Management of Feature Interactions with Transactional Regions

Thomas Cottenier
UniqueSoft, LLC
thomas.cottenier@uniquesoft.com

Aswin van den Berg
UniqueSoft, LLC
aswin.vandenberg@uniquesoft.com

Thomas Weigert
Missouri University of S&T
weigert@mst.edu

Abstract

This paper presents a modeling language to modularize the features of a system using orthogonal regions and to manage the interactions between these features. Orthogonal regions are a language construct to structure a state machine into a set of semi-independent behaviors. We introduce two concepts to manage the interactions between regions. First, we present a notion of interface between regions which captures the essence of their interactions. Second, we introduce a transactional composition operator to synchronize the regions and check the interaction for non-determinism and termination. The approach is evaluated by comparing a monolithic legacy implementation of a telecommunication component to two refactored implementations. Our results show that transactional region composition can achieves independence between the implementations of the features of the system and that it improves the cohesion of the regions, compared to classic regions.

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Language Constructs and Features – *classes and objects, modules, packages.*

General Terms Design, Languages

Keywords feature interaction, statecharts, modularity

1. Introduction

Systems that are decomposed into features take the form S = F1 + F2 + F3 where each Fi is a feature module and + denotes a feature composition operation. Structuring a system by feature requires implementing each feature into a separate module. Ideally, this module implements a cohesive piece of functionality that can be understood, implemented and tested independently of the other features so that different development teams can focus on different features of the system. Isolating the implementation of the different features also provides the flexibility of delivering different variants of a system by assembling different combinations of feature modules. However, this assembly requires a thorough understanding of the interactions between features. A feature interaction occurs when one feature modifies the way another feature contributes to the overall behavior of the system. The feature interaction problem is known to be a difficult problem of general importance [1]. The approach presented in this paper is most directly applicable to distributed systems where components interact asynchronously. We believe however that it is relevant to all systems that need to be decomposed into a set of semi-independent features for modularity or flexibility purposes.

Our goal is to allow a feature of a system to be understood and implemented independently of other features. In this paper, we present a modeling language that supports the modularization of features using orthogonal regions. Orthogonal regions are a language construct to structure state diagrams into a set of semi-independent behaviors [2]. Orthogonal regions have been widely adopted in state machine based formalisms such as the UML. When system features are modularized using regions, the feature interactions take the form of interactions between regions. These region interactions need to be understood and managed to avoid conflicts and inconsistencies. In most cases, synchronization entities such as additional states and transitions have to be added to the regions to coordinate their execution.

When the number of regions in the system grows, the interactions between regions can become very hard to maintain. Our language therefore introduces a concept of interface between regions. Such an interface concisely captures the essence of the interaction and avoids direct dependencies between regions that implement different features.

Nevertheless, the coordination of the regions still requires synchronization. The core behavior of each region becomes tangled with synchronization behavior. The introduction of states whose purpose is to coordinate with the interface region is a source of errors and can easily lead to deadlocks. In order to address these issues, our modeling language introduces a transactional composition operator that simplifies the synchronization of regions and whose semantics detects deadlocks.

We evaluate interface regions and transactional composition by comparing size, coupling and diffusion metrics for

AOSD'12, March 25–30, 2012, Potsdam, Germany.

three versions of an industrial telecommunication component. The first version was developed as a monolithic state machine by a third party. The second and third versions were obtained by refactoring the monolithic version for maintenance purposes. The second version uses regions to modularize the features of the system. The third version uses interfaces between regions and transactional composition semantics. Our results show that interface regions and transactional composition improve the quality of the system and support the management of the interactions between the system features.

The paper is organized as follows. Section 2 introduces the case study and describes the features of the system. Section 3 introduces orthogonal regions and presents the operational semantics of our modeling language. Section 4 discusses interactions between regions and interface regions and details the syntax and semantics of the transactional composition operator. Section 5 presents the experiment setting. Section 6 discusses the metrics used to evaluate the system and presents the results of our study in terms of size, coupling and complexity. Section 7 presents related work and Section 8 concludes this paper.

2. Case Study: Cellular Network

2.1 Cellular Network

The component used in the case study is a base station controller (BSC) of a cellular network. A mobile subscriber (MS) communicates with the network through a set of base stations (BTS), which are controlled by a BSC. Each base station provides coverage over a cell of the network. Adjacent base stations are grouped into paging groups. Data packets flow from the network to a base station through a series of routers (RTR) controlled by the BSC. The subscriber context is maintained in a database (CTXT). The communication between the network and the mobile subscriber is secured by an authentication protocol (AK). The BSC monitors network incoming data for a mobile subscriber through a proxy router (PROXY). The behavior of the system can be described according to two complementary decompositions: a feature-based decomposition and a scenario-based decomposition.

2.2 Base Station Controller Features

The behavior of the base station controller can be decomposed into the following features:

1. Lease Management (LSM): tracks which component of is currently handling the session with the MS.
2. Context Management (CX): handles the download and upload of context information between BTS and BSC
3. Authentication Keys (AK): handles the synchronization of the authentication keys between BTS and BSC.
4. Idle Mode Management (IMM): tracks when the subscriber enters or exits idle mode.
5. Handover (HO): handles the transfer of a session from a serving BTS to a target BTS.
6. Paging Control (PC): performs paging when inbound traffic is detected for a MS in idle model.
7. Proxy Monitoring (PY): monitors traffic intended for a MS when the MS is in idle mode
8. Router Control (RR): updates the routing tables when a handover is performed. The system supports three types of routing: nomadic (NRR), simple IP (SRR) and mobile IP router control (MRR).

2.3 Base Station Controller Scenarios

The requirements of the system are described using basic scenarios. Each scenario describes how the BSC responds to an input. These scenarios are organized as follows:

1. Lease Request (LREQ): a BTS requests control over a session or renews a session.
2. Lease Release Request (LREL): the subscriber enters idle mode and the BTS releases control over the lease
3. Idle Mode Request (IMREQ): the MS requests to enter idle mode.
4. Exit Idle Mode Request (XIM): the MS requests to exit idle mode.
5. Power Down Indication (PDIND): the subscriber device is powering down.
6. Context Download Request (CXDLD): a base station requests a copy of the subscriber context.
7. Reentry Complete Indication (REIND): a handover was successfully performed.
8. Location Update Request (LUREQ): a subscriber in idle mode has moved to a new paging group.
9. MIP Request and Verify (MIPRV): sent by the MS to maintain a mobile IP session with the network.

3. Orthogonal Regions

3.1 Introduction

We use orthogonal regions to modularize the implementation of the system features. Figure 1 presents a state machine composed of four regions, represented using the statechart notation. Each region implements a different feature of the cellular network component. The *LSM* region implements the lease management feature. The *IMM* region implements the idle mode management feature. The *PY* region implements the proxy monitoring region and the *RTR* region controls a router.

Each region is composed of a set of states and a set of transitions between these states. Transitions are executed in response to stimuli, which can be external events or internal events. The trigger of a transition is expressed using a logical expression over the occurrence of events. The execution of a transition corresponds to the execution of the actions of the transition. These actions are defined using statements and expressions. Transitions can also be guarded by logical expressions over the state of other regions. The guard *[IMM::ACT]* on the *LSM* transition triggered by the signal *LEASE_REQ* indicates that the transition is only enabled when the *IMM* region is in the state *ACT*.

