Synthesis of Poka Yoke Activity Diagrams

February 14, 2025

1 Introduction

This document describes the Poka Yoke activity synthesis algorithm and discusses its internal steps. This document does not explain what the code implementation of the algorithm is doing exactly in full detail. Instead, it explains more abstractly what the different steps are conceptually, why they are needed, and what they require and ensure. Therefore, the explanation may sometimes deviate slightly from how the code implementation is organized, and may sometimes be more abstract. This document is intended to be a living document, to be extended continuously while further developing the synthesis algorithm.

The remainder of this document is organized as follows. Section 2 gives preliminaries for understanding the activity synthesis algorithm, in particular the formalisms and tools that are involved. Section 3 discusses the activity synthesis algorithm. First a high-level overview of the algorithm is given, and then its steps are discussed in more detail.

2 Preliminaries

2.1 Formalisms

The activity synthesis algorithm deals with various different formalisms that each have their own terminology. A high-level overview is presented below.

Finite automata. Finite automata are finite directed graphs consisting of *locations*, and *edges* that are labeled with an *event*. 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.

A deterministic finite automaton (DFA) is a finite automaton that is deterministic. Any finite automaton that is nondeterministic is a nondeterministic finite automaton (NFA).

Rather than a single DFA or NFA, we typically consider multiple automata that interact, e.g., by means of synchronizing events. We will not give a full introduction of all these concepts here.

Extended finite automata (EFA). EFAs are finite automata that are used in the presence of *data (properties)*, i.e., variables that have a value. The *(execution) state* of an EFA typically refers to the current valuation of all data properties, plus the locations that are currently active.

The edges of EFAs have *guards* which are state predicates, as well as *updates* which are state transformers. Semantically, an edge can be taken when its guard holds with respect to the state of the EFA are the current execution point, and this state is then updated (transformed) according to the edge update.

UML models and activities. For our purposes we use a subset of the UML2 metamodel[1]. In our restricted scope, UML models consist of any number of UML enum declarations, and exactly one UML class. A UML class contains:

- Any number of UML properties. UML properties have a type and optionally a default value. There are three supported types: Booleans, bounded integers, and enums. Default property values are then expressions of the appropriate type.
- 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 UML activities. Likewise to EFAs, action guards are state predicates, and action updates are state transformers, both of which are expressed over the UML properties as defined in the UML model. Moreover:
  - Any action is either deterministic or nondeterministic. An action is defined to be *deterministic* if the action has at most one effect. An action that is not deterministic is defined to be *nondeterministic*. The execution semantics of nondeterministic actions is that by executing the action, one of its effects is nondeterministically chosen and executed.
  - Any action is either atomic or nonatomic. The atomicity of an action can be indicated in UML Designer via a checkbox in the Properties view. If an action is indicated to be *atomic*, then no other actions can be performed during the execution of the atomic action. If an action is indicated to be *nonatomic*, then other actions may be performed in-between the start and end of the nonatomic action. A nonatomic action can only be started when its guard holds, and the effects of a nondeterministic action are applied when the action ends.
- Any number of UML constraints that model the requirements for synthesis. A UML constraint can be a state invariant that is expressed over the UML properties in the UML model. A UML constraint can

[1]See `https://www.omg.org/spec/UML/2.5.1` (Accessed 2024-08-28) for the full UML2 specification.

also be an action exclusion invariant, which indicates that the execution of some action requires the system to be in a certain state as expressed by a state invariant.

- Any number of UML activities, which can either be *abstract* or *concrete*.
  - Abstract UML activities are 'empty' in the sense that they contain no nodes nor control flows. An abstract UML activity consists of preconditions, postconditions, and occurrence constraints. The preconditions and postconditions are state invariants that are expressed over the specified UML properties. Occurrence constraints are roughly of the form 'A must happen at least M times and at most N times' where A can be either an action or an activity. Occurrence constraints thus limit the number of occurrences of some action, or calls to some activity, in a to-be-synthesized activity.
  - A concrete UML activity is a finite directed graph consisting of *nodes* and *control flows*[2]. A node can be a *control node* (initial, final, fork, join, decision, or merge node), be an *call opaque behavior node* which executes an action by calling an opaque behavior, or be an *opaque action node* containing a guard and zero or more effects, thereby essentially 'inlining' an action. So likewise to UML opaque behaviors, also opaque action nodes are either deterministic or nondeterministic, and either atomic or nonatomic.

Both abstract and concrete UML activities may contain preconditions, postconditions, and occurrence constraints. The preconditions and postconditions are state invariants that are expressed over the specified UML properties. Occurrence constraints are roughly of the form 'A must happen at least M times and at most N times' where A can be either an action or an activity. Occurrence constraints thus limit the number of occurrences of some action, or calls to some activity, in a to-be-synthesized activity. In the case of abstract activities, these constraints are used as input for synthesizing a concrete activity that adheres to these constraints. If an abstract activity is allowed to call a concrete activity by its occurrence constraints, then the preconditions and postconditions of that concrete activity are used by the synthesis algorithm to determine when the concrete activity can be called by the to-be-synthesized activity, and once called, when the concrete activity has finished its execution. (Moreover, in principle it would also be possible to later use the preconditions and postconditions of concrete activities for verification, e.g., property checking.) Note that occurrence constraints are temporary constructs and are intended to be removed later, which is also future work.

[2]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.

The activity synthesis algorithm takes UML models as input, containing any number of abstract UML activities. The goal of activity synthesis is then to synthesize a concrete UML activity for every abstract UML activity, and update the UML model by replacing all abstract UML activities by the concrete synthesized ones.

