Modular Reasoning about Region Composition

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

Region composition is an operation where transitions of different automaton are woven together according to synchronization constraints. Reasoning about properties across regions is difficult, which is problematic in systems that are assembled by composing a large number of regions. We introduce two transactions constructs to enforce causality properties between transitions of a state machine. We show that transactions can be checked statically and that they support modular reasoning about region composition by preserving liveness properties within the scope of a transaction.

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

General Terms Design, Languages, Verification

Keywords Statecharts, Composition, Liveness

1. Introduction

Orthogonal regions are a language construct to modularize the implementation of different features of a system into separate automatons. Region composition consists of interleaving the transitions of the different regions according to a set of synchronization constraints. Although region composition mechanisms are less expressive than Aspect-Oriented Programming (AOP) language features, they share a certain number of characteristics. Regions that implement different features can react to the same events thereby implicitly weaving their behavior. The execution of a transition in one region can be guarded by the state of another region, thereby modifying its control flow.

We decompose a system by modularizing different features of the system into different regions. In systems that are composed of a large number of regions, the interactions between the regions make it difficult to reason in a modular way about the system. Modular reasoning about region composition means that we can understand some properties of the entire system by looking at the topology of a single region or a subset of regions.

In practice, debugging a system that is composed from multiple regions is a difficult task because it requires reasoning about the set of valid configurations of the system, which increases exponentially with the number of regions.

In this paper, we extend the statechart language to include two concepts of transactions. Transactions help reasoning about the properties of a state machine by establishing causality relations between different transitions within a region or across regions. We show that these relations can be checked statically and that they improve the ability to reason about liveness properties across regions.

2. Region Composition

2.1 Introduction

Statecharts [1] introduce 2 constructs to manage system complexity: hierarchy and orthogonality. Hierarchy is used to cluster and refine behaviors. Orthogonality allows different features of the system to be represented using orthogonal automatons which interact through synchronization signals. Figure 1 shows two regions *R1* and *R2* that interact through 3 types of interactions:

1. Common inputs: both regions respond to input *u*. When *u* is received in the state configuration *(B, Y)* or *(C, Y)*, region *R1* steps into state *A* and region *R2* steps into state *X* in a single step of execution. The order of execution between the transitions does not matter.
2. Internal events: an internal event *i* is generated in *R2* using the *gen* keyword. The signal is consumed in *R1* when it is in state *B*. When input *t* is received in configuration *(B, X)* and region *R2* steps into state *Y*, the event i is generated. This event is sensed in the next step of execution, causing region *R1* to step into state *C*.
3. Guards: Two transitions in region *R1* are guarded by the state *Y* of region *R2*, meaning that these transitions can only be triggered when region *R2* is in state *Y*. Guards are evaluated at the beginning of each step. If the guard evaluates to false, the input is discarded by the region.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FOAL'12, March 26, 2012, Potsdam, Germany.
Copyright 2012 ACM 978-1-4503-1099-4/12/03...$10.00.

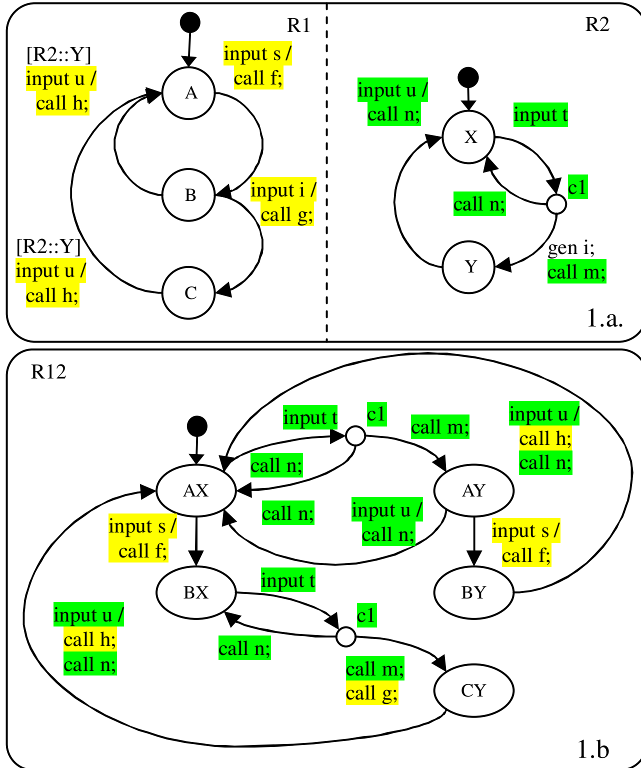

Figure 1. Composition of 2 orthogonal regions

2.2 Semantics

Figure 1.b shows a state diagram composed of a single region that was obtained by weaving together the regions of Figure 1.a. 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. The series of steps that are executed in response to an external event until the state machine reaches a quiescent state is called a macro-step. The corresponding configurations are called macro-configurations. Our approach follows the following guiding principles:

1. Determinism: our semantics do not allow non-determinism.
2. No shared variables between regions. The sharing of variables between regions 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.

A state machine is composed of an event queue and a state machine structure. The state machine structure is composed of a set of regions. Each region is composed of a set of states and a set of transitions. A transition is composed of a list of actions. Properties of the static structure of the state machine are expressed as follows:

- $A \in R_1$: R_1 contains state A
- *transition (for A input req action$_1$)* $\in R_1$: region R_1 contains the transition characterized by the source state A and the trigger *req*, along which *action$_1$* is executed.

Actions supported by the language include the following:

- Input action: an input action takes the form *'for' A '['* *g ']' 'input' req* '{' *action$_1$ .. action$_n$ '}'* where *A* is the name of a state of the region, *g* is a boolean expression, *req* is the name of a signal and *action$_1$ .. action$_n$* is a list of actions. The signal *req* is consumed when the region is in state *A* and the guard *g* is true.
- Nextstate action: a nextstate action takes the form '*nextstate' A ' ;'* where *A* is the name of a state of the region. The action makes the region step into state *A*.
- Stop action: a stop action takes the form *'stop' ';'*. A stop action terminates the state machine.
- Output action: an output action takes the form *'output' rsp ';'*, where *rsp* is the name of a signal. The action outputs the signal *rsp*.
- Gen action: a gen action takes the form '*gen' ev ';'*. The action generates the internal event *ev*.
- Call action: a call action takes the form '*call' fun ';'*. The action calls the function *fun*.
- Compound action: a compound action has the form *{ action$_1$ .. action$_n$ }* where *action$_1$ .. action$_n$* is a list of actions. A compound action executes the sequence of its actions.
- Decision action of the form '*switch' c '{' '['v$_1$']' action$_1$ .. '['v$_n$']' action$_n$ '}'* where *c* is an expression, *v_1 and v_n* are values to which the actions *action$_1$* and *action$_n$* are associated.

The state machine executes as defined by the lisp pseudo-code for the state machine interpreter Listing 1.

The *sm* structure contains the queue of external signals received by the state machine. The *sm* structure also contains the set of states and transitions of the state machine. The *status* structure maintains the current configuration of the state machine, as well as the list of events to be processed.

The *execute-statemachine* function contains the main loop of the system. The state machine will execute steps until a stop action is executed. Each step updates the status of the state machibe.

The *execute-step* function executes a single step of the state machine. If the list of internal events is empty, the state machine is in a quiescent state and will fetch the next external signal from the queue. If the queue is empty the state machine is blocked in the wait statement. When a new signal is received, the state machine is notified and it resumes its execution. Next, the *execute-step* function computes the set of enabled transitions based on the current configuration, the set of active events and the set of transitions of the state machine. Finally, all the enabled transitions are executed and the status of the state machine is updated by the *execute-transitions* function.

The *execute-transition* function executes a single transition. It updates the status of the state machine based on the actions executed along the transition. Nextstate and stop actions update the configuration, whereas gen actions add internal events to the set of active events to be considered in the next execution step.

