Synthesis of Poka Yoke Activity Diagrams

July 30, 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 *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 are state predicates, as well as *updates* which are state transformers.

(UML) model. UML models consist of any number (i.e., zero or more) of UML enum declarations as well as exactly one UML class. An UML class contains:

- Any number of UML properties, whose type are either Boolean, bounded integer or enum.
- Any number of UML opaque behaviors that have exactly one *guard* and zero or more *effects*. These opaque behaviors model *actions* to be performed in synthesized activities. In case an action has at most one effect we say the action is *deterministic*, 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.

(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 connected to 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.

Note that this document explains the conceptual steps performed by the activity synthesis algorithm, rather than explaining what the code implementation is doing exactly. Therefore, the steps in Algorithm 1 may sometimes

[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      cifSpec := transform-uml-to-cif(umlModel);
3      cifSupervisor := data-based-synthesis(cifSpec);
       // Generate the CIF state space as a minimal DFA.
4      cifStatespace := generate-statespace(cifSupervisor);
5      cifStatespace := ensure-single-source-and-sink(cifStatespace);
6      cifStatespace_proj := event-based-projection(cifStatespace);
7      cifStatespace_min := dfa-minimization(cifStatespace_proj);
       // Synthesize a minimal Petri Net.
8      (petriNet, regionMapping) :=
        petrinet-synthesis(cifStatespace_min);
       // Transform the Petri Net to an UML activity without control flow
          guards.
9      umlModel_concr := transform-to-activity(petriNet, umlModel);
       // Compute the control flow guards for the UML decision nodes.
10     umlModel_guards := compute-edge-guards(umlModel_concr, ...);
       // Post-process the synthesized activity.
11     umlModel_post := postprocess-activity(umlModel_guards);
12     return umlModel_concr;
```

deviate slightly from how the code implementation is organized, and may be more abstract.

3.1 Operations

This section describes the operations performed by activity synthesis. A short explanation is given for every operation in Algorithm 1, followed by the motivation for having that operation, followed by its preconditions and postconditions.

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.

Motivation. With respect to synthesis, the UML model specifies the plant (i.e., UML opaque behaviors) and requirements (i.e, UML constraints). These are translated one-to-one to CIF, to enable data-based synthesis.

Preconditions. 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 UML activity, without nodes and control flow.

Postconditions. 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 single controllable events. All opaque behaviors that model nondeterministic actions are translated to multiple CIF events, namely a controllable one for starting the action, and uncontrollable ones for each of their nondeterministic effects to end the action. Such separate uncontrollable events must be defined since data-based synthesis in CIF disallows controllable events that can happen nondeterministically. So, for data-based synthesis, we need a controllable event to (controllably) start some nondeterministic action, and uncontrollable events to nondeterministically perform one of its effects. Moreover, in case the UML model contains nondeterministic 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 nondeterministic 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 nondeterministic actions, an

extra implicit postcondition is generated, stating that no nondeterministic 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 steps.

- 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 control flow 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 nondeterministic action, the action effects (plural, as in, more than one) are translated on the corresponding uncontrollable events instead. These control flow 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 control flow guards and marked predicates the requirement can be enforced.

3.1.2 Data-based synthesis

The operation `data-based-synthesis`(*cifSpec*) 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`).

Motivation. Our goal of performing data-based synthesis is to compute all extra restrictions on actions that must be considered by the to-be-synthesized activity to never violate a specified requirement. Data-based synthesis computes a minimally restrictive supervisor for going from a state that satisfies the activity precondition, to a state that satisfies the postcondition, without violating requiements, running into deadlocks, etc. We will later make the behavior of

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

this supervisor explicit (Section 3.1.3), to be able to synthesize a compact Petri Net for it (Section 3.1.7) that is then transformed to an activity (Section 3.1.8).

Once we have the UML activity structure at a later stage in the activity synthesis algorithm, we will need to compute the guards of the outgoing edges of its decision nodes (see Section 3.1.9). To be able to do that, we need to configure data-based synthesis to disable all BDD predicate simplifications. Otherwise, the extra synthesized conditions would be simplified with respect to specified requirements (and probably other things as well), so that the conditions become smaller while the requirements are explicit in the CIF supervisor. Instead, we want the synthesized conditions to completely capture all imposed restrictions, including requirements.