We will only consider UML models that are *valid* with respect to the Poka Yoke validator. For example, UML models should not contain abstract UML activities that have nodes or control flows, and should not use double underscores '__' in any names of UML elements.

Petri Nets. A Petri Net is a finite directed graph consisting of *places*, *transitions*, and *arcs*. We consider *elementary Petri Nets*, where (by definition) any place holds 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 and their semantics can be found in the standard literature.

The activity synthesis algorithm may use a particular category of Petri Nets, namely *free-choice Petri Nets*. This is because we have observed that free-choice Petri Nets can be translated to concrete UML activities that are more intuitive for users to understand than general Petri Nets. Intuitively, free-choice means that all choices (i.e., places with multiple outgoing arcs) can be made without additional constraints (i.e., the target transitions of these outgoing arcs do not have additional token requirements). Free-choice Petri Nets thus have simpler choice patterns compared to general Petri Nets, which may translate to more intuitive UML activities.

2.2 Standard and tooling

The activity synthesis algorithm deals with various different standards and tools. A brief overview is given below.

CIF. CIF is a specification language for discrete event systems (as well as timed and hybrid ones), as well as a toolset that supports the development process of supervisory controllers. CIF is part of the Eclipse ESCET toolkit, see https://eclipse.dev/escet (accessed 2024-07-31) for further details. We use the data-based synthesis tool of CIF to do supervisory controller synthesis, and the CIF explorer tool for state space generation.

Petrify. Petrify is a tool for Petri Net synthesis. Given some state machine (or finite automaton), the goal of Petri Net synthesis is to synthesize a (minimal) Petri Net that is trace-equivalent to the input state machine. Petrify has its own input and output specification language. We use Petrify to synthesize a Petri Net from a state machine, as a stepping stone for synthesizing UML activities. This is because Petri Nets can more compactly represent concurrency, by forking/joining from a Petri Net transition. In contrast, in state machines, concurrency is represented by explicit interleaving, i.e., as diamond patterns. Petri Net synthesis can automatically turn such diamond patterns in the input state machine to more compact fork/join patterns in the resulting Petri Net. Further information on Petrify can be found here: `https://www.cs.upc.edu/~jordicf/petrify` (accessed 2024-07-31).

PNML. PNML stands for Petri Net Markup Language, and is an XML-based syntax for representing Petri Nets. We use the PNML metamodel to represent Petri Nets internally in the code implementation. Further information can be found here: `https://pnml.lip6.fr` (accessed 2024-07-31).

UML. We use the UML metamodel to represent UML models and UML activities in the code implementation. Further information can be found here: `https://wiki.eclipse.org/MDT-UML2` (accessed 2024-07-31).

2.3 Algorithms for synthesis

The activity synthesis procedure internally uses two types of synthesis algorithms, namely supervisory controller synthesis (currently using CIF), and Petri Net synthesis (currently using Petrify).

Supervisory controller synthesis is used to compute a minimally restrictive supervisor, which is essentially a state machine that describes all safe system behavior. After that, Petri Net synthesis is used to synthesize a (possibly free-choice) Petri Net from this state machine, which is then transformed to a concrete UML activity. The intermediate step of synthesizing a Petri Net is performed since Petri Nets are structurally much closer to UML activities compared to state machines, and thus easier to translate.

The field of Petri Net synthesis is developed around the *theory of regions*. Intuitively, the idea is to find groups of locations in the input state machine that can be turned into single places in the output Petri Net. These groups are then called regions. Depending on the type of Petri Net you aim to synthesize (e.g., general or free-choice ones), there are slightly different requirements of what exactly constitutes a region. Petrify is essentially an implementation of the theory of regions, and is able to synthesize both general and free-choice Petri Nets (among other kinds of Petri Nets).

We will not explain data-based synthesis and Petri Net synthesis here. More information on these types of syntheses can be found on the webpages of Eclipse ESCET and Petrify linked above.

3 Activity Synthesis

Algorithm 1 shows the high-level activity synthesis algorithm. The entry point is `synthesize-all`, which iterates over all abstract UML activities in the given input UML model, synthesizes a concrete UML activity for them, and replaces the abstract activity by the concrete synthesized one. The operation `get-ordered-abstr-activities` gives the list of all abstract activities in the input UML model, in the order in which they are to be synthesized. Such an ordering is needed to enable synthesizing activities that can call other activities, i.e., to support synthesis of hierarchical specifications. If some activity depends on some other activity with respect to the hierarchy, we should synthesize the other activity first. An abstract activity depends on all abstract and concrete activities in scope, unless expressed otherwise via occurrence constraints. In particular, if there is an occurrence constraint expressing that some activity may not be called (i.e., called at most zero times), then the abstract activity will not depend on that activity. In case any cycles are detected in these dependencies, then the synthesis algorithm will terminate and an error will be reported.

The algorithm for synthesizing a single UML activity, `synthesize-single`, is a sequence of operations performed on the input UML abstract activity. The remainder of this section explains all operations in `synthesize-single`. A short explanation is given for every operation, followed by the motivation for having that operation, followed by its preconditions and postconditions.

