Synthesis of Poka Yoke Activity Diagrams

Version 0.1 (June 2024)

1 Introduction

2 Preliminaries

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.

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 edges.

This operation produces a CIF specification that:

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

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;
```

- 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[1]. 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

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

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. We need to have this statespace explicitly unfolded, to be able to construct input for Petri Net synthesis, in order to synthesize an activity.

The general requirements and guarantees of state space generation are as described on https://eclipse.dev/escet/cif/tools/explorer.html. Moreover, due to the way our UML/CIF input for synthesis is constructed, the resulting statespace has particular 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, all paths 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.

Note that state space generation could later become a performance bottleneck, e.g., in case there are many initial states. If this problem materialized in practical cases, then we could consider symbolic state space generation instead, 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 as described above, to ensure it has exactly one initial location and exactly one marked location.