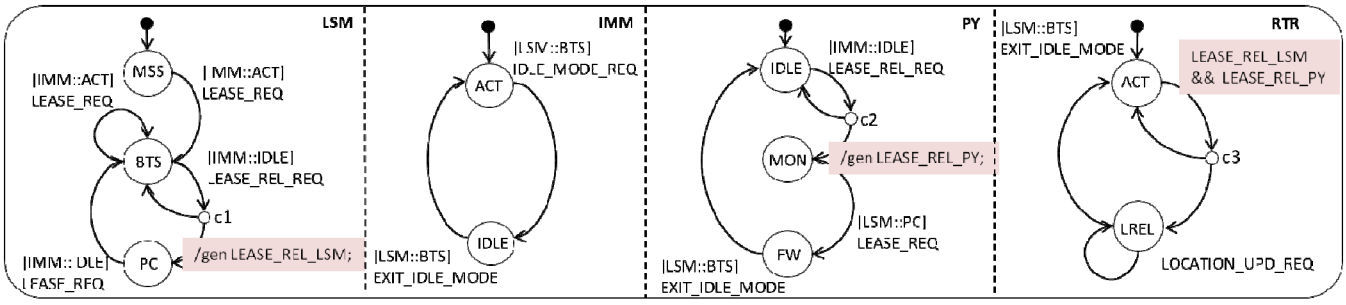

Figure 2. A state machine composed of 4 regions that implement different features of a telecom component.

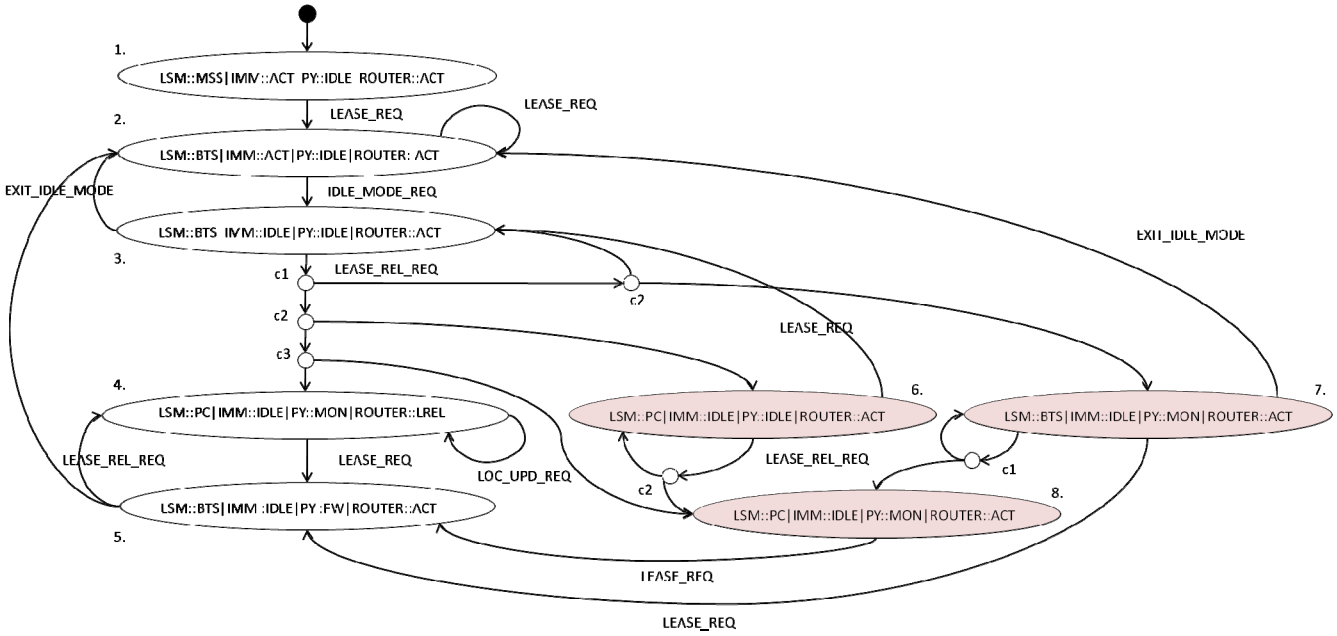

Figure 1. State machine obtained by flattening the 4 regions into a single region.

In Figure 1, the transition from state *BTS* to state *PC* in the *LSM* region is triggered when the external signal *LEASE_REL_REQ* is received by the state machine and the *IMM* region is in the *IDLE* state. The transition evaluates a conditional expression *c1* and executes the corresponding decision branch. Depending on the conditional, the transition either generates the internal event *LEASE_REL_LSM* (using the *gen* keyword) and steps into the *PC* state or it steps back into the *BTS* state. The *LEASE_REL_LSM* internal event triggers the transition from *ACT* to *LREL* in the *RTR* region if and only if the *LEASE_REL_PY* internal signal is generated during the same step of execution as indicated using the conjunction in the trigger of the transition.

Figure 2 shows a state diagram composed of a single region that was obtained by weaving together the regions of Figure 1. The states of the state machine of Figure 2 correspond to the reachable combination of the states of the regions. Such a combination is called a configuration. A state machine that consists of multiple regions executes according to steps between state configurations. A step corresponds to the execution of a set of enabled transitions from different regions.

In the example of Figure 1, the external signal *LEASE_REL_REQ* triggers a series of steps in the configuration *(BTS, IDLE, IDLE, ACT)*. The first step simultaneously executes the transition from *BTS* to *PC* in the *LSM* region and the transition from *IDLE* to *MON* in the *PY* region. Depending on the evaluation of the conditionals, the state machine steps into the configurations *(BTS, IDLE, MON, ACT), (PC, IDLE, IDLE, ACT), (PC, IDLE, MON, ACT)* or back into *(BTS, IDLE, IDLE, ACT)*. In case both events *LEASE_REL_LSM* and *LEASE_REL_PY*.are generated in the first step, a second step is triggered, which consist of the execution of the *ACT* to *LREL* transition in the *RTR* region. The state machine then reaches a quiescent state, where it has nothing left to executed The series of steps that are executed in response to the external event *LEASE_REL_REQ* until the state machine reaches a quiescent state is called a macro-step. The corresponding configurations are called macro-configurations. In the woven state machine of Figure 2, only the macro-configurations are represented as states.

The semantics of step execution are further detailed in the next section.

3.2 Operational Semantics or Regions

The semantics used in this paper are close to the semantics of classic statecharts as implemented in the Statemate tool [3]. Our approach follows the following guiding principles:

1. Determinism: Our semantics do not allow non-determinism to avoid differences of behavior between simulation and execution of the generated code on the target platform.
2. No shared variables: Our semantics do not allow the sharing of variables between regions. The sharing of variables between regions violates encapsulation and is a cause of non-determinism. Transitions that access the same variable in the same step can cause race conditions.
3. Causality: Internal events are only sensed in the step that follows the step in which they are generated. They do not persist after the following step.
4. Asynchrony: External signals received by the state machine are queued in a buffer. The state machine fetches a signal from the queue and executes until it reaches a quiescent state, after which it fetches the next signal from the queue. Timeout events are treated as external events and put in the same queue as external signals.