Forward reachability is configured simply for performance reasons, and might make synthesized conditions smaller. We should later evaluate if, and how much, forward reachability actually contributes to that.

Preconditions. All preconditions of the Data-Based Synthesis tool apply[4].

Postconditions. All guarantees of the Data-Based Synthesis tool apply. Moreover, since BDD predicate simplification is disabled, the resulting CIF supervisor contains no requirement invariants. These requirements are instead included in the synthesized conditions.

3.1.3 State space generation

The operation `generate-statespace`(*cifSupervisor*) executes the CIF Explorer tool that comes with Eclipse ESCET[5]. This tool unfolds the state space expressed by the given CIF specification.

Motivation. We need to explicitly unfold the (safe) state space of the synthesized supervisor to be able to construct input for Petri Net synthesis, in order to later synthesize an activity. The state space that is generated from the synthesized supervisor expresses all possible orderings of events, taking into account the synthesized guards and the original action guards as specified in the UML model. The goal of Petri Net synthesis is then to find a compact Petri Net representation of all these possible orderings, whose structure can then relatively easily be translated to a UML activity.

As a side remark; state space generation could later become a performance bottleneck, e.g., in case there are many initial states or large diamond patterns. If this problem materializes, then we could consider symbolic state space generation instead of explicit state space generation, and possibly adapting the Petri Net synthesis algorithms to directly use these symbolic specifications.

[4] See https://eclipse.dev/escet/cif/tools/datasynth.
[5] See https://eclipse.dev/escet/cif/tools/explorer.html.

Preconditions. Since state space generation uses the CIF Explorer tool that comes with Eclipse ESCET, all preconditions from that tool apply.

Postconditions. All guarantees of the CIF Explorer tool from Eclipse ESCET apply. Noteworthy is that the CIF state space will have state annotations, `@state(...)`, that indicate the values of all variables in every location. This information will later be used for computing control flow guards (see Section 3.1.9).

Moreover, due to the way our UML/CIF input for synthesis is constructed (e.g., by Section 3.1.1), the resulting state space has the following properties:

- All initial locations in the state space correspond to states that satisfy the precondition of the to-be-synthesized activity.
- All marked locations in the state space 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 state space is deadlock-free. That is, any path from any location in the state space will either end up in a marked location, or will loop. 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 activity synthesis chain will have crashed already before generating the state space), there is at least one initial location and at least one marked location.
- For every location it 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 state space, 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).

3.1.4 Ensuring a single source and sink location

The operation `ensure-single-source-and-sink`(*cifStatespace*) transforms the single automaton in the given CIF specification, *cifStatespace*, to ensure it has exactly one initial (source) location and exactly one marked (sink) location.

Motivation. Having exactly one initial location is required for the event-based projection and DFA minimization, described in Section 3.1.5 and Section 3.1.6.

Moreover, having exactly one initial location makes it easier to synthesize activities that must handle multiple initial states. To elaborate on that, 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 then 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 post-condition, then this single final node will be preceded by a merge node. So `ensure-single-source-and-sink`(...) ensures that the CIF specification already has the right structure with respect to that.

Preconditions. The input CIF specification is required to:

- Not contain any CIF initialization predicates nor any CIF marker predicates. (This is ensured 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.

Postconditions. This operation guarantees that:

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

3.1.5 Event-based projection

The operation `event-based-projection`(*cifStatespace*) projects the single automaton in the given CIF specification for all controllable events. This means that all uncontrollable events are projected away, i.e., *cifStatespace* is transformed to a DFA without keeping any uncontrollable events.