Example (Running Example)**.** Let us consider a simple UML model, named `BitFlipper`, where we aim at synthesizing an activity that flips a single nondeterministically initialized boolean value. The UML model is composed of a single class `ActiveClass`, which contains two properties: a boolean named `init`, initialized as false, denoting whether the system is initialized, and a boolean named `bit`, not initialized. The class also contains an abstract activity called `ActivityMain`, and two opaque behaviors named `flip` and `initialize`. The activity has precondition `not init`, and postcondition `bit and init`. The `flip` opaque behavior's effect is to flip the boolean `bit` after it is initialized: its guard is `init`, whilst its effects consist of a single assignment `bit := not bit`. The opaque behavior `initialize` turns `init` from false to true, and instantiates `bit` either as true of false in a nondeterministic fashion. The model outline in the PokaYoke UML Designer environment is depicted in Fig. 1.

3.1 Transforming the abstract UML activity to CIF

The `transform-uml-to-cif`($umlActivity_{abstr}$) operation translates a given abstract UML activity, $umlActivity_{abstr}$, together with all relevant context from the UML model (e.g., UML class properties, constraints, opaque behaviors, concrete activities that may be called, etc.) to a CIF specification to be used for supervisory controller synthesis.

Algorithm 1: Synthesis of Poka Yoke activity diagrams

```
   // Synthesizes a concrete activity for every abstract one in the model.
1  procedure synthesize-all($umlModel$)
2      $umlActivities_{abstr}$ := get-ordered-abstr-activities($umlModel$);
3      for every activity $umlActivity_{abstr}$ in $umlActivities_{abstr}$ do
4          $umlActivity_{concr}$ := synthesize-single($umlActivity_{abstr}$);
5          replace $umlActivity_{abstr}$ by $umlActivity_{concr}$ in $umlModel$;
6      end

   // Synthesizes a concrete activity for the given abstract activity.
7  procedure synthesize-single($umlActivity_{abstr}$)
       // Synthesize a CIF supervisor using data-based synthesis.
8      $cifSpec$ := transform-uml-to-cif($umlActivity_{abstr}$);
9      $cifSupervisor$ := data-based-synthesis($cifSpec$);
       // Generate the CIF state space as a minimal DFA.
10     $cifStatespace$ := generate-statespace($cifSupervisor$);
11     $cifStatespace$ := ensure-single-source-and-sink($cifStatespace$);
12     $cifStatespace_{proj}$ := event-based-projection($cifStatespace$);
13     $cifStatespace_{min}$ := dfa-minimization($cifStatespace_{proj}$);
       // Synthesize a minimal Petri Net.
14     ($petriNet$, $regionMapping$) :=
        petrinet-synthesis($cifStatespace_{min}$);
       // Transform the Petri Net to an activity without control flow guards.
15     $umlActivity_{concr}$ := transform-to-activity($petriNet$);
16     $umlActivity_{red}$ := reduce-non-atomic-patterns($umlActivity_{concr}$);
       // Compute the control flow guards for the UML decision nodes.
17     $umlActivity_{guards}$ := compute-edge-guards($umlActivity_{red}, \ldots$);
       // Post-process the synthesized UML activity.
18     $umlActivity_{post}$ := postprocess-activity($umlActivity_{guards}$);
19     return $umlActivity_{post}$;
```

Motivation. With respect to supervisory controller synthesis, the UML model of the abstract input activity $umlActivity_{abstr}$ specifies the plant and requirements, i.e., the UML opaque behaviors and UML constraints. Moreover, the abstract activity itself contains the name of the to-be-synthesized activity as well as its preconditions, postconditions, and occurrence constraints. These plant and requirement constructs are translated one-to-one to CIF, to enable running its data-based synthesis tool. In case the abstract activity may be able to call other concrete activities, then all nodes and control flows of these concrete activities are translated as well. These concrete activities are then considered part of the plant specification for data-based synthesis.

Preconditions. This operation requires the UML model of the abstract input activity to be valid. Moreover, every occurrence constraint of $umlActivity_{abstr}$

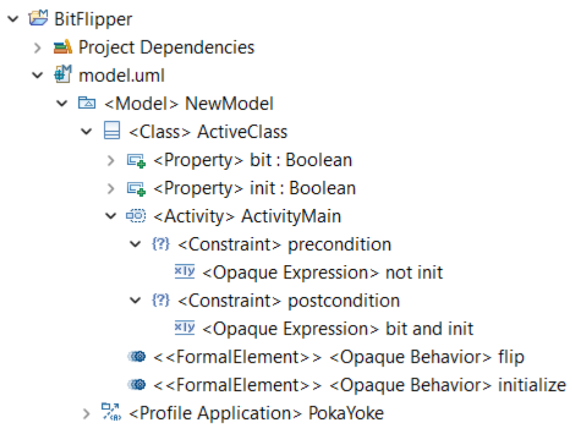
BitFlipper
Project Dependencies
model.uml
<Model> NewModel
<Class> ActiveClass
<Property> bit : Boolean
<Property> init : Boolean
<Activity> ActivityMain
<Constraint> precondition
<Opaque Expression> not init
<Constraint> postcondition
<Opaque Expression> bit and init
<<FormalElement>> <Opaque Behavior> flip
<<FormalElement>> <Opaque Behavior> initialize
<Profile Application> PokaYoke

Figure 1: The `flip` model visualized within the PokaYoke UML Designer environment.

must be expressed over actions and/or activities. If the to-be-synthesized activity is able to call other activities by its occurrence constraints, then all these activities must be concrete, and are not allowed to call other activities (i.e., they must be flattened).