The semantics of our language make it more appropriate for distributed systems such as telecom systems rather than embedded real-time systems. However, the concepts presented in this paper are applicable to other statechart variants or other concurrent programming formalisms. To simplify this discussion, we do not consider hierarchy, state entry and exit actions or history. We do not discuss static reactions, spontaneous transitions or transition priorities.

Listing 1 defines the semantics of the state machine execution in Lisp pseudo-code. It is assumed that a data structure *sm* containing the topology of the state machine is available. A state machine executes steps until it terminates. Each step takes as input a status, which consists of a configuration and a set of events and produces a new status. The execution of a step consists of computing the set of enabled transitions for each region based on the current status, followed by the execution of the enabled transitions. If the set of events is empty, the state machine enters a quiescent state after which it fetches a new event from the queue.

The set of enabled transitions is computed by matching the trigger and the guards of each outgoing transition with respect to the current configuration and set of events. The set of triggers and guards are first normalized in the disjunctive normal form so that each element in the list is a conjunct of guards or triggers. The matching functions evaluate whether the set of events matches the expression of the trigger and whether the configuration matches the expression of the guards. The guard and trigger logical expression support conjunction, disjunction and negation. Conjunction of events means that all events are generated during the same step, as events only persist for a single step.

The execution of a step consists of executing all the enabled transitions. Each transition execution updates the status of the step by adding the events generated during the transition execution to the set of events and by updating the configuration based on the next state of the transition. The configuration is updated by replacing the region's previous state by the new state.

```
(defstruct status
           configuration events termination)

(defun execute-statemachine (sm)
 (let (status)
  (while (status.configuration != stop)
   (set status (execute-step sm status)))))

(defun execute-step (sm status)
 (when (status.events = ())
  (let ((event (pop-event-from-queue sm)))
   (when event
    (push event status.events))))
 (if (status.events != ())
  (let ((trs (get-enabled-transitions sm
                                      status)))
   (set status.events ())
   (set status (execute-transitions sm trs
                                    status)))
  else
   status))

(defun execute-transitions (sm trs status)
 (let ((ustatus status))
  (dolist (tr trs)
   (set ustatus (execute-transition sm tr
                                    ustatus)))
  (set status ustatus)))

(defun execute-transition (tr sm status)
  (set tr (bind-parameters-to-actuals tr status))
  (let (a)
   (while (set a (get-next-action a tr))
    (when (is-gen-action a)
     (push a.event status.events))
    (when (is-nextstate-action a)
     (set status.configuration
      (update-configuration sm
         status.configuration (get-nextstate a)))
    (when (is-stop-action a)
      (set status.configuration stop))
    (execute a)))
   status)
```

Listing 1. Statemachine execution semantics

```
(defun get-enabled-transitions (sm status)
 (let (enabled-transitions)
  (dolist (r sm.regions)
   (set enabled-transitions
      (append enabled-transitions
        (get-enabled-transition r sm status)))))

(defun get-enabled-transition (r sm status)
 (let (enabled-transitions)
  (dolist (tr (outgoing-transitions r
                        status.configuration))
   (dolist (g tr.guards)
    (when (match-guard g status.configuration)
     (dolist trig in tr.triggers
      (when (match-trigger (trig, status.events))
       (push tr enabled-transitions))))))
 enabled-transitions))
```

Listing 2. Computation of the set of enabled transitions

3.3 Checker Semantics

The state machine checker computes the set of macro-configurations of the system and detects problems with the region composition such as deadlocks. The checker algorithm can also be used to statically compose the regions into a single region by weaving together the actions of the region transitions, as illustrated in Figure 2.

The checker algorithm of Listing 3 performs a depth-first traversal on the reachable configurations by iterating over all possible external inputs and paths.

```
(defvar *macro-configs* ())

(defun check-statemachine (sm)
 (let (status)
  (traverse-statemachine sm status)))

(defun traverse-statemachine (sm status)
 (if (status.events = ())
  (when (not (find status.configuration
                   *macro-configs*))
   (push status.configuration *macro-configs*)
   (dolist (s *signals*)
    (push s status.events)
    (compute-step sm status))
  else
   (compute-step sm status)))

(defun compute-step (sm status)
 (let (new-statuses)
  (dolist (r sm.regions)
   (let ((tr (get-enabled-transition r sm
                                     status)))
    (when tr
      (set new-statuses (append new-statuses
           (collect-reachable-statuses r
                        (list status) tr))))))
  (dolist (st new-statuses)
   (traverse-statemachine sm st))))

(defun collect-reachable-statuses (r statuses tr)
 (dolist (a tr.actions)
  (when (is-gen-action a)
   (let (event (get-signal a)
    (dolist (st statuses)
     (when (not st.termination)
      (push event st.events)))))
  (when (is-nextstate-action a)
   (dolist (st statuses)
    (when (not st.termination )
     (set st.configuration
       (update-configuration st.configuration
                             (get-nextstate a)))
     (set st.termination t))))
  (when (is-stop-action a)
   (dolist (st statuses)
    (when (not st.termination)
     (set st.configuration stop)
     (set st.termination t))))
  (when (is-decision-action a)
   (let ((branches (get-decision-branches a))
         (continuation (get-continuation tr a))
    (dolist (br branches)
     (set br.actions (append br.actions
                             continuation.actions))
     (set statuses (append statuses
  (collect-reachable-statuses r statuses br))))))
```

Listing 3. Checker traversal of the state machine

The checker algorithm is exponential with respect to the number of paths and regions. In our experience, the exponential complexity has not proven to be a problem as systems are usually composed of a small set of coarse-grained regions. Hierarchy can also be used to reduce the number of regions if needed.

During the state machine traversal, configurations for which the set of internal events is empty are marked as macro-configuration. The algorithm then considers all possible external inputs (stored in **signals**) to drive the traversal further recursively. For each status, the algorithm collects all the statuses that are reachable trough the next step.

If the new statuses correspond to transient configurations, the next steps are computed. Otherwise, the configuration is a macro-configuration and the algorithm performs the next traversal. Conditional branches are expanded so that the algorithm can cover all possible combinations of target configurations and sets of generated events. In practice, the transitions are first normalized and caching is used to avoid the exponential blow up due to branching. The checker algorithm can detect the following situations:

1. Non-determinism exists in the system when the number of enabled transitions is larger than one within a region:

```
(when ((length enabled-transitions) > 1)
 (error "found non-determinism"))
```

2. A deadlock occurs when a macro-configuration does not have enabled-transitions for any external signal:

```
(dolist (mc *macro-configs*)
 (let ((trs ()))
  (dolist (s *signals)
   (let ((status (make-status mc s)))
    (set trs (append trs
              (get-enabled-transitions sm status)
  (when (trs = ())
   (error "found deadlock")))
```

3. An internal event is generated in one step but not consumed by a trigger. The event is lost. This can be detected by adding the following code to compute-step:

```
(let ((triggers ())
     (trs (get-enabled-transitions sm status)))
 (dolist (tr trs)
  (set triggers (append triggers tr.triggers)))
 (dolist (ev status.events)
  (when (not (find ev triggers))
   (warning "internal event ev is lost"))))
```

4. Managing Region Interactions

4.1 Region Interactions

The composition of Figure 1 generates macro-configurations that are inconsistent with the requirements. The configurations 6, 7 and 8 in Figure 2 violate the system consistency invariants. In our case study, releasing the lease consists of three tasks:

1. The lease management region checks if the request originates from the BTS that is currently holding the lease. If the request is valid, the region releases the lease to the paging controller (state *PC*).