Motivation. Recall that uncontrollable CIF events were created for nondeterministic actions, to model the nondeterministic execution of their effects. However, these uncontrollable events and their corresponding edges are an internal, intermediate step that should not be visible in the synthesized UML activity. So at some point, these uncontrollable events have to be eliminated. This elimination should be done before Petri Net synthesis due to the atomicity constraint. To clarify this further, recall that the goal of Petri Net synthesis is to find a minimal Petri Net whose behavior is trace equivalent to the CIF state space that was given as input to Petri Net synthesis. While doing so, Petri Net synthesis aims to reduce diamond pattern in the state space to fork/join constructs in Petri Nets as much as possible. However, due to the atomicity constraint, the uncontrollable events do not give perfect diamond patterns, since whenever some nondeterministic action is being executed, the atomicity constraint enforces that no other action can be performed, thus impacting interleaving. Therefore, we perform event-based projection on the CIF state space, to restore the diamond patterns that got disrupted by the atomicity constraint, before doing Petri Net synthesis.

Note that Petrify, which is the tool we use for Petri Net synthesis, also has a built-in option `--hide` to hide a list of given events. In earlier versions of our implementation, we used this option instead of doing event-based projection on the level of CIF. However, Petrify's hiding option seems broken, in the sense that we observed that Petrify does not always hide all events in the specified list. This further motivates doing this on the level of CIF instead.

Preconditions. Since `event-based-projection`(...) uses the automaton projection tool that comes with Eclipse ESCET[6], all preconditions from that tool apply. Notably, the input should be a valid CIF specification (e.g., it does not accept automata where some locations have state annotations and some do not) that contains a single automaton, which in our case is the CIF state space. Moreover, this single automaton must have a single initial location.

Postconditions. All guarantees of the automaton projection tool from Eclipse ESCET apply. Notably, event-based projection guarantees that the resulting automaton after projection is a DFA that contains only the projected events, and that is language equivalent to the input specification with respect to those events. In our case, this means that the resulting CIF specification no longer

[6]See https://eclipse.dev/escet/cif/tools/eventbased/projection.html.

contains the uncontrollable events, and is language equivalent to the input modulo those events. The resulting DFA is not guaranteed to be minimal.

3.1.6 DFA minimization

The `dfa-minimization`($cifStatespace_{proj}$) operation minimizes the single deterministic automaton in the given CIF specification, $cifStatespace_{proj}$.

Motivation. We use DFA minimization to minimize the input for Petri Net synthesis, to avoid unnecessary work and possibly non-optimal results.

Moreover, note that Petrify, which is the tool we use for Petri Net synthesis, also has a built-in option `--mints` to minimize the input graph modulo trace equivelance. From the documentation of Petrify[7] this option seems to have some interaction with the `--hide` option. However, as explained in Section 3.1.5, `--hide` does not seem to always work. Therefore, in addition to event-based projection, we also do the minimization on the level of CIF.

Preconditions. Since `dfa-minimization`(...) uses the Event-based DFA minimization tool that comes with Eclipse ESCET[8], all preconditions from that tool apply. Notably, the input should be a CIF specification containing one deterministic automaton with a (single) initial location. In our case this is the CIF state space. Note that the previous steps of the activity synthesis algorithm ensure that the input we give to `dfa-minimization`(...) is deterministic and has exactly one initial location.

Postconditions. All guarantees of the Event-based DFA minimization tool from Eclipse ESCET apply. Notably, the result is a minimal DFA that has the same language as the input DFA. (Note that minimal DFAs are also unique in the sense that there cannot exist two different minimal DFAs with the same language. But at the moment we do not make use of this uniqueness property.)

The resulting automaton has exactly one initial location due to it being a DFA. Moreover, this operation preserves the property from Section 3.1.4 that there is exactly one marked (sink) location. To see why, suppose that event-based projection and/or DFA minimization would somehow have split-up the single marked location into multiple ones. Let us take two of them and refer to these locations as m_1 and m_2. Neither m_1 nor m_2 can have outgoing edges. But then there can be no word in the automaton language that would distinguish m_1 and m_2. Hence they must be the same location.

3.1.7 Petri Net synthesis

The operation `petrinet-synthesis`($cifStatespace_{min}$) performs Petri Net synthesis to compute a minimal free-choice Petri Net whose behavior is trace equiv-

[7] Which is included in the distribution that can be downloaded from https://www.cs.upc.edu/~jordicf/petrify/distrib.

