Synthesis of Poka Yoke Activity Diagrams

Version 0.1 (June 2024)

1 Introduction

2 Preliminaries

2.1 Formalisms

Our activity synthesis algorithm deals with various different formalisms that each have their own terminology. Here is a high-level overview.

Finite automata. Finite automata are finite directed graphs consisting of *locations* and *edges*. Locations can be marked *initial* when they are 'starting' locations. Locations can also be *marked* when they are accepting locations, with the standard meaning of acceptance from automata theory.

Extended finite automata (EFA). EFAs are finite automata that are used in the presence of *state*, i.e., variables that have a value. The edges of EFAs have *guards*, which is a state predicate, and *updates*, which is a state transformer.

(UML) Model. A UML Model may consist of a number of UML enum declaration, and consists of a single UML Class, which may contain:

- A number of UML properties, whose type is either Boolean, bounded integer or an enum as is defined in the UML model.
- A number of UML opaque behaviors that have a *guard* and zero or more *effects*. These opaque behaviors model *actions* to be performed in the synthesized activities. In case such an action has at most one effect we say the action is *deterministic*, and otherwise it is *nondeterministic*. The execution semantics of nondeterministic actions is that by executing the action, one of its effects is nondeterministically chosen and executed.
- A number of UML constraints that model the requirements for synthesis.

- An abstract UML activity, with preconditions, postconditions, and optimality constraints. Optimality contraints are roughly of the form 'action A must happen at least M times and at most N times' and thus limit the number of occurrences of some action in a to-be-synthesized activity. Abstract UML activities are 'empty' in the sense that they contain no nodes and no control flow.

We will only consider UML models that are *valid* with respect to the Poka Yoke validator (see the code implementation).

(UML) Activity. Activities are finite directed graphs consisting of *nodes* and *control flow*[1]. A node can either be a *control node* (initial, final, fork, join, decision, or merge node), or an *call opaque behavior node* which executes an action by calling an opaque behavior[2]. The goal of the activity synthesis algorithm is to synthesize (concrete) UML activities for the specified abstract UML activities in UML models, and updating the UML model by replacing these abstract activities by the concrete syntehsized ones.

Petri Net. A Petri Net is a finite directed graphs consisting of *places*, *transitions*, and *arcs*. Any place in a Petri Net can have at most one *token*. Transitions in a Petri Net can be *fired* to move tokens around between adjacent places. If some transition can fire, we say that it is *enabled*. Any arc in a Petri Net must be connecting a place with a transition. That is, there cannot be an arc from a place to some other place, or from a transition to some other transition. Further details on Petri Nets can be found in the standard literature.

2.2 Tooling and standards

2.3 Supervisory controller synthesis

2.4 Petri Net synthesis

3 Activity Synthesis

Algorithm 1 shows the high-level activity synthesis algorithm. This algorithm is a sequence of operations performed on some input UML model, *umlModel*, containing an abstract activity, to produce a UML model, $umlModel_{concr}$, where the abstract activity is replaced by a synthesized concrete activity. The remainder of this section explains these operations and their preconditions and guarantees.

[1]The UML2 metamodel uses ActivityNode and ActivityEdge, but here it may be better to talk about control flow rather than edges due to the possible ambiguity with automata edges.

[2]Technically there are other types of activity nodes that are supported by our formalim as well, like UML opaque actions and other types of call behavior nodes. But for the purpose of this document and understanding the synthesis algorithm, let us keep those out of scope.

Algorithm 1: Synthesis of Poka Yoke activity diagrams

```
1  procedure activity-synthesis(umlModel)
       // Synthesize a CIF supervisor using data-based synthesis.
2      cifModel := transform-uml-to-cif(umlModel);
3      cifSupervisor := data-based-synthesis(cifModel);
       // Generate the CIF state space as a minimal DFA.
4      cifStatespace := generate-statespace(cifSupervisor);
5      cifStatespace := ensure-single-source-and-sink(cifStatespace);
6      cifStatespace_rem := remove-state-annotations(cifStatespace);
7      cifStatespace_proj := event-based-projection(cifStatespace_rem);
8      cifStatespace_min := dfa-minimization(cifStatespace_proj);
       // Synthesize a minimal Petri Net.
9      (petriNet, regionMapping) :=
        petrinet-synthesis(cifStatespace_min);
       // Transform the Petri Net to an UML activity without edge guards.
10     umlActivity := transform-to-activity(petriNet);
       // Compute the edge guards for the UML decision nodes.
11     umlActivity_guards := compute-edge-guards(...);
       // Post-process the synthesized activity.
12     umlActivity_post := postprocess-activity(umlActivity_guards);
       // Replace abstract UML activity by the synthesized one.
13     umlModel_concr := replace-abstract-activity(umlActivity_post);
14     return umlModel_concr;
```

3.1 Operations

Activity synthesis uses the following operations.