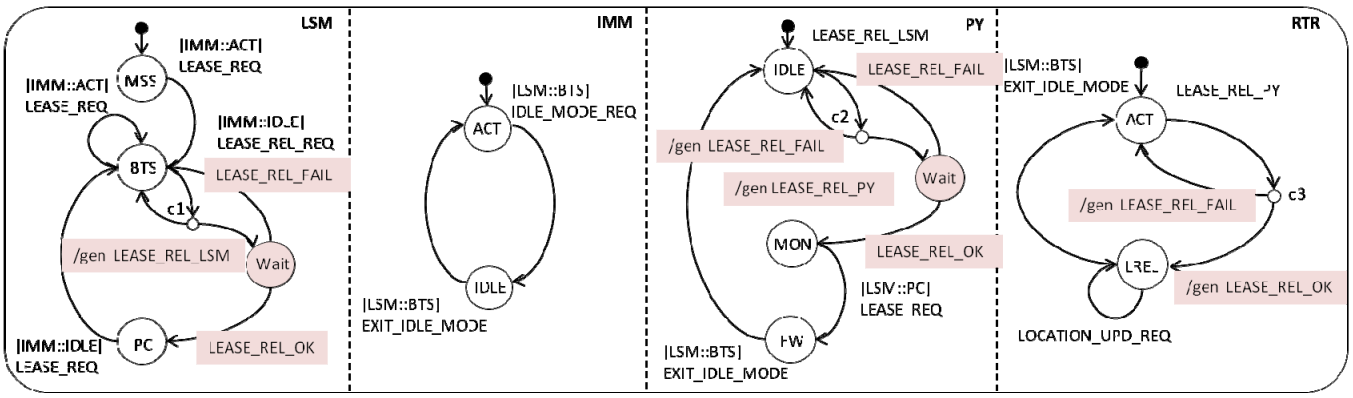

Figure 3. Interactions between the regions of a state machine

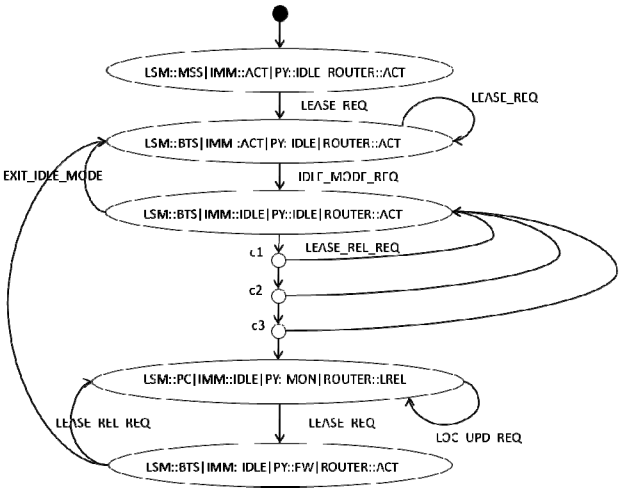

Figure 4. Flattened state machine generated from Figure 3

2. The proxy region is responsible for monitoring inbound traffic when the lease is released. If the request succeeds, the proxy region enters the monitoring state *MON*.
3. The router region needs to update the routing tables by sending an ARP request. If the ARP is sent successfully, the router region enters the *LREL* state.

Each of these tasks can be described and implemented independently of the other tasks. However, when the regions are composed with each other, the combinations of the branches in the different regions lead to inconsistent states. In configuration 6, the lease was released in the *LSM* region, but the proxy component failed to start monitoring the network. Inbound data will not trigger the paging procedure. In configuration 7, the proxy is monitoring the network traffic for inbound traffic, but the lease release operation failed in the *LSM* region. Incoming traffic will unnecessarily trigger the paging procedure. In configuration 8, the lease has been released and the proxy is monitoring the network. However, the routing table was not updated, and inbound traffic will not be received by the MS.

The composition requires synchronization to prevent inconsistencies. Figure 3 shows a version of the system where the region synchronization logic was modified to avoid inconsistent configurations. Figure 4 presents the resulting woven state machine which shows that the undesirable configurations are not reachable anymore.

In the solution of Figure 3, an order of execution between the participating regions was selected. The lease management region first determines whether it is able to process the request. If so, it propagates the request to the proxy region and enters a waiting state. The proxy region evaluates its conditional expression. If successful, it propagates the request to the router region and enters a waiting state. Otherwise, it notifies the lease management region. The router region sends back an acknowledgment if the request is successful or a failure signal if it fails. If successful, the *LSM*, *PY* and *RTR* regions will step into *(PC, MON, LREL)*. If unsuccessful, they will step back into *(BTS, IDL, ACT)*. The undesirable configurations are not reachable. However, the solution is not satisfactory with respect to:

1. Information hiding. The data carried by the signals is propagated from region to region. Each region has access to the entire payload of the signal
2. Coupling. The synchronization signals tightly couple the regions to each other. Adding a region to the interactions will require changes to the implementations of the other participant regions.
3. Cohesion. A synchronization state was introduced in two regions. These states do not correspond to logical states of the feature implemented by the region. Logic that was previously implemented in a single transition is now split over three transitions.
4. Termination. The synchronization structure requires that a *LEASE_REL_OK* or *LEASE_REL_FAIL* internal event is always generated in one of the steps that follow the regions entering a wait state or the system will enter a partial deadlock (one region enters a deadlock, while the others continue execution). It is the developer's responsibility to detect and prevent these situations.

4.2 Interface Regions

We introduce a design pattern to address the information hiding and coupling issues with the solution presented in Figure 3. The pattern uses an interface region *IF* to mediate between the interacting regions.

```
region IF {
 forstate S {
  [LSM::BTS && IMM::IDL
 && PY::IDL  && RTR::ACT]
  input LEASE_REL_REQ(lr){
   gen LEASE_REL_LSM(lr.a);
   gen LEASE_REL_PY( lr.b);
   gen LEASE_REL_RTR(lr.c);
   nextstate Wait; } }
 forstate Wait {
  input LEASE_REL_COMMIT(){
   output LEASE_REL_RSP(OK);
   nextstate S; }
  input LEASE_REL_ABORT() {
   output LEASE_REL_RSP(ER);
   nextstate S; } } ... }
```

```
region LSM {
 forstate BTS {
  input LEASE_REL_LSM(a){
   if (c1) {
    gen LEASE_REL_LSM_OK();
   } else {
   gen LEASE_REL_LSM_FAIL();
   }
   nextstate Wait; } }
 forstate Wait {
  input LEASE_REL_COMMIT(){
   nextstate PC; }
  input LEASE_REL_ABORT() {
   nextstate BTS; } } .. }
```

```
region PY {
 forstate IDL {
  input LEASE_REL_PY(b){
   if (c2) {
    gen LEASE_REL_PY_OK();
   } else {
    gen LEASE_REL_PY_FAIL();
   }
   nextstate Wait; } }
 forstate Wait {
  input LEASE_REL_COMMIT(){
   nextstate MON; }
  input LEASE_REL_ABORT() {
   nextstate IDL; } } .. }
```

```
region RTR {
 forstate ACT {
  input LEASE_REL_RTR(c){
   if (c3) {
    gen LEASE_REL_RTR_OK();
   } else {
   gen LEASE_REL_RTR_FAIL();
   }
   nextstate Wait; } }
 forstate Wait {
  input LEASE_REL_COMMIT(){
   nextstate MON; }
  input LEASE_REL_ABORT() {
   nextstate IDL; } } .. }
```

```
gen LEASE_REL_COMMIT() when LEASE_REL_LSM_OK()   && LEASE_REL_PY_OK()   && LEASE_REL_RTR_OK();
gen LEASE_REL_ABORT()  when LEASE_REL_LSM_FAIL() || LEASE_REL_PY_FAIL() || LEASE_REL_RTR_FAIL();
```

Listing 4. The lease release transition using an interface region

Listing 4 shows a textual representation of the lease release transitions in the *IF*, *LSM*, *PY* and *RTR* regions. The *IF* region interacts with the 3 other regions. However, The *LSM*, *PY* and *RTR* regions do not interact directly with each other. Listing 4 produces the same set of configurations as the state machine of Figure 3.