[8] See https://eclipse.dev/escet/cif/tools/eventbased/dfa-minimize.html.

alent to the single automaton in the given CIF specification, *cifStatespace*$_{min}$. Moreover, this operation produces a *region mapping*, which is a mapping that relates the input specification *cifStatespace*$_{min}$ to the synthesized Petri Net.

We use the Petrify tool for performing Petri Net synthesis[9]. We thereby use the following options of Petrify: `-opt` to try to find the best possible result; `-fc` to synthesize a free-choice Petri Net; `-ip` to produce a Petri Net with intermediate places (otherwise certain places could be omitted to make the result a bit smaller for visualization purposes); and `-log` to generate a log file.

In our activity synthesis algorithm, we use PNML[10] as the intermediate format for representing Petri Nets. Therefore, part of the `petrinet-synthesis`(...) operation is to translate *cifStatespace*$_{min}$ to the input language of Petrify, and transforming the output of Petrify to PNML.

Moreover, although Petrify constructs a region mapping internally, there is no way to retrieve this region mapping by, e.g., some command-line option. The developers of Petrify recommended us to write a separate algorithm to 'recover' this region mapping, by co-simulating the input and output of Petrify to find out which CIF locations correspond to which Petri Net places. Therefore, the implementation of `petrinet-synthesis`(...) also requires recovering this region mapping, at least as long a Petrify is being used.

Motivation. By having computed a CIF state space, *cifStatespace*$_{min}$, we are still quite distant from an UML activity. The main reason is that concurrency can more concisely be represented in UML activities, via their fork and join nodes, with respect to state machines and automata. In contrast, *cifStatespace*$_{min}$ has all concurrent interleaving explicitly unfolded as diamond patterns. We now somehow have to detect all (diamond) patterns of concurrent interleaving in *cifStatespace*$_{min}$ and translate those to fork/join patterns in the to-be-synthesized activity. This is done by means of Peri Net synthesis, which is a field of research aiming to do exactly that, but then on Petri Nets rather than (UML) activities. Nevertheless, Petri Nets and activities are quite closely related, in the sense that the semantics of activities is usually defined as a Petri Net semantics. Moreover, there are particular classes of Petri Nets, in particular *free-choice* Petri Nets, that can relatively straightforwardly be translated to activities. Our main strategy is therefore to synthesize a minimal free-choice Petri Net from *cifStatespace*$_{min}$, and translate that to an UML activity. We thus use Petri Nets as an intermediate formalism in our activity synthesis algorithm.

However, translating a free-choice Petri Net to an UML activity is not yet sufficient: we then still have to compute the control flow guards for the decision nodes of the synthesized UML activity. This step must be done separately (see Section 3.1.9), since to the best of our knowledge, Petri Net synthesis with data/state is an open research field for which tooling is not available. Therefore, we must be able to relate the output of Petri Net synthesis to the input specification, *cifStatespace*$_{min}$. Such a relation is (or should be) produced

[9] See https://www.cs.upc.edu/~jordicf/petrify.
[10] See https://pnml.lip6.fr.

by the Petri Net synthesis algorithm, and is called a *region mapping*. The reason for this, is that Petri Net synthesis is based on the *theory of regions*. Without going too deep into this theory; the main idea is to group locations from $cifStatespace_{min}$ together so that every such group (roughly) corresponds to a Petri Net place. These groups are then called *regions*. So the Petri Net algorithm can produce a mapping from Petri Net places to the regions from which they are formed, thereby providing an input-output relation. With this relation, we can later find the relevant data/state and synthesized guards on the level of CIF, for the Petri Net places that will become UML decision nodes, and from those calculate the control flow guards. Section 3.1.9 explains this further.

Preconditions. Since Petrify is used for Petri Net synthesis, all preconditions from that tool apply. However, these preconditions do not seem to be well-documented. In any case the input CIF specification should not contain identifiers which are reserved keywords in the input formalism of Petrify. But since all Petrify reserved keywords seem to start with a dot, e.g., `.inputs` and `.graph`, this is (probably) already ensured as valid UML models do not contain identifiers containing a dot.