Postconditions. This operation produces a CIF specification that:

- For every UML enum declaration in the UML model, contains a corresponding CIF enum declaration.
- Contains a CIF plant for the single UML class. This plant is a flower automaton, containing one location that is initial and marked, and only self-loops.
- Contains a discrete variable 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 actions that are both atomic and deterministic are translated as single controllable events. All other opaque behaviors, that model nonatomic and/or nondeterministic actions, are translated to multiple CIF events, namely a controllable one for starting the action, and uncontrollable ones for each of their effects to end the action. Such separate uncontrollable 'end' events must be defined since:
  - In case of a nonatomic action, we must allow other actions to be performed while executing the nonatomic action. Thus, we create

separate start and end events, to allow other CIF events to be performed in between. Moreover, the end events are defined to be uncontrollable so that data-based synthesis cannot influence when a nonatomic action ends. An example of a nonatomic action could be a robot movement. A synthesized controller cannot influence when such a robot movement finishes.

  - In case of a nondeterministic action, separate ‘end’ events are needed since data-based synthesis in CIF requires controllable events to be deterministic. 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 nonatomic and/or atomic nondeterministic actions, extra internal CIF variables are created to ensure that the actions are properly executed:

  - For every nonatomic action in the UML model, an *active variable* is declared and maintained in the CIF specification. The active variable of a nonatomic action is a Boolean that indicates whether the action is currently being executed (`true`) or not (`false`). The start event of a nonatomic action can only be performed when the active variable of that action is `false`, and performing it will set the active variable to `true`. Any end event of a nonatomic action can only be performed when the active variable of the action is `true`, and performing it will set the active variable to `false`.
  - In case the UML model contains atomic nondeterministic actions, a single 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 the atomic nondeterministic action. This atomicity variable is defined to be an integer between 0 and n, with n the total number of atomic nondeterministic actions. If the atomicity variable is 0, then no atomic nondeterministic action is being executed. The start event of any atomic nondeterministic action will set the atomicity variable to be the index of that action. If the atomicity variable is greater than 0, then only an end event of the corresponding (indexed) action can be performed, and performing such an end event will reset the active variable to 0. We add necessary guards to all created CIF edges to ensure that indeed no other CIF event can be performed when the active variable is not 0.

- For every concrete UML activity that can be called by the to-be-synthesized activity:

  - The translated CIF specification contains a Boolean-typed CIF discrete variable for every control flow of the concrete UML activity. This variable then indicates whether the corresponding control flow contains a token (`true`) or not (`false`).

  - The translated CIF specification contains CIF event declarations corresponding to all nodes of the concrete UML activity. We translate concrete activity nodes in the same way as we translate actions (i.e., opaque behaviors), as explained above. Control nodes (i.e., fork, join, decision, merge, initial, and final nodes) are always translated as atomic deterministic actions since their execution should be instant. Action nodes (i.e., opaque actions, call behavior nodes that call an opaque behavior, and shadowed call behavior nodes that call an activity) are translated likewise to opaque behaviors: as actions with user-specified guards and effects, that can be atomic or nonatomic, and deterministic or nondeterministic. During this translation, we define extra guards and effects to ensure proper token handling: an activity node can only be executed (as an action) if its incoming control flows have the required tokens for that, and after executing the node, the outgoing control flows receive the proper tokens. In case an outgoing control flow has a guard, then this guard is translated as an extra condition for putting a token on that control flow, i.e., for making the corresponding CIF variable `true`.

    Unlike the translation of opaque behaviors, every CIF event that is created for an activity node becomes uncontrollable, with the exception of initial nodes. This is because calling an activity intuitively amounts to 'executing its initial node', and this is a controllable event (that is, the synthesis algorithm has control over when some concrete activity is being called). However, once a concrete activity has been called, its execution proceeds uncontrollably, which is why all other CIF events are made uncontrollable.

    A concrete activity can only be called in a state where its precondition is satisfied. Hence, we translate their preconditions as extra guards for executing the initial nodes. Likewise, we translate their postconditions as extra guards for executing the final nodes.

- Contains an 'initial' predicate from the conjunction of all translated preconditions of the abstract UML activity. 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 abstract UML activity. There may be multiple user-defined postconditions defined for this activity. In case any internal active variables or atomicity variable were declared as described earlier, extra postconditions are generated which express that no nonatomic and/or nondeterministic actions can be active. Moreover, in case the abstract activity has occurrence constraints, extra postconditions are generated which express that that occurrence constraints must be satisfied, i.e., the actions/activities that are subject to occurrence constraints must have happened the specified number of times. Furthermore, in case concrete activities were translated, extra postconditions are generated expressing that none of their control flows must have a token. 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 postcondition, for which an algebraic variable is created as well. This algebraic variable is then used as the 'marked' predicate. Thus, a marked state is a state where the (combined) user-written activity postcondition is satisfied, where no nonatomic and nondeterministic actions are active, where every occurrence constraint is satisfied, and where no translated control flow holds a token.

- Contains requirement invariants stating that the postcondition disables any CIF event. In other words, if you would reach a system state where the activity postcondition holds, then no further actions would 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 unnecessary steps. And they ensure that the synthesized UML activity will have no further actions after having reached a final node.

- Contains an edge in the flower automaton plant for every defined CIF event declaration. An edge is defined for every declared CIF event, both controllable and uncontrollable ones. Every CIF event, and thereby also every CIF edge, corresponds to a translated action, i.e., a UML opaque behavior or a UML activity node. The action guards are then translated to guards that are put on the edges of the corresponding 'start' events of the action, and action effects are translated to edge updates for the 'end' events (thereby taking into account that the 'start' events of atomic deterministic actions also 'end' the action). Thus, we do not put user-written action guards on any edges corresponding to 'end' events, since we don't want conditions on any 'end' events, and by making them uncontrollable, any synthesized constraints are pushed back to the controllable events. However, we do put guards and effects on the (un)controllable events to correctly handle and maintain the atomicity variable and the active variables.