When compiling the state machine program, a checker is used to compute the set of reachable macro-configurations of the system and to detect problems with the region composition such as deadlocks or non-determinism. The checker algorithm can also be adapted to statically compose the regions into a single region by weaving together the actions of the region transitions into macro-transitions between macro-configurations, as illustrated in Figure 1.b. The checker algorithm performs a depth-first traversal on the reachable configurations by iterating over all possible external inputs and paths. During the state machine traversal, configurations for which the set of internal events is empty are macro-configuration. The algorithm then considers all possible external inputs to drive the traversal further. We refer to [2][3][4] for a detailed description of statechart model checking.

```
(defstruct sm queue states transitions)
(defstruct status configuration events)
(defun execute-statemachine (sm)
 (let (status)
  (while (status.configuration != stop)
   (set status (execute-step sm status)))))
(defun execute-step (sm status)
 (while (status.events = ())
  (let ((event (pop-event-from-queue sm)))
   (if event
    (push event status.events)
    else
   (wait))))
 (let ((trs (get-enabled-transitions sm status))
      (events status.events))
  (set status.events ())
  (set status (execute-transitions sm trs
                                  status events))))
(defun execute-transitions (sm trs status events)
 (let ((ustatus status))
  (dolist (tr trs)
   (set ustatus (execute-transition sm tr
                                  ustatus events)))
   (set status ustatus)))
(defun execute-transition (tr sm status events)
 (let (a)
  (while (set a (get-next-action a tr))
   (when (is-input-action a)
    (set tr (bind-params-to-actuals a events))
   (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. State machine interpreter

3. Reasoning about Region Composition

3.1 Temporal Properties of Statecharts

In this section, we introduce a temporal logic notation to reason about the semantics of region composition. The notation was adapted from Linear Temporal Logic (LTL) to take into account concepts that are proper to the semantics of region composition, such as steps and macro-steps.

We correlate the topology of a region and the execution traces of the state machine by relating two levels of properties: transition properties and step properties.

Transition properties express temporal properties about the execution of actions within a transition. They are used to define relationships between actions executed within the scope of the same invocation of the *execute-transition* function. Executing a transition consists of executing a sequence of actions. Given an index *i* in a sequence of actions α of the same transition and a property *p*, we have:

$$(\alpha, i) \text{ satisfies } F_T\, p \text{ iff } \exists\, j \ge i,\ p(\alpha(j))$$

These operators are used to derive runtime temporal properties from the static structure of a transition. For example, we have:

$$\textit{transition (for A input req nextstate B)} \in R_1$$
$$\Rightarrow G(\textit{for } R_1 :: A \textit{ input req} \Rightarrow F_T(\textit{nextstate } R_1 :: B))$$

where G is the global invariant operator. A transition from state *A* triggered by the signal *req* in region R_1, during which nextstate *B* action is executed, implies that region R_1 will step into state *B* when signal *req* is received in state *A*.

Step properties express temporal properties about the traces of the composed system. It is not possible to reason about temporal properties of actions that are executed in different macro-steps without knowledge of the other entities the state machine communicates with. Transitions that are executed within the same step execute independently of each other and do not interact. Yet, we can reason about actions of transitions that execute in different steps of the same macro-step. Given an index *i* in a sequence of steps σ of the same macro-step and a property *p*, we have:

$$(\sigma, i) \text{ satisfies } F_S\, p \text{ iff } \exists\, j \ge i,\ p(\sigma(j))$$

Step properties are used to express causality relationships between transitions. The following relation expresses that when the transition from state *A* triggered by the signal *req* is executed in region *R1*, the transition from state *X* triggered by the event *i* and which executes the action output *rsp* is eventually executed in region *R2*, within the same macro-step.

$$G(\textit{transition(for R1::A input req)}$$
$$\Rightarrow F_S(\textit{transition(for R2::X input i output rsp)}))$$