The *LSM*, *PY* and *RTR* implementation regions follow the same structure as the *IF* interface region. All regions include a synchronization state *Wait*, which separates the step that evaluates the conditionals from the step that executes the outcome of the transition. The interface region propagates the request to all participating implementation region, passing only the data that they need to proceed with the execution. Next, the implementation regions execute their decision step. The regions produce internal signals that indicate the outcome of the local decision phase. Finally, all regions execute the same global outcome.

The outcomes are defined using *gen* statements: if all three *OK* events are generated in the same step, the *COMMIT* outcomes will be executed in each region. If one of the *FAIL* events is generated, the *ABORT* outcome is executed. Gen statements are implemented through textual substitution of the triggers corresponding to the left hand side of the gen statement by the expression on its right hand side.

Listing 4 shows the general case where each regions produces different types of events. It is however not required that each region contributes to the mapping function of the *gen* statement or that they produce distinct types of events. The solution has the following advantages:

1. Information hiding. The interface region only propagates the parameters that are required by each region. Regions do not have access to data they do not need.
2. Coupling. The implementation regions do not interact or dependent on each other directly. All dependencies are between the interface regions and the implementation regions.

However, interface regions do not address the cohesion and termination issues highlighted in Section 4.1.

4.3 Transactional Regions

The management of region interactions is complicated by the termination issue. The function that maps local events to the different possible outcomes of the interaction needs to be deterministic and free of deadlocks. The set of outcomes needs to be complete: one of the outcomes must always be selected before the end of the macro-step. The synchronization states should never be part of a macro-configuration. This property cannot be checked automatically as the checker has no way of distinguishing synchronization states from other states. This constraint complicates maintenance and refinement of the system. During maintenance tasks, branches can be introduced in the regions that mistakenly omit to trigger decision events.

We therefore introduce a language construct to distinguish between state and transitions that correspond to macro-steps and states and transitions whose primary purpose is synchronization. Listing 5 shows how the *IF* and *LSM* regions are synchronized using the transaction construct. A transaction is a statement that has a name and a set of outcomes. In the example of Listing 5, the transaction *lrel* spans over the regions *IF* and *LSM*. The outcomes of the transactions are captured by a set of transitions with complementary triggers. The transaction can only complete in two ways: *LEASE_REL_COMMIT* is generated or *LEASE_REL_ABORT* is generated, based on the mapping function of the transaction expressed using a *gen* statement.

```
region IF {
 forstate S {
  [LSM::BTS]
  input LEASE_REL_REQ(lr){
   gen LEASE_REL_LSM(lr.a);
   transaction lrel {
    input LEASE_REL_COMMIT{
     output LEASE_REL_RSP1;
     nextstate S; }
    input LEASE_REL_ABORT {
     output LEASE_REL_RSP2;
     nextstate S; } } ... }
```

```
region LSM {
 forstate BTS {
  input LEASE_REL_LSM(a){
   if (c1) {
    gen LEASE_REL_LSM_OK; }
   else {
   gen LEASE_REL_LSM_FAIL;}
   transaction lrel {
    input LEASE_REL_COMMIT{
     nextstate PC; }
    input LEASE_REL_ABORT {
     nextstate BTS; } }..}
```

```
gen LEASE_REL_COMMIT() when LEASE_REL_LSM_OK();
gen LEASE_REL_ABORT()  when LEASE_REL_LSM_FAIL();
```

Listing 5. Transaction between *IF* and *LSM*

A transaction contains a set of outcomes. An outcome has the same syntax as the input part of a *forstate*, except that it cannot be guarded. Transactions with the same name in different regions need to have the same set of triggers for their outcomes. Transactions have the following properties:

1. Determinism. There should not exist a status for which the number of enabled outcomes is larger than 1.
2. Termination. One of the outcomes of a transaction is always executed before the completion of the macro-step in which the transaction started.

Transactions are implemented by translating the transaction statements into synchronization states, and by translating the outcomes into transitions. The algorithm to perform the translation is as follows. In each region:

1. Move the declarations used in the outcomes but declared in the context of the transition to the scope of the region and rename the declarations and references.
2. Generate a new synchronization state based on the name of the transaction, and annotate it as a generated synchronization state.
3. Replace the transaction statement by a *nextstate* action that steps into the synchronization state.
4. Generate a *forstate* for the synchronization state and add the outcomes of the transaction as input parts of the *forstate*.

This transformation is performed before running the checker. As the transaction outcomes are implemented as transitions, the checker automatically ensures that the outcomes are deterministic. The termination property is enforced by running the following code after the execution of the checker:

```
(dolist (mc *macro-configs*)
  (when (contains-synchronization-state mc)
    (error "synch state in macro-configuration")))
```

The check enforces that a synchronization state is never part of a macro-configuration. Its incoming transitions and outgoing transitions are always executed in the same macro-step. The transactional composition construct has the following advantages:

1. Cohesion. Actions that are executed during the same macro-step are syntactically located in the same transition. The notation avoids the need to introduce states whose purpose is to coordinate with other regions.
2. Termination. The semantics of transactions automatically detect partial deadlocks that would not be reported otherwise.

5. Case Study: Implementation

We evaluate the transactional composition by comparing three implementations of an industrial telecommunication component using modularity metrics. The first version of the system is a monolithic implementation. The second version uses classic regions to encapsulate the features of the system. The third version uses interface regions and transactional region composition.

5.1 Monolithic Implementation

The monolithic implementation was performed by a third party using a commercial modeling tool. The state machine is composed of a single region of 3 states: *Init*, *Idle*, and *Active*, corresponding to the idle mode feature. The states of the other features are encoded using flags. The model is structured according to the basic scenarios described in Section 2.3. The modeling tool used did not support regions or another mechanism to modularize features. The features described in Section 2.2 are therefore not modularized.

A typical transition in the monolithic implementation is represented in Listing 6. The actions corresponding to the different features have been annotated with a color code that corresponds to different features. The transition checks and updates the state of the LSM feature using the *itsLeaseState* flag at lines 3, 16 and 32. The state of the proxy feature is encoded using the *g_proxyEnabledFlag* at lines 20, 24, 36, 41, 45, 48 and 53. The state of the router feature is encoded using *the g_simIpSs* at line 9, the *g_MipFlag* at line 11 and the flags passed to the *sendFaARP* function at lines 14 and 30.