Moreover, the input CIF specification must contain exactly one automaton, i.e., the minimized CIF state space computed in the previous steps. This automaton must have an explicit alphabet that contains no duplicate events.

Furthermore, the event name `__reset` must not be in the automaton alphabet. This is because `petrinet-synthesis`(...) will create an auxiliary edge named `__reset` before Petrify is invoked, which is removed later from the output of Petrify. This extra `__reset` edge connects the single marked location to the single initial location, turning the input automaton into a big loop. This is needed by Petrify, since it seems unable to do Petri Net synthesis in case the input automaton has sink locations. After having invoked Petrify, the synthesized Petri Net will contain exactly one transition for `__reset` (since the input automaton has exactly one marked location, exactly one initial location, and we added exactly one `__reset` edge), so it's easy to remove it again.

Finally, the string `__to__` must not occur in any event names in the input automaton. This is later needed for transforming Petrify output to PNML, where `__to__` will be used as part of PNML arc identifiers.

Postconditions. All guarantees from Petrify apply, with respect to the options that we used. In particular, the output is an optimal free-choice Petri Net that is trace equivalent to the input automaton. The synthesized Petri Net is given in PNML format.

As explained above, the `__reset` event has been removed from this Petri Net. The synthesized Petri Net contains exactly one place that initially holds a token, which has no incoming arcs and exactly one outgoing arc to a transition named `__start`. The synthesized Petri Net contains exactly one sink place with no outgoing arcs and a single incoming arc from a transition named `__end`. These two places will later become the initial and final node of the synthesized activity.

Moreover, the synthesized Petri Net has been normalized (by relabeling all places). This is needed since Petrify seems able to produce nondeterministic results in the sense that running Petrify multiple times on the same model may give different results. We therefore normalize the Petri Nets, for example to allow proper regression testing.

Furthermore, a region mapping is calculated that indicates how Petri Net places correspond to locations in the input automaton. This region mapping is thus a mapping from Petri Net places to sets of automaton locations.

3.1.8 Transform Petri Net to activity

The operation `transform-to-activity`(*petriNet*, *umlModel*) transforms the given free-choice Petri Net *petriNet* in PNML format to a (concrete) UML activity, which replaces the abstract UML activity in *umlModel*, which is the original UML model that was given as input to the activity synthesis algorithm.

Motivation. The overall aim of the activity synthesis chain is to synthesize UML activities. Since activities are reasonably close to Petri Nets, we do this by first synthesizing a Petri Net using known techniques, and transforming this Petri Net to an activity using `transform-to-activity`(...).

Note that translating Petri Nets to activities in the general sense is non-trivial since Petri Nets may have patterns that are difficult to translate to an activity (in particular when the Petri Net has choice patterns where some choices have further token constraints). However, we use a particular kind of Petri Nets, namely free-choice ones, which are straightforwardly translatable to activities.

Preconditions. The input *petriNet* must be in PNML format. It must have exactly one place that has an initial token, which must additionally have no incoming arcs and exactly one outgoing arc to a transition named `__start`. It must have exactly one (sink) place that doesn't have any outgoing arcs, which must additionally have exactly one incoming arc from a transition named `__end`.

The input *umlModel* must be the original UML model that was given as input to the activity synthesis algorithm. See also the preconditions described in Section 3.1.1. Notably, it must have a single abstract UML activity.

Postconditions. The output is an UML model in which the abstract UML activity is replaced by a concrete one, which has been translated from *petriNet*. The concrete UML activity contains:

- Exactly one initial node and one final node, which have been translated from the two places described above in the preconditions section.
- Fork nodes, join nodes, decision nodes, and merge nodes, as translated from similar patterns in *petriNet*. For example, a Petri Net place with multiple incoming arcs and one outgoing arc is translated to an UML merge node.

- Call behavior nodes for all action nodes, that call the appropriate UML opaque behavior for the action.