  Furthermore, edges are guaranteed not to contain conditional 'if'-updates, since these are not supported by data-based synthesis. Any conditional updates have been eliminated from the CIF specification. They are turned into conditional expressions, which are supported by data-based synthesis.

- Contains requirement automata for the occurrence constraints defined for $umlActivity_{abstr}$. Occurrence constraints are roughly of the form 'A must happen at least M times and at most N times' where A refers either to an action or an activity. Such constraints are translated as requirement automata, containing a discrete variable that maintains how often the action has already occurred, or how often the activity has already been called, respectively. In case A is an action, this variable is incremented every time the action occurs. In case A is an activity, this variable is incremented every time the initial node of the activity has been executed. The occurrence requirement can then be expressed over this requirement (using edge guards and marked predicates).

Example (Running Example)**.** The translation of the UML `BitFlipper` model to CIF is shown in Algorithm 2. In the CIF model, first the events are defined, either controllable (denoting the start of an action) or uncontrollable (usually describing a nondeterministic or external event). The `BitFlipper` CIF model provides two controllable events, `flip` and `initialize` that relate to starting the opaque behaviors defined in the UML model. Two uncontrollable events denote the two nondeterministic result of the `bit` initialization. The boolean property `bit` is transformed into a discrete boolean `disc bool` variable, which is not initialized explicitly, whereas `init` is initialized as false. The activity precondition and postcondition are translated into algebraic boolean variables. Then we define the initial state predicate `not init` to be the algebraic `precondition` variable, to indicate the initial states for performing synthesis. Moreover, we define the marked state predicate `marked` to be the algebraic `postcondition` variable, to indicate that a postcondition state must always be reachable in the controlled system. The auxiliary variable `__activeAction` is used to connect the start and end events of nonatomic and/or nondeterministic actions, in this case `initialize`. Finally, the CIF model denotes the `flip` event as an edge with guards `init` and `__activeAction = 0`, and effects composed of the single update `bit := not bit`.

3.2 Data-based synthesis with CIF

The operation `data-based-synthesis`(*cifSpec*) executes the data-based synthesis tool of CIF. Data-based synthesis is thereby configured to do forward reachability (`--forward-reach=true`), and to do no BDD predicate simplification (i.e., removing all simplifications from `--bdd-simplify`). Forward reachability is done for performance reasons, and because it may lead to more readable results (in particular UML edge guards). And by disabling all simplifications, we ensure that the synthesized conditions fully capture under which conditions the controllable events are enabled in the controlled system.

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 specified requirements. 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

Algorithm 2: CIF translation of the `BitFlipper` UML model.

```
controllable flip;
controllable initialize;
uncontrollable initialize__result_1;
uncontrollable initialize__result_2;
plant automaton ActiveClass:
  disc bool bit in any;
  disc bool not init;
  disc int[0..1] __activeAction;
  alg bool precondition = not init;
  alg bool __precondition = precondition;
  alg bool postcondition = bit and init;
  alg bool __postcondition__activeAction = __activeAction = 0;
  alg bool __postcondition = postcondition
    and __postcondition__activeAction;
  initial __precondition;
  marked __postcondition;
  location:
    initial;
    marked;
    edge flip when init, __activeAction = 0
      do bit := not bit;
    edge initialize when not init, __activeAction = 0
      do __activeAction := 1;
    edge initialize__result_1 when __activeAction = 1
      do init := true, bit := true, __activeAction := 0;
    edge initialize__result_2 when __activeAction = 1
      do init := true, bit := false, __activeAction := 0;
end
requirement invariant ActiveClass.__postcondition disables flip;
requirement invariant ActiveClass.__postcondition
  disables initialize;
requirement invariant ActiveClass.__postcondition
  disables initialize__result_1;
requirement invariant ActiveClass.__postcondition
  disables initialize__result_2;
```

requirements, running into blocking situations, etc. We will later make the behavior of this supervisor explicit (Section 3.3), to be able to synthesize a compact Petri Net for it (Section 3.7) that is then transformed to a concrete activity (Section 3.8).

After this transformation, we need to separately compute the guards of the control flows that go out of UML decision nodes (Section 3.10). This computation make use of the extra restrictions that are synthesized by data-based synthesis. Moreover, this computation requires that these extra restriction predicates are *not* simplified, which CIF does by default, hence the extra configuration.

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.

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. The guards of the supervisor automaton, for each controllable event, are the full/complete conditions under which these controllable events are enabled in the controlled system.

3.3 State space generation

The operation `generate-statespace`(*cifSupervisor*) executes the CIF explorer tool, which unfolds the state space of the given CIF specification, *cifSupervisor*.

Motivation. We need to explicitly unfold the (safe) state space of the synthesized supervisor *cifSupervisor* to be able to construct input for Petri Net synthesis, in order to later synthesize a concrete UML activity. With 'explicitly unfold' we mean that all data (e.g., discrete variables) is eliminated, leading to a state space that is a DFA/NFA. The reason is that Petrify does not have symbolic Petri Net synthesis algorithms, i.e., it cannot handle data. As input Petrify requires a DFA or NFA (i.e., a traditional supervisor in the form of a single DFA/NFA), rather than an EFA. Petrify does not support variables.

The state space that is generated by `generate-statespace` from the synthesized supervisor expresses all possible orderings of events, taking into account the extra synthesized guards and the original action guards as specified in the original input 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 be translated to a concrete 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, 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.

Preconditions. All preconditions of the CIF explorer tool apply.