The transition has a complex control flow based on conditional expression over the return values of operations and flags that encode the state of the features. The implementations of the features are clearly tangled. It is hard to determine that the transition always sends a location update response back to the BTS. The router feature is not modularized and the three versions of the router feature are implemented within the transition.

5.2 Region-Based Implementation

The region-based implementation implements the same behavior as the monolithic implementation using regions. It was obtained through successive refactoring of the monolithic implementation. Each feature is implemented as a separate region. The states of the features are encoded using symbolic states rather than flags. The interactions between the regions are managed as in Figure 3. For each scenario that cuts across multiple regions, an order of execution was selected between the regions. Each region attempts to execute the request. If successful, it propagates the request to the next region and enters a synchronization state. Eventually, all regions handle the request successfully or a failure signal is generated.

5.3 Transactional Implementation

The transaction-based implementation was obtained by further refactoring of the region-based implementation. It uses interface regions to decouple the regions from each other and transactional composition to manage the synchronization between regions. Listing 7 shows the transitions of the interface, lease management, proxy and router regions that interact to implement the lease release request.

```
1.forstate Idle {
2. input M_LSM_MSS_LEASE_RELEASE_REQ(hdr_in, lrel) {
3.  if (itsLeaseState == BTS) {
4.   if (Ignore23bitAndCompare(lrel.BSId, g_BSID)) {
5.    switch (lrel.ReasonCode) {
6.     case Lease_Hold_Timer_expiration : {
7.      g_rc = sendProxyMonitorDataInd(lrel.SsIp);
8.      if (g_rc == MOB_OK) {
9.       switch (g_simIpSs) {
10.       case false : {
11.        if (g_MipFlag == MIP) {
12.         g_rc = sendFaLLC(); }
13.        else {
14.         g_rc = sendFaARP(ARP_WAIT_ONE); }
15.        if (g_rc == MOB_OK) {
16.         itsLeaseState = PC;
17.         if (! timeStampIsOld(lrel.TimeStamp)) {
18.          g_timestamp = lrel.TimeStamp; }
19.         reset (T_LEASE_DURATION());
20.         g_proxyEnabledFlag = true;
21.         sendLeaseReleaseResponse(g_MacAddr, OK); }
22.         nextstate -;
23.        else {
24.         g_proxyEnabledFlag = false;
25.         sendLeaseReleaseResponse(g_MacAddr, FAIL);
26.         nextstate -; } }
27.       case true : {
28.        g_rc = checkVlanIdInUseAndRouterIp();
29.        if (g_rc == MOB_OK) {
30.         g_rc = sendFaARP(ARP_RSP_WAIT);
31.         if (g_rc == MOB_OK) {
32.          itsLeaseState = PC;
33.          if (! timeStampIsOld(lrel.TimeStamp)) {
34.           g_timestamp = lrel.TimeStamp; }
35.          reset (T_LEASE_DURATION());
36.          g_proxyEnabledFlag = true;
37.          sendLeaseReleaseResponse(g_MacAddr, OK);
38.          nextstate -; }
39.         else {
40.        sendLeaseReleaseResponse(lrel.MacAddress,FAIL);
41.           sendProxyIdleModeRelease();
42.           stop; } }
43.        else {
44.        sendLeaseReleaseResponse(lrel.MacAddress,FAIL);
45.          sendProxyIdleModeRelease();
46.          stop; } } }
47.      else {
48.       g_proxyEnabledFlag = false;
49.       sendLeaseReleaseResponse(g_MacAddr, FAIL);
50.       nextstate -; } }
51.    case Idle_Mode_System_Timer_Expiration_at_BS : {
52.    sendLeaseReleaseResponse(lrel.MacAddress,FAIL);
53.     sendProxyIdleModeRelease();
54.     stop; } }
55.  else {
56.   sendLeaseReleaseResponse(lrel.MacAddress, FAIL);
57.   nextstate -; } }
58. else {
59.  sendLeaseReleaseResponse(lrel.MacAddress, FAIL);
60.  nextstate -; } } }
```

Listing 6. Lease Release - Monolithic implementation

First, the *IF* interface region defines the preconditions for processing the lease release request based on the state of the participant regions. Second, the request is propagated to the participant regions. Finally, the interface region sends back a response to the environment based on the outcome of the transaction. The region *IF* declares the outcomes of the transaction in terms of two types of exceptions: *LEASE_RELEASE_FAIL* and *LEASE_RELEASE_EXIT*. *LEASE_RELEASE_PROCEED* is executed when the *LEASE_RELEASE_OK* signal is generated and none of the exceptions are generated in the same step. The exception outcomes cover the cases where one exception is generated

```
1. gen LEASE_RELEASE_PROCEED() when  LEASE_RELEASE_OK()
2.     && ! LEASE_RELEASE_FAIL && ! LEASE_RELEASE_EXIT;
3. region IF {
4. forstate Idle {
5. [LSM::BTS && PROXY::Idle && ROUTER::Idle]
6.  input M_LSM_MSS_LEASE_RELEASE_REQ(hdr_in, lrel) {
7.   gen LEASE_RELEASE_LSM (lrel.BSId, lrel.ReasonCode,..
8.   gen LEASE_RELEASE_PROXY (lrel.SsIp);
9.   gen LEASE_RELEASE_ROUTER ();
10.  transaction lrelease {
11.   input LEASE_RELEASE_PROCEED() {
12.    sendLeaseReleaseResponse(lrel.MacAddress, OK);
13.    nextstate Idle; }
14.   input LEASE_RELEASE_FAIL()&& !LEASE_RELEASE_EXIT {
15.    sendLeaseReleaseResponse(lrel.MacAddress, FAIL);
16.    nextstate Idle; }
17.   input LEASE_RELEASE_EXIT() {
18.    sendLeaseReleaseResponse(lrel.MacAddress, FAIL);
19.    nextstate Idle; } }
20. [! (LSM::BTS && PROXY::Idle && ROUTER::Idle)]
21. input M_LSM_MSS_LEASE_RELEASE_REQ(hdr_in, lrel) {
22.  sendLeaseReleaseResponse(lrel.MacAddress, FAIL);
23.  nextstate Idle;  } } } }
24.region LSM {
25. forstate BTS {
26.  input LEASE_RELEASE_LSM(BSId, ReasonCode,TimeStamp){
27.   if (Ignore23bitAndCompare(BSId, g_BSID)) {
28.   switch (lrel.ReasonCode) {
29.     case Lease_Hold_Timer_expiration : {
30.      gen LEASE_RELEASE_OK ();      }
31.     case Idle_Mode_System_Timer_Expiration_at_BS : {
32.      gen LEASE_RELEASE_EXIT (); }
33.   else {
34.    gen LEASE_RELEASE_FAIL(); }
35.   transaction lrelease {
36.    input LEASE_RELEASE_PROCEED () {
37.     if (! timeStampIsOld(lrel.TimeStamp)) {
38.      g_timestamp = lrel.TimeStamp; }
39.      reset (T_LEASE_DURATION());
40.      nextstate PC; }
41.    input LEASE_RELEASE_FAIL() && !LEASE_RELEASE_EXIT{
42.      nextstate BTS; }
43.    input LEASE_RELEASE_EXIT () {
44.      stop; } } } }.. }
45.region PROXY {
46. forstate Idle {
47.  input LEASE_RELEASE_PROXY (SsIp) {
48.   g_rc = sendProxyMonitorDataInd(lrel.SsIp);
49.   if (g_rc != MOB_OK) {
50.    gen LEASE_RELEASE_FAIL(); }
51.   transaction lrelease {
52.    input LEASE_RELEASE_PROCEED() {
53.     nextstate MON; }
54.    input LEASE_RELEASE_FAIL() && !LEASE_RELEASE_EXIT{
55.     nextstate Idle; }
56.    input LEASE_RELEASE_EXIT() {
57.     sendProxyIdleModeRelease();
58.     nextstate Idle; } } } } .. }
59.region SIP_ROUTER {
60. forstate Idle {
61.  input LEASE_RELEASE_ROUTER () {
62.   g_rc = checkVlanIdInUseAndRouterIp();
63.   if (g_rc == MOB_OK) {
64.    g_rc = sendFaARP(ARP_RSP_WAIT);
65.    if (g_rc != MOB_OK) {
66.     gen LEASE_RELEASE_EXIT(); } }
67.   else {
68.    gen LEASE_RELEASE_EXIT(); }
69.   transaction lrelease {
70.    input LEASE_RELEASE_PROCEED() {
71.     nextstate LREL; }
72.    input LEASE_RELEASE_FAIL || LEASE_RELEASE_EXIT {
73.     nextstate Idle; } } } } .. }
```