3.1.1 Transforming UML to CIF

The `transform-uml-to-cif`(*umlModel*) operation translates a given UML model, *umlModel*, to a CIF specification to be used for synthesis. With respect to synthesis, the UML model specifies the plant (i.e., UML opaque behaviors) and requirements (i.e, UML constraints and pre/postconditions), which are translated one-to-one to CIF. This operation requires:

- The input UML model to be valid. See also Section 2.
- The input UML model to contain exactly one UML class.
- The single class within the UML model to have a classifier behavior that is an abstract activity, without nodes and control flow.

This operation produces a CIF specification that:

- For every UML enum declaration, contains a corresponding CIF enum declaration.

- Contains a CIF plant for the single UML class. This plant is a flower automaton, containing one location and only self-loops.
- Contains discrete variables for every UML class property. If a UML class property has a default value, then this value is translated as the default value of the CIF variable. If not, then the corresponding CIF variable is specified to have any value initially, with the 'in any' CIF construct.
- Contains CIF event declarations corresponding to all defined UML opaque behaviors. All opaque behaviors that model deterministic actions are translated as a single controllable event. All opaque behaviors that model non-deterministic actions are translated to multiple CIF events, namely a controllable one for starting the action, and uncontrollable ones for each of its non-deterministic effects to end the action. Such separate uncontrollable events must be defined, since data-based synthesis in CIF disallows controllable events that can happen non-deterministically. So, for data-based synthesis, we need a controllable event to (controllably) start some non-deterministic action, and uncontrollable events for the non-deterministic effects. Moreover, in case the UML model contains non-deterministic actions, an internal *atomicity variable* is declared and maintained in the CIF specification, to ensure that no event may occur between the start and end event of a non-deterministic action.
- Contains an 'initial' predicate from the conjunction of all translated preconditions of the classifier behavior activity of the single UML class. There may be multiple preconditions defined for this activity. Each of these preconditions are translated as Boolean algebraic variables in CIF, for better traceability. The conjunction of all these Boolean algebraic variables then forms the activity precondition, for which an algebraic variable is created as well. This algebraic variable is then used as the 'initial' predicate, which limits the number of initial states to only the ones satisfying the precondition.
- Contains a 'marked' predicate from the conjunction of all translated postconditions of the classifier behavior activity of the single UML class. There may be multiple postconditions defined for this activity. Each of these postconditions are translated as Boolean algebraic variables in CIF, for better traceability. The conjunction of all these Boolean algebraic variables then forms the activity precondition, for which an algebraic variable is created as well. Moreover, in case there are non-deterministic actions, an extra implicit postcondition is generated, stating that no non-deterministic action must be active (with respect to the atomicity variable explained earlier) for the postcondition to hold. This combined postcondition is then used as the 'marked' predicate.
- Contains requirement invariants stating that the postcondition disables any CIF event. In other words, if you reach a state where the activity postcondition holds, then no further actions have to be taken as they will

not contribute to coming closer to a postcondition state. These requirements can be seen as an optimization for synthesis and later state space generation, to avoid considering irrelevant actions/events.

- Contains an edge in the flower automaton plant for every defined UML opaque behavior. An edge is defined for all declared CIF events, both controllable and uncontrollable ones. All controllable CIF events correspond one-to-one to UML opaque behavior definitions, and thus their edge guards are the translated action guard. Moreover, in case of a deterministic action, the edge updates are the translated action effect. In case of a non-deterministic action, the action effects (plural, as in, more than one) are translated on the corresponding uncontrollable events instead. These edge guards and updates also correctly handle and maintain the atomicity variable, as explained before.

- Contains requirement automata for the optimality constraints defined in the UML model[3]. Optimality contraints are roughly of the form 'action A must happen at least M times and at most N times'. Such constraints are translated as requirement automata, containing a discrete variable that maintains how often the action has already occurred. This variable can then be incremented every time the action occurs, and via edge guards and marked predicates the requirement can be enforced.

3.1.2 Data-based synthesis

The operation `data-based-synthesis`(*cifModel*) executes the data-based supervisory controller synthesis tool that comes with ESCET/CIF. Data-based synthesis is thereby configured to do forward reachability (`--forward-reach=true`) for performance reasons, and to do no BDD predicate simplification (i.e., removing all simplifications from `--bdd-simplify`). The latter configuration is needed to be able to later compute edge guards.

The goal of performing data-based synthesis in our algorithm is to compute all extra restrictions on actions that are required for activities to never violate a specified requirement.

The requirements and guarantees of data-based synthesis are as described on https://eclipse.dev/escet/cif/synthesis-based-engineering.

3.1.3 State space generation

The operation `generate-statespace`(*cifSupervisor*) executes the CIF explorer tool that comes with ESCET/CIF. This tool is used to unfold the statespace of the synthesized supervisor, *cifSupervisor*. We need to have this statespace explicitly unfolded to be able to construct input for Petri Net synthesis, in order to later synthesize an activity.