Postconditions. All guarantees of the CIF explorer tool apply. The CIF explorer produces a CIF specification that contains exactly one automaton—the CIF state space. Noteworthy is that this state space will have state annotations `@state(...)` that indicate the values of every variable in every location, as well as the current location of each automaton that has at least two locations. This information will later be used for computing control flow guards (in Section 3.10).

Moreover, due to the way our UML/CIF input for synthesis is constructed (as result of Section 3.1), 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. This includes all implicit postconditions: no nonatomic and/or nondeterministic actions are active in those states, no translated UML control flows have a token, and every occurrence constraint is satisfied.
- 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 non-blocking. That is, any path from any location in the state space will either end up in a marked location (if it's not a marked location itself), 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 terminated already before having generated the state space), there is at least one initial location and at least one marked location.
- Any edge with an atomic nondeterministic event will end up in a location where only the uncontrollable end events of that nondeterministic action are possible. This property is a consequence of the execution semantics of atomic actions. If an atomic nondeterministic action is being executed in some location in the CIF state space, then by the atomicity constraint that we generated earlier, the only thing that could happen is an uncontrollable event to finish the atomic action.

3.4 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 event-based projection and DFA minimization, described in Sections 3.5 and 3.6, resp.

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 to-be-synthesized activity.

The situation is likewise for marked locations. The single CIF marked location that is guaranteed by `ensure-single-source-and-sink` will directly correspond to the single final node in the to-be-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 at which these different ways are merged. 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 (non-trivial) CIF initialization predicates nor any CIF marker predicates in components. (This is ensured if *cifStatespace* is generated by the CIF explorer.)
- Contain exactly one automaton with an explicit alphabet. (Which is already guaranteed by the CIF explorer.)
- 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 new 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 new single initial location has all state annotations of every original initial location. The original initial locations are now no longer initial.
- The resulting CIF specification has a single new 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 new single marked location has all state annotations of every original marked location. The original marked locations are no longer marked.
- Apart from initial/marked locations, the CIF specification is unchanged.

3.5 Event-based projection

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

Motivation. Recall that uncontrollable CIF events were created for atomic nondeterministic actions (in Section 3.1), to model the nondeterministic yet atomic 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 patterns in the state space to fork/join constructs in Petri Nets as much as possible. However, due to the atomicity constraint, the uncontrollable events of atomic nondeterministic actions do not give perfect diamond patterns, since whenever some atomic 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 performing Petri Net synthesis. We do not eliminate the uncontrollable end events of nonatomic actions since no atomicity constraints are imposed on such actions, i.e., their diamond patterns are still intact.

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. We used this option in earlier versions of our implementation 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 hiding/projection on the level of CIF instead.

Preconditions. Since `event-based-projection` uses the automaton projection tool that comes with CIF[3], 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. The input CIF specification must contain a single automaton, which in our case is the CIF state space, which in turn must have exactly one initial location.

[3]See `https://eclipse.dev/escet/cif/tools/eventbased/projection.html` (accessed 2024-08-01).

Postconditions. All guarantees of the automaton projection tool from CIF apply. Notably, event-based projection guarantees that the resulting automaton after projection is a DFA that contains only the projected events, and that is trace-equivalent to the input specification with respect to those events. In our case, this means that the resulting CIF specification no longer contains the uncontrollable events of atomic nondeterministic actions, and is trace-equivalent to the input modulo those events. The resulting DFA is not guaranteed to be minimal. We will minimize it in the next step.

3.6 DFA minimization

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

Motivation. We use DFA minimization to minimize the input for Petri Net synthesis, such that we get the smallest possible UML activity that is still optimal.

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 equivalence. From the documentation of Petrify[4] this option seems to have some interaction with the `--hide` option. However, as explained in Section 3.5, `--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 CIF[5], all preconditions from that tool apply.

Notably, the input should be a CIF specification containing exactly one deterministic automaton with exactly one initial location. In our case this is the projected CIF state space. Note that the previous event-based projection step ensures that the input we give to `dfa-minimization` is indeed deterministic and has exactly one initial location, thus allowing DFA minimization.

Postconditions. All guarantees of the DFA minimization tool from CIF apply. Notably, the result is a CIF specification containing a single minimal DFA that is trace-equivalent to the DFA in the input CIF specification.

The resulting automaton is a DFA with exactly one initial location. Moreover, this operation preserves the property from Section 3.4 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

[4] Which is included in the distribution that can be downloaded from the Petrify webpage.

[5] See https://eclipse.dev/escet/cif/tools/eventbased/dfa-minimize.html(accessed 2024-08-01).

Algorithm 3: Minimized CIF model.

```
controllable flip;
controllable initialize;
controllable __start;
controllable __end;
automaton minimal:
  alphabet flip, initialize, __start, __end;
  @state(projected = "s1")
  location s1:
    initial;
    edge __start goto s2;
  @state(projected = "s2")
  location s2:
    edge initialize goto s4;
  @state(projected = "s4")
  location s3:
    marked;
  @state(projected = "s3")
  location s4:
    edge __end goto s3;
    edge flip goto s5;
  @state(projected = "s5")
  location s5:
    edge __end goto s3;
end
```

can be no word in the automaton language that would distinguish m_1 and m_2. Hence they must be the same location.

Example (Running Example)**.** The minimized CIF model is shown in Algorithm 3. The CIF model has a `__start` and an `__end` event, denoting the start and end of the activity to be synthesized. The CIF automaton is composed of five locations:

- location $s1$ is the initial location, with one outgoing edge towards $s2$ after event `__start`;
- location $s2$ has one outgoing edge towards $s4$, after event `initialize`;
- location $s3$ is marked, namely is a final location, hence without outgoing edges;
- location $s4$ has two outgoing edges towards $s3$, after event `__end` (in case `bit` is initialized as true), and towards $s5$ after event `flip` (otherwise);
- location $s5$ has one outgoing edge towards $s3$.