We also define the next step operator X_S:

$$(\sigma, i) \text{ satisfies } X_S\, p \text{ iff } p(\sigma(i+1))$$

which indicates that a property holds in the next step of execution, within the same macro-step.

3.2 Liveness Properties of Regions

Region composition does not affect the control flow within transitions. We therefore have the property:

$$G(p \Rightarrow F_T(q)) \Rightarrow G(p \Rightarrow F_S(q))$$

which expresses that region composition preserves liveness properties within a transition.

However, reasoning about properties across regions is difficult. Figure 2 shows two regions that interact to produce a response *rsp* to the request *req*. In order to guarantee that region R_2 will respond to the internal event *i*, we need to ensure that R_2 will be in state *X* in the step that directly follows the step where internal event *i* is generated.

$$G(\,for\ R_1::A\ input\ req \Rightarrow X_S(R_2::X))$$

Reasoning about the state of the next step requires evaluating different possibilities: when the *req* signal is consumed by region R_1 either

1. Region R_2 is in state *X*, and there does not exist a transition from *X* to another state than *X*.
2. Region R_2 is in another state *S*, and it will always step into state *X* when the signal *req* is received

In terms of a state *S*, $S \neq X \in R_2$:

$$\begin{aligned} &G(\,for\ R_1::A\ input\ req \Rightarrow X_S(\,R_2::X)) \\ &\Leftrightarrow G(for\ R_1::A\ input\ req \Rightarrow \\ &\quad (R_2::S \wedge G(for R_2::S\ input\, req \Rightarrow F_T(nextstate R_2::X))) \\ &\quad \vee (R_2::X \wedge G(for R_2::X\ input\, req \Rightarrow F_T(nextstate R_2::X)))) \end{aligned}$$

If *transition(for X input req)* $\notin R_2$ and *transition(for * input req nextstate X)* $\notin R_2$, the condition simplifies to:

$$\begin{aligned} &G(\,for\ R_1::A\ input\ req \Rightarrow X_S(\,R_2::X)) \\ &\quad \Leftrightarrow G(for\ R_1::A\ input\ req \Rightarrow R_2::X) \end{aligned}$$

A guard *[R_2::X]* could be used to prevent the *req* signal from being consumed when the region R_2 is not ready to collaborate in the interaction. However, this means the request will be discarded by the system or saved for later processing depending on the guard semantics chosen. The language does not provide support to statically validate the composition between regions. If the interaction requires multiple steps, reasoning about the execution requires ensuring that all regions align to execute specific internal events. The complexity of the necessary conditions to guarantee liveness from the perspective of one region increases exponentially with the number of other regions it interacts with. Hence, reasoning about region composition is hard.

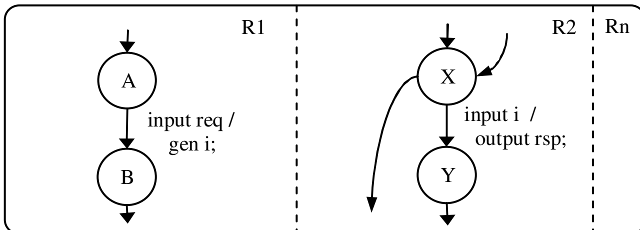

Figure 2. Interactions between regions

4. Transactional Regions

4.1 Closed Transactions

Closed transactions group a set of transitions of one or more regions, and enforce the following properties:

1. The transitions are always executed in the same macro-step
2. There exists a direct causality relation between the transitions through internal events.

Transitions of other regions can participate in the macro-step, but are not allowed to affect the control flow of the transaction. Figure 5 shows a transaction that spans over 2 regions and groups the transitions *transition(for A input req)* $\in R_1$ and *transition(for X input i output rsp)* $\in R_2$. The transaction ensures that the transitions have a direct causality relation and will always execute in the same macro-step. We can therefore derive liveness properties between actions of different regions, such as:

$$\frac{\begin{aligned} &G(\,transition(for\ R1::A\ input\ req\ gen\ i) \\ &\Rightarrow F_S(transition(for\ R2::X\ input\ i\ \ output\ rsp\ ))) \end{aligned}}{G(\,for\ R1::A\ input\ req \Rightarrow F_S(output\ rsp))}$$

When region R_1 is in state *A* and receives a request *req*, the state machine eventually sends a response *rsp* within the same execution macro-step.

Closed transactions are enforced by the state machine checker. During the traversal, the checker computes the set of all possible macro-configurations and macro-steps. It can therefore ensure that transitions that are part of the same transaction are always executed in the same macro-step. The direct causality property is enforced by considering each macro-step that contains the transaction and ensuring that all transitions are triggered by events generated within the transactions.

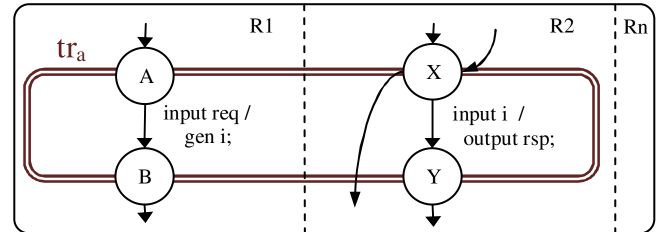

Figure 3. A closed transaction across regions

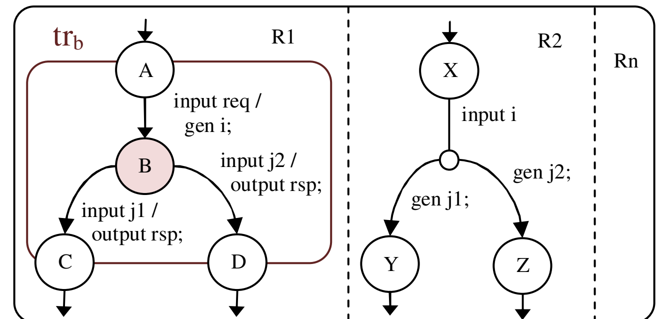

Figure 4. An open transaction within a region

4.2 Open Transactions