The general requirements and guarantees of state space generation are as described on https://eclipse.dev/escet/cif/tools/explorer.html. The

[3]Note that optimality constraints are temporary, and intended to be removed.

CIF statespace will have state annotations, `@state(...)`, that indicate the values of all variables in every location. This information will later be used for computing edge guards in the synthesized activity.

Moreover, due to the way our UML/CIF input for synthesis is constructed, the resulting statespace has the following properties:

- All initial locations in the statespace correspond to states that satisfy the precondition of the to-be-synthesized activity.
- All marked locations in the statespace correspond to states that satisfy the postcondition of the to-be-synthesized activity.
- Marked locations do not have outgoing edges. This is because it does not make sense to perform further actions after the activity postcondition has been satisfied.
- The statespace is deadlock-free. That is, any path from any location in the statespace will either end up in a marked location, or will loop forever. In other words, the only locations from which no further edges can be taken are the marked locations.
- Unless the supervisor was empty (in which case the synthesis chain will have crashed already before generating the statespace), there is at least one initial location and at least one marked location.
- For every location is holds that all events on outgoing edges are either all controllable, or all uncontrollable. This property is a consequence of the atomic execution semantics of actions. If a nondeterministic action is being executed in some location in the CIF statespace, then by the atomicity constraint the only thing that could happen is an uncontrollable event to finish the atomic action. And conversely, if no nondeterministic action were being executed, then only controllable events can be executed to start some new action (assuming the location is not marked).

Note that state space generation could later become a performance bottleneck, e.g., in case there are many initial states. If this problem materializes in practical cases, then we could consider symbolic statespace generation instead of explicit statespace generation, and/or adapting the Petri Net synthesis algorithms to directly use these symbolic specifications.

3.1.4 Ensuring a single source and sink location

The operation `ensure-single-source-and-sink`(*cifStatespace*) transforms a given CIF statespace specification, *cifStatespace*, as described above, to ensure it has exactly one initial location and exactly one marked location.

Having exactly one initial location later helps, for synthesizing activities that must handle multiple initial states. We would like to synthesize activities that have exactly one initial node and exactly one final node (in order to keep the activities themselves, as well as their execution semantics, understandable).

However, it may happen that *cifStatespace* has multiple initial locations, for example when the activity precondition allows having more than one initial state. In such cases, we want the synthesized activity to have one initial node, and from there have a decision node that has outgoing edges for the multiple things that can happen. Thus, the single CIF initial location that is guaranteed by `ensure-single-source-and-sink` will directly correspond to the single initial node in the synthesized activity.

The situation is likewise for final locations. The single CIF final location that is guaranteed by `ensure-single-source-and-sink` will directly correspond to the single final node in the synthesized activity. In case the to-be-synthesized activity has multiple different ways to satisfy the activity postcondition, then this single final node will be preceded by a merge node.

As a precondition, `ensure-single-source-and-sink` requires the input CIF specification to:

- Not contain any CIF initialization predicates. (Which is guaranteed if *cifStatespace* is generated by the CIF explorer.)
- Not contain any CIF marker predicates. (Which is guaranteed if *cifStatespace* is generated by the CIF explorer.)
- Contain exacty one automaton, with an explicit alphabet.
- Not contain declarations/identifiers with the names '__init', '__done', '__start', or '__end'. These will be the names of the new initial (source) location, the new marked (sink) location, and the auxiliary events that connect these locations to the original initial/marked locations. More on this later.

The `ensure-single-source-and-sink` operation guarantees that:

- The resulting CIF specification has two new declared controllable events, '__start' and '__end', which have also been added to the alphabet of the automaton.
- The resulting CIF specification has a single initial location named '__init', even when it already had a single initial location. Moreover, auxiliary edges with event '__start' have been added to the automaton that go from the new '__init' location to the original initial locations. The original initial locations are now no longer initial.
- The resulting CIF specification has a single marked location named '__done', even when it already had a single marked location. Moreover, auxiliary edges with event '__end' have been added to the automaton that go from the original marked locations to the new '__done' location. The original marked locations are now no longer marked.
- Other than the extra initial/marked locations, the CIF statespace is unchanged.

3.1.5 Removing state annotations

The operation `remove-state-annotations`(*cifStatespace*) removes all state annotations, `@state(...)`, from locations in the given CIF statespace, *cifStatespace*, in which a nondeterministic action is being executed.

Recall that, during the UML-to-CIF conversion, all nondeterministic actions are 'split' into controllable and uncontrollable events, to start and end the action, respectively. Moreover, an atomicity constraint is imposed, stating that no action can happen while some other nondeterministic action is being executed. Of course, in the to-be-synthesized activity, we do not want to see such 'splitted' nondeterministic actions, but we want to see them as single nodes instead. Therefore, we need to get rid of all these uncontrollable events, so that by the time we do Petri Net synthesis, we just have single events for the nondeterministic actions. The 'getting rid of uncontrollable events' step is done by another operation, called event-based projection. However,