This state space is the input to Petrify, that synthesizes a Petri Net from it.

3.7 Petri Net synthesis

The operation `petrinet-synthesis`($\mathit{cifStatespace}_{min}$) performs Petri Net synthesis to compute a minimal Petri Net whose behavior is trace-equivalent to the single automaton in the given CIF specification, $\mathit{cifStatespace}_{min}$. This operation attempts to synthesize a free-choice Petri Net, and if that fails, it synthesizes an ordinary Petri Net instead. Moreover, this operation produces a *region mapping*, which is a mapping that relates the input specification $\mathit{cifStatespace}_{min}$ to the synthesized Petri Net.

We use the Petrify tool for performing Petri Net synthesis. 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. If synthesis fails with the `-fc` option, Petrify will be invoked again without `-fc`.

In our activity synthesis algorithm, we use PNML as the intermediate format for representing Petri Nets. Therefore, part of the `petrinet-synthesis` operation is to translate $\mathit{cifStatespace}_{min}$ 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 (projected and minimal) CIF state space, $\mathit{cifStatespace}_{min}$, we are still quite distant from a concrete UML activity. The main reason is that concurrency can more concisely be represented in UML activities, via their fork and join nodes, compared to state machines and automata. In contrast, $\mathit{cifStatespace}_{min}$ has all concurrent interleaving explicitly unfolded as diamond patterns. We now somehow have to detect all diamond patterns of concurrent interleaving in $\mathit{cifStatespace}_{min}$ and translate those to fork/join patterns in the to-be-synthesized UML activity. This is done by means of Petri 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 token-based semantics. Our main strategy is therefore to synthesize a minimal Petri Net from $\mathit{cifStatespace}_{min}$ (in this section), and translate that to a UML activity (in Section 3.8). We thus use Petri Nets as an intermediate formalism in our activity synthesis algorithm.

Moreover, there are particular classes of Petri Nets, like *free-choice* Petri Nets, that are likely to lead to more intuitive UML activities. (This is based on our observations from earlier experiments. We haven't yet done an extensive study or comparison to see whether this is always the case.) Therefore we attempt to synthesize a free-choice Petri Net whenever possible.

However, translating a Petri Net to a UML activity structure is not yet sufficient: we then still need to compute the control flow guards for the decision nodes of the synthesized UML activity. These control flow guards are needed, since without them any UML decision node could freely follow any of its outgoing control flows, leading to executions that are not allowed by the synthesized CIF supervisor. This step must be done separately (in Section 3.10), 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, $\mathit{cifStatespace}_{min}$. Such a relation is (or should be) produced 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 $\mathit{cifStatespace}_{min}$ 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 3.10 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 `__loop` must not be in the automaton alphabet. This is because `petrinet-synthesis` will create an auxiliary self-loop edge named `__loop` before Petrify is invoked, which is removed later from the output of Petrify. This auxiliary self-loop is added to make Petrify work. This is because Petrify seems unable to do Petri Net synthesis in case the input automaton has deadlock locations, i.e., locations without outgoing edges. And recall that $\mathit{cifStatespace}_{min}$ has a single marked location which is a sink location, and thus when you get into this location during some execution, the execution deadlocks. Hence, to resolve this, we add an extra self-loop edge that connects the single marked location to itself, to still allow 'progress' from this sink location. After having invoked Petrify, the synthesized Petri Net will contain exactly one transition for `__loop` (since the input automaton has exactly one marked location and we added exactly one `__loop` edge), so it's easy to remove it again.

Finally, the string `__to__` must not occur in any event name 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. In particular, the output is an optimal (possibly free-choice) Petri Net that is trace-equivalent to the input automaton. The synthesized Petri Net is returned in PNML format.

As explained above, the `__loop` event has been removed from this Petri Net. The synthesized Petri Net contains exactly one place that initially holds a token. This place has no incoming arcs and exactly one outgoing arc to a transition named `__start`. The returned Petri Net also contains exactly one sink place with no outgoing arcs and at least one incoming arc from a transition named `__end` (there could be multiple such arcs, in contrast to the initial place). These two places will later become the initial and final node of the synthesized activity.

Petrify may have performed *label splitting*. This means that Petrify may have chosen to split up certain events e in $\mathit{cifStatespace}_{min}$ into events that are named $e/1$, $e/2$, etc. This is done by Petrify for technical reasons that are not explained further here, but which are needed to enable Petri Net synthesis. The consequence of label splitting is that the synthesized Petri Net may have multiple transitions for some CIF events e that are then labeled incrementally, as $e/1$, $e/2$, etc. We can later easily detect such duplication and remove these 'duplication markers'.

Moreover, as an internal step, the synthesized Petri Net has been normalized (i.e., all its places have been relabeled). This is needed since Petrify seems able to produce nondeterministic results in the sense that running Petrify multiple times on the same input model may give different results. But these different results are the same modulo names of places. We therefore normalize synthesized Petri Nets, for example to allow proper regression testing.

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

Example (Running example)**.** The Petrify output is depicted in Fig. 2. Notice that the end event `__end` has been split into two separate transitions that will be merged together (in fact, as a UML merge node) in a following step. Further, the final place `p5` is equipped with a self loop, which will also be removed in one of the post-processing steps.

