable nature of OCL. Our case study re-emphasizes the importance of the feedback and discipline provided by a mechanical analyzer.

Acknowledgments

We appreciate the assistance of Dr. Jay Flanz of Massachusetts General Hospital, and Didier Leyman and Philippe Thirionet of Ion Beam Applications (IBA).

This research was supported by: grant 0086154 ('Design Conformant Software') from the ITR program of the National Science Foundation; grant 6895566 ('Safety Mechanisms for Medical Software') from the ITR program of the National Science Foundation; and by the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

References

[1] Brat, G., Havelund, K., Park, S., and Visser, W. Java PathFinder – A second generation of a Java modelchecker. In *Workshop on Advances in Verification* (July 2000).

[2] Clark, T., and Warmer, J., Eds. *Object Modeling with the OCL: The Rationale behind the Object Constraint Language.* No. 2263 in LNCS. Springer-Verlag, 2002.

[3] Fekete, A., Lynch, N., Merritt, M., and Weihl, W. Commutativity-based locking for nested transactions. *Journal of Computer and System Sciences 41*, 1 (Aug. 1990), 65–156.

[4] Food and Drug Admininstration. FDA Statement on Radiation Overexposures in Panama. `http://www.fda.gov/cdrh/ocd/panamaradexp.html`.

[5] Holzmann, G. J. The Model Checker SPIN. *IEEE Transactions on Software Engineering 23*, 5 (May 1997), 279–295.

[6] Hussmann, H., Demuth, B., and Finger, F. Modular Architecture for a Toolset Supporting OCL. In *Proceedings of UML 2000: Advancing the Standard* (York, UK, Oct. 2000), A. Evans, S. Kent, and B. Selic, Eds., no. 1939 in LNCS.

[7] Jackson, D. Automating First-Order Relational Logic. In *Proc. ACM SIGSOFT Conf. Foundations of Software Engineering (FSE)* (Nov. 2000).

[8] Jackson, D., Shlyakhter, I., and Sridharan, M. A micromodularity mechanism. In *ACM SIGSOFT Conference on Foundations of Software Engineering / European Software Engineering Conference* (Vienna, Sept. 2001).

[9] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science* (Washington, D.C., 1990), IEEE Computer Society Press, pp. 1–33.

[10] Leveson, N. G., and Turner, C. An investigation of the Therac-25 accidents. *IEEE Computer 7*, 26 (1993), 18–41.

[11] McMillan, K. L. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993. `http://www.cs.cmu.edu/~modelcheck/smv.html`.

[12] MIT Software Design Group. The Alloy Analyzer. `http://alloy.mit.edu`.

[13] Queille, J.-P., and Sifakis, J. Specification and Verification of Concurrent Systems in CESAR. *LNCS 137* (1982), 337–351.

[14] Richters, M., and Gogolla, M. OCL: Syntax, Semantics, and Tools. In Clark and Warmer [2], pp. 42–68.

[15] Ricks, R. C., Berger, M. E., Holloway, E. C., and Goans, R. E. *REACTS Radiation Accident Registry: Update of Accidents in the United States*. International Radiation Protection Association, 2000.

[16] Rinard, M. C., and Diniz, P. C. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Transactions on Programming Languages and Systems 19*, 6 (1997), 942–991.

[17] Warmer, J., Ed. *Response to the UML 2.0 OCL RfP (ad/2000-09-03)*. Object Management Group, Jan. 2003. Revised submission, version 1.6. OMG Document ad/2003-01-07.

[18] Warmer, J., and Kleppe, A. *The Object Constraint Language: Getting your models ready for MDA*, 2^{nd} ed. Addison-Wesley, Aug. 2003.

[19] Weihl, W. E. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers 37*, 12 (1988), 1488–1505.

[20] Wu, P., and Fekete, A. An empirical study of commutativity in application code. In *Proceedings of International Database Engineering and Applications Symposium* (Hong Kong, July 2003).