Listing 7. Lease Release - Transactional implementation

but not the other and the case where both are generated in the same step.

6 Case Study Results

6.1 Metrics

The metrics used to evaluate the approach include:

- Lines of code of an entity or a feature of the system.
- Number of transitions and number of states of a region
- Coupling between transitions: the coupling index between transition *i* and transition *j* is 1 when transitions *i* and *j* access a shared variable or interact through an internal signal or a guard, 0 otherwise.
- Concern diffusion over transitions: the diffusion index between concern *i* and transition *j* is 1 if the transition implements part of concern *i*, 0 otherwise. The diffusion of a concern is the sum of its diffusion index with all transitions, divided by the number of transitions. It reflects the amount of scattering of its implementation.

6.2 Results

Table 1 compares the size of the 3 implementation in terms of lines of code (LOC), number of regions, number of states and number of transitions. All three implementations are about the same size in terms of lines of code. The region-based implementation is slightly smaller, due to the elimination of replication through the modularization of the features. This reduction is partly compensated by the additional structure of the new regions, states and transitions defined in the system. The transactional implementation is slightly larger is size, due to the interface region.

The region-based implementation contains many more states and transitions than the monolithic one. This is due to two factors. First, most of the basic scenarios introduce transitions in multiple regions. Second, the interactions between regions introduce many synchronization states. The synchronization causes behavior that is logically executed during the same macro-step to be split over multiple transitions.

The transactional implementation eliminates the waiting states and allows the behavior of one macro-step to be syntactically represented as a single cohesive transition. Compared to the region-based implementation, the transactions eliminate 45 states and 54 transitions. The transactional implementation is more structured than the monolithic one, but avoids the scattering of behavior over a large number of transitions by maintaining behavior that is executed in the same macro-step in the same transition. The transactional implementation is more cohesive than the region-based implementation.

	LOC	Regions	States	Transitions
Monolithic	2709	1	3	30
Region-based	2679	10	76	122
Transactional	2739	11	32	81

Table 1. Size of the three implementations

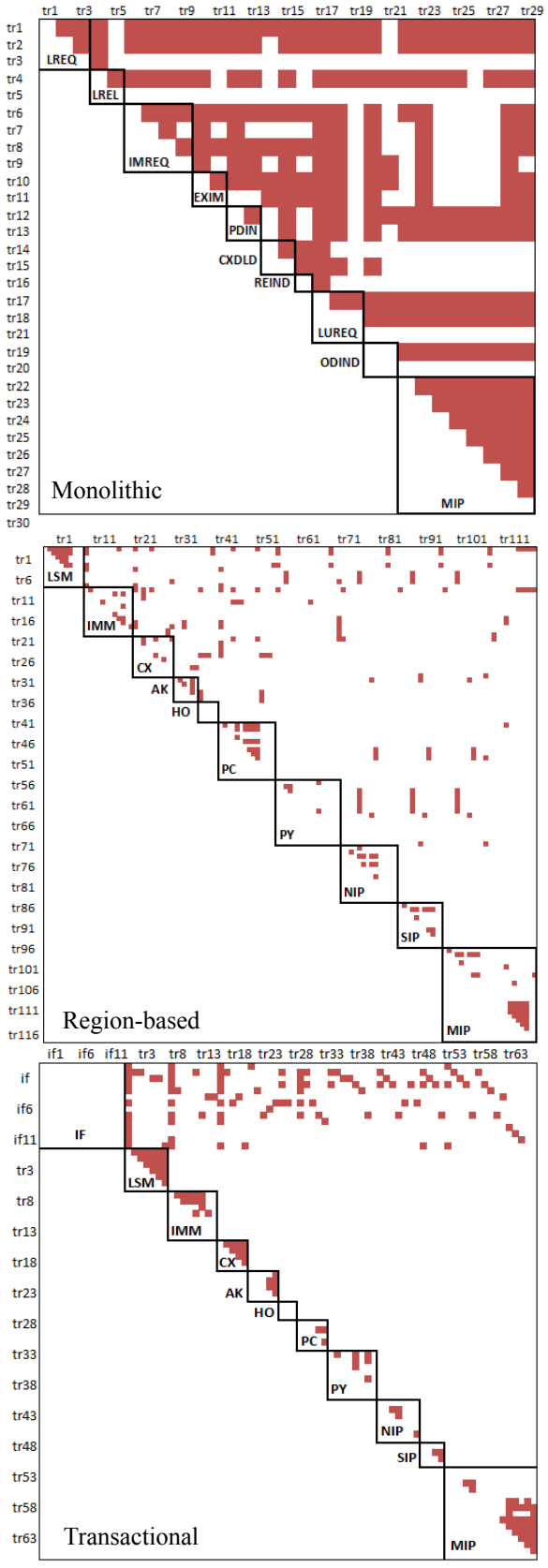

Figure 5. DSM's for the 3 implementations

	av.diff scenario	max.diff scenario	av.diff features	max.diff features
Monolithic	5.3%	6.7%	28.7%	60.0%
Region-based	6.8%	22.3%	10.0%	19.0%
Transactional	6.3%	16.2%	8.2%	18.5%

Table 2. Diffusion of the scenarios and features

Figure 5 compares the design structure matrices (DSM) [4] for the three implementations. The axes correspond to transitions of the state machine. The value of a matrix entry represents the coupling index between these transitions. The transitions are grouped according to the basic scenarios or the region they are part of by boxes along the diagonal.

The DSM of the monolithic implementation shows that the transitions that implement a scenario are tightly coupled with transitions that implement other scenarios. The scenarios cannot be implemented independently of each other.

The DSM for the region-based implementation is much sparser than the monolithic matrix, due to the large number of additional synchronization transitions. The number of coupling dependencies between regions is relatively smaller than the coupling dependencies between basic scenarios. However, there are still an important number of dependencies between regions, corresponding to guards and internal signal dependencies.

Finally, the DSM for the transactional implementation is denser than the region-based DSM, due to elimination of synchronization states. The DSM does not contain dependencies between the regions that implement the features of the system. All dependencies are concentrated on the interface region, at the top of the matrix.