Open transactions group a set of states of one or more regions, and enforce the following properties:

1. The states are always entered within the same macro-step
2. The states are never part of a macro-configuration

Figure 4 shows a state machine where the state R_1::B is annotated as being part of a transition tr_b. As R_1::B is never part of a macro-configuration, the transition *transition (for A input req)* must always be executed always in the same macro-step as transition *transition (for B input j1)* or transition *transition (for C input j2).*

The transaction makes it possible to derive liveness property between actions of different transitions, such as

$$\frac{G(\ transition(for\ R_1::A\ input\ req\ nextstate\ B) \Rightarrow F_S(\ transition(for\ R_1::B\ input\ j1\ \ output\ rsp\ ))\ \vee\ transition(for\ R_1::B\ input\ j2\ \ output\ rsp\ )))}{G(\ for\ R_1::A\ input\ req \Rightarrow F_S(output\ rsp))}$$

Open transactions support modular reasoning about the properties of a state machine. Liveness properties within one region can be ensured without requiring knowledge about all regions that participate in the macro-step. The compiler will generate an error if there exists a path in the system for which *j1* or *j2* is not generated in the same macro-step. However, there is no guarantee that *j1* or *j2* will be generated by region R_2. Region R_2 can be replaced by another region that always generates the interval event *j1* or the event *j2*, making one of the branches unreachable.

Open transactions are enforced by the state machine checker. The checker computes the set of all reachable macro-configuration, and will ensure that states that are part of an open transaction are never part of a macro-configuration.

5. Related Work

This works builds on research on the verification of temporal properties of statecharts. [3] shows how statecharts can be checked for safety, liveness and existence properties using CTL. In [4], the authors show that temporal model checking of state charts is tractable on large systems.

Behavioral contracts for aspects [5][6] intend on controlling the effects of aspects on a base system by adding constraints to an interface. Transactions restrict the effects of region composition to modifications that preserve liveness properties, and can therefore be seen as type of contract. We are not aware of approaches that explicitly aim at preserving liveness properties of a base program. Verification of aspect-oriented programs techniques [7] could be used to generalize the transaction concepts to contracts for aspect-oriented programs.

6. Conclusions

Regions are a language construct to modularize the implementation of different features of a system into separate automatons. The automatons are composed according to a set of synchronization constraints. Reasoning about the correctness of region composition requires knowledge of the global state configuration of the system. Ensuring liveness properties across transitions is therefore difficult, as it requires evaluating all possible state combinations of the other regions. We introduce a concept of transaction which groups together transitions that should always be executed in the same macro-step. We show that this property can be checked statically and that it support modular reasoning about region composition, by preserving liveness properties across transitions within the scope of a transaction.

References

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

[2] Cottenier, T., van den Berg, A., Weigert, T. 2012. Management of Feature Interactions with Transactional Regions. In Proceedings of the International Conference on Aspect-Oriented Software Development, Postdam, Germany.

[3] Zhao, Q. and Krogh, B. 2006. Formal Verification of Statecharts Using Finite-State Model Checkers, IEEE Transactions on Control Systems Technology, 14:5. 943-950.

[4] Mikk, E., Lakhnech, Y., Siegel, M. and Holzmann, G. 1998. Implementing Statecharts in PROMELA/SPIN. In Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques. 90-101.

[5] Sullivan, K.J., Griswold, W.G., Rajan, H., Song, Y., Cai, Y., Shonle, M., Tewari, N. 2010. Modular aspect-oriented design with XPIs. ACM Transactions on Software Engineering and Methodology. 20:2

[6] Bagherzadeh, M., Rajan H., Leavens, G. and Mooney, H. 2010. Translucid Contracts: Expressive Specification and Modular Verification for Aspect-Oriented Interfaces, In Proceedings of the 10th International Conference on Aspect-Oriented Software Development, Porto de Galinhas, Brazil. 141-152.

[7] Krishnamurthi S. and Fisler K. 2007. Foundations of Incremental Aspect Model-Checking. ACM Transactions on Software Engineering and Methodology. 16: 2.