- No guards on any control flows. Control flow guards are computed later, in Section 3.1.9.

3.1.9 Compute control flow guards

The operation `compute-edge-guards`($umlModel_{concr}$, ...) computes guards for all control flows in the single concretized UML activity in $umlModel_{concr}$. These guards are computed by using intermediate results from earlier steps in the activity synthesis algorithm (abbreviated here as ...), notably the extra conditions that have been synthesized earlier by data-based synthesis, as well as the CIF state annotations in the generated/projected/minimized CIF state space.

Motivation. At this point in the activity synthesis algorithm we managed to synthesize the structure of the concrete UML activity. This activity may have decision nodes. The execution of these decision nodes should follow one of its outgoing control flows, based on the current state of the system. However, the guards that determine which control flow should be followed are missing at this point. These choice guards were not yet computed since Petrify (or any other known algorithm/implementation to synthesize Petri Nets) cannot handle data. So we need to compute them separately, and add them to the UML activity. However, this is a non-trivial task that requires various intermediate steps.

Firstly, for every action defined in $umlModel_{concr}$ we need to know its action guard, as well as the condition that was synthesized for it during data-based synthesis (recall Section 3.1.2)[11]. The former can directly be obtained from the UML model. The latter can straightforwardly be obtained from the CIF supervisor, by looking up the synthesized condition for the controllable event that was defined for the action (recall Section 3.1.1).

Secondly, for every decision node in the synthesized UML activity, we need to know the set of all states in which the system may be at that point. These sets of states can be found by tracing back the CIF state annotations from the earlier generated CIF state space (see Section 3.1.3). So for every UML activity decision node[12] we must: (1) determine the Petri Net place corresponding to it, (2) use the region mapping produced in Section 3.1.7 to determine which CIF locations correspond to that Petri Net place, and (3) collect the state annotations from these locations in the generated CIF state space. Step (3) is slightly tricky in case there were non-deterministic actions, as we should not consider the state annotations of any 'intermediate' CIF locations in which a non-atomic action is being executed (e.g., since the atomicity variable is true in

[11] In fact, we only really need this for the actions that are directly connected to a decision node by a control flow.

[12] Note that we could do this for any UML activity node. But since we only need to compute guards for control flows out of decision nodes, we only do this for decision nodes.

any such intermediate state, and choice guards should not depend on internal details). Therefore, any such state annotations must be filtered out first.

We can now compute the choice guards with the information gathered so far. Given any control flow from some decision node D to some (call behavior) action node A, its guard is $\texttt{simplify}(\texttt{simplify}(A_{cond}, A_{guard}), D_{pred})$, where:

- A_{guard} is the guard predicate of A as defined in the input UML model.
- A_{cond} is the extra synthesized condition predicate for A.
- D_{pred} is the predicate describing all states the system may be in at the point of executing D.
- $\texttt{simplify}(p, q)$ is an operation that simplifies some given predicate p with respect to some predicate q, and returns the simplified predicate.

Note that all these predicates are represented in the code implementation as Binary Decision Diagrams (BDD). This is because BDDs allow easy manipulation of Boolean functions, in this case simplification of BDDs, i.e., `simplify`.

To elaborate further on guard computation; one might consider to simply use A_{cond} as the control flow guard, without doing any simplification. However, it is known that data-based synthesis may synthesize long control conditions that are difficult to read and understand by engineers. Moreover, A_{cond} may be expressed over internal state, like internal CIF variables for encoding optimality constraints that keep track of the number of occurrences of actions, which should not end up in control flow guards. It therefore makes sense to simplify A_{cond} by using any contextual information we have at hand. We can simplify A_{cond} with respect to the guard of A, since any restrictions covered by the guard of A are already considered by the execution semantics of activities. We can further simplify against D_{pred}, i.e, the knowledge of which states the system can be in while evaluating D, as it does not make sense for the control flow guard to restrict irrelevant system states. With these simplifications we can hopefully also eliminate any parts of the control flow guard that depends on internal state. We are not sure though whether such parts can actually always be eliminated.

Preconditions.

Postconditions.