Table 2 shows the average and maximum diffusions for the scenarios and the features of the three implementations of the system. A diffusion value of 5% means that the implementation of a scenario or a feature spreads over 5% of the transitions of the state machine. The average value indicates the average diffusion for all concerns. The maximum value indicates the diffusion of the concern that has the highest diffusion.

The results of Table 2 indicate that the implementations of the features are less scattered in the region-based and transactional implementations than in the monolithic implementation. The modularization of the features increases the diffusion of the scenarios, but in an acceptable manner. Compared to the region-based implementation, the transactional implementation reduces the diffusion of both the scenarios and the features. This is due to the transactional composition, which eliminate synchronization transitions.

6.3 Discussion

Our case study presents an example of a system that can be decomposed and understood in two complementary manners. When the basic scenarios are used as the primary decomposition, the reaction of the system to an external event is easy to understand: all the conditions to evaluate and actions to be performed are located in the body of one or two transitions. However, these scenarios cannot be implemented independently as they interact in complex ways trough shared variables and flags. The solution is also more rigid, as the features implemented by the system are scattered all over the transitions that implement the scenarios.

Feature-based decomposition increases flexibility. Different versions of the system that support different combinations of features can be delivered. In the case study, the isolation of the simple, nomadic and mobile IP router features allowed us to reduce the memory footprint of a session by 15%, by only loading the features required by the subscriber. The solution also gains in modularity, as the features can be implemented independently of each other using an interface region.

However, the interactions between the features can be hard to manage. The transactional composition semantics alleviate this problem by requiring that all features that interact within a basic-scenario share a common set of outcomes. The composition semantics ensure that synchronization problems within transactions will be detected.

The decomposition into features also makes the basic reaction of the system to an external event harder to understand. The actions executed in response to an event are scattered over the different regions. Yet, the reaction of the system can be understood in terms of a sequence of transactions in the interface region. The transactions provide a high-level, yet precise, view of the execution of each basic scenario and its possible outcomes.

The semantics of the transactional composition are more complex then the semantics used to perform a scenario-based decomposition. However, the transactional composition does not need to be understood by all developers. One of the main objectives of the decomposition by feature is to allow each feature to be implemented, tested and maintained by a different team. The team implementing a feature only needs to know about the outcomes defined in the interface region. It is then the role of the integration team to understand the interactions between the features and define the guards and the mapping between the events generated locally and the global outcomes.

The use of interface regions and transactions does not preclude the use of classic synchronization mechanisms. The decision to isolate features into regions and use interface regions should be guided by the need to enable the independent activation of a feature or the need to enable the independent development of the feature.

7 Related Work

Transactional region composition can be seen as a form of symmetric AOP [6]. Each region provides a view of the behavior of the system, seen from the perspective of a feature. Interface regions define design rules [4] or crosscutting interfaces [7]. They define the preconditions, triggers and post-conditions for transitions of the participating re-

gions. Transactions define an interface that cuts across the decomposition into features, and enables the system to support a decomposition into features and a decomposition into scenarios simultaneously.

In [8], Mussbacher considers the modularization and management of features within scenarios. The main decomposition is defined using scenarios but the language provides support to modularize features. Our approach uses features as the primary decomposition while providing language support to capture the essence of scenarios using transactions.

The work presented in this paper builds on a large body of research in the area of Aspect-Oriented Modeling [9]. In [10], Elrad identifies the similarities between aspect-oriented compositions and orthogonal regions and proposes to model aspect-oriented composition using internal signals between orthogonal regions. In [11], Mahoney proposes a notation to decouple regions by extracting internal signals from statecharts using an aspect-oriented notation. Zhang [12] introduces an aspect-oriented notation to modularize crosscutting concerns across different regions of UML state machines. Protocol modeling [13] proposes a semantic for the orthogonal composition of behaviors using a state machine-based notation based on CSP. Neither of these approaches achieves complete independence between the regions. The AOM approaches use small examples to illustrate the proposed syntax and do not address the issue of scalability and maintenance of large systems. Neither of the proposed approaches is evaluated quantitatively.

8 Conclusions

We show through an industrial case study that orthogonal regions can be used to modularize the features of a complex system. However, the interactions between the features of the system require a large amount of synchronization between the regions. The synchronization tightly couples the regions to each other and reduces the cohesion of each region. We therefore introduce a notion of interface region that decouples the regions from each other. We also introduce a transactional composition operator that reduces the diffusion of behavior caused by synchronization and facilitates the detection and management of feature interactions. We evaluate the approach by comparing a monolithic implementation of a real-world telecom system to a region-based implementation and a transactional implementation using size, coupling and diffusion metrics. Our results show that the transactional implementation is more modular and more cohesive than the monolithic and region-based implementations and that it supports the independent implementation and deployment of features.

References

[1] Bouma, L.G., Griffeth, N. and Kimbler, N. 2000. Feature Interactions in Telecommunications Systems. Computer Networks 32:4.

[2] Harel, D. 1987. Statecharts: A visual formalism for complex systems, Science of Computer Programming, Volume 8, Issue 3. 231-274.

[3] Harel, D. and Naamad, A. 1996. The STATEMATE Semantics of Statecharts, ACM Transactions on Software Engineering and Methodology (TOSEM), Volume 5, Issue 4. 293-333.

[4] Baldwin, C. and Clark, K. 2000. Design Rules vol I, The Power of Modularity. MIT Press.

[5] Garcia, A et al. 2005. Modularizing design patterns with aspects: a quantitative study. In Proceedings of the 4th international conference on Aspect-oriented software development, Chicago, USA. 3-14.

[6] Tarr, P., Ossher, H., Harrison, W. and Sutton, S. 1999. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In proceedings of the 21st international conference on Software engineering. Los Angeles, USA. 107-119.

[7] Kiczales, G. and Mezini, 2005. M. Aspect-oriented programming and modular reasoning. In proceedings of the 27th international conference on software engineering, St Louis, USA. 49-58.

[8] Mussbacher, G., Amyot, D., Weigert, T. and Cottenier, T. 2009. Feature Interactions in Aspect-Oriented Scenario Models. In proceedings of the 10th International Conference on Feature Interactions in Software and Communication Systems, Lisbon, Portugal. 75-90.

[9] Kienzle J. et al. Report of the 14th International Workshop on Aspect-Oriented Modeling. In Models in Software Engineering, LNCS 6002, 98-103.

[10] Elrad, T., Aldawud, O. and Bader, A. 2002. Aspect-Oriented Modeling: Bridging the Gap between Implementation and Design. In proceedings of the 1st conference on Generative Programming and Component Engineering, Pittsburgh, USA, LNCS 2487. 189-201.

[11] Mahoney, M., Bader, A., Aldawud, O. and Elrad, T.: 2004. Using Aspects to Abstract and Modularize Statecharts. The 5th Aspect-Oriented Modeling Workshop in Conjunction with the UML 2004 conference. Lisbon, Portugal.

[12] Zhang, G., Hölzl, M. and Knapp, A. 2007. Enhancing UML State Machines with Aspects. In proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems. Nashville, USA. LNCS 4735. 529-543.

[13] McNeile, A. and Roubtsova, E. 2010. Aspect-Oriented Development Using Protocol Modeling. In A Common Case Study for Aspect-Oriented Modeling Approaches, Transactions on Aspect Oriented Software Development, Volume 7. LNCS 6210. 115-150.