3.8 Transforming the Petri Net to an activity

The operation `transform-to-activity`(*petriNet*) transforms the given (possibly free-choice) Petri Net *petriNet* in PNML format to a concrete UML activity.

Motivation. The overall aim of the activity synthesis chain is to synthesize concrete UML activities. Since activities are reasonably close to Petri Nets, we

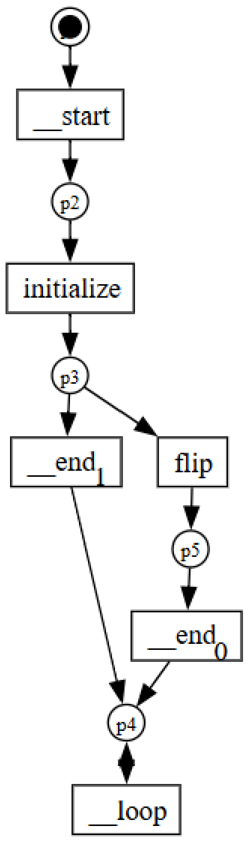

Figure 2: The Petrify output model.

do this by first synthesizing a Petri Net using known techniques, and transforming this Petri Net to a concrete UML activity using `transform-to-activity`.

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 does not have any outgoing arcs, which must additionally have at least one incoming arc from a transition named `__end`.

Postconditions. The output is a concrete UML activity which has been translated from *petriNet*. The returned concrete UML activity contains:

- Exactly one initial node and exactly one final node, which have been translated from the two places described above in the preconditions paragraph.
- One auxiliary opaque action named `__start`, which has been created for the start event introduced earlier, as explained in Section 3.4. Then the initial node has exactly one outgoing control flow to this `__start` action.
- An auxiliary opaque action named `__end` for every end event introduced earlier, as explained in Section 3.4.
- No other opaque action, apart from the `__start` and `__end` opaque actions. Then all these opaque `__end` actions have exactly one outgoing control flow that goes to the final node.
- Fork/join/decision/merge control 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 a UML merge node. The only control nodes that have multiple incoming control flows are join and merge nodes. The only control nodes that have multiple outgoing control flows are fork and decision nodes.
- Call behavior nodes for every transition in *petriNet* that performs an action. These nodes then call the appropriate UML opaque behavior of the action. Any duplication that was introduced by label splitting (recall from Section 3.7) has been removed during the process of finding the right UML opaque behavior corresponding to a Petri Net transition.
- No guards on any control flows. Control flow guards are computed later, in Section 3.10.

Furthermore, `transform-to-activity` ensures that every control flow that goes out of a decision node will end in either some (call behavior) action node, or in the final node (in which case there was also an action, but it was the auxiliary internal `__end` action). This is a consequence of the structure of the Petri Nets that are synthesized by Petrify, and the patterns that they could have. To give another example of a pattern that cannot occur: it is not possible for the concrete UML activity to have a control flow that goes from some decision node to some other decision node, because in the Petri Net there cannot be an arc from a place to another place.

3.9 Reducing nonatomic patterns

The operation `reduce-non-atomic-patterns` merges any nonatomic patterns in the input concrete UML activity that can be merged.

Note that, in the code implementation, this reduction step is actually done on the Petri Net level rather than the UML level, as well as the next step of computing control flow guards, described in Section 3.10. This is done for technical reasons that for now are too detailed for the purpose of this document, which is why we chose to explain this (and the next) step on the UML level instead. The underlying reason is that the intermediate Petri Net representations have a nicer structure than the translated UML activities when it comes to computing control flow guards. Moreover, we have to reduce nonatomic patterns before computing control flow guards, which is why the code implementation performs also this reduction step on Petri Net level.

Motivation. Recall that in the UML-to-CIF transformation (Section 3.1) we created separate controllable start events and uncontrollable end events for nonatomic and atomic nondeterministic actions. Then, in the earlier event-based projection step (Section 3.5) we got rid of the end events of the atomic nondeterministic actions, by projecting those out. However, the end events of nonatomic actions are still in the CIF specifications, and thus also in the Petri Net specifications, and thus also in the UML activity specifications. The `reduce-non-atomic-patterns` operation will attempt to merge the starts and ends of nonatomic actions back to single call behavior actions wherever possible.

However, merging these starts and ends is rather delicate, and not always possible. To see why, let us have a look at nonatomic action patterns. Consider a nonatomic nondeterministic action A that has two nondeterministic effects. Then one (controllable) start event and two (uncontrollable) end events were defined for A during the UML-to-CIF translation. Let us call these $\mathsf{A}_{\mathsf{start}}$, $\mathsf{A}_{\mathsf{end1}}$, and $\mathsf{A}_{\mathsf{end2}}$, respectively. Figure 3 (left) then shows an example of a nonatomic pattern for A as it may currently look on the UML activity level. The action $\mathsf{A}_{\mathsf{start}}$ has the original guard of A but no effects. The actions $\mathsf{A}_{\mathsf{end1}}$ and $\mathsf{A}_{\mathsf{end2}}$ both have guard `true`, and have the first and the second nondeterministic effect of A, respectively. The intermediate decision node is not allowed to be connected to any other nodes than the starts and ends of A, otherwise the pattern doesn't apply. We aim to (eventually) merge this pattern back into a single call behavior node that simply calls A.

However, we cannot directly merge this pattern here since that would complicate choice guard computation later on. If we would merge the pattern here, then we would have to push down the decision node to below the merged 'call A' action. By doing so, it would become (more) difficult to compute the control flow guards, as there may be all kinds of forking/joining/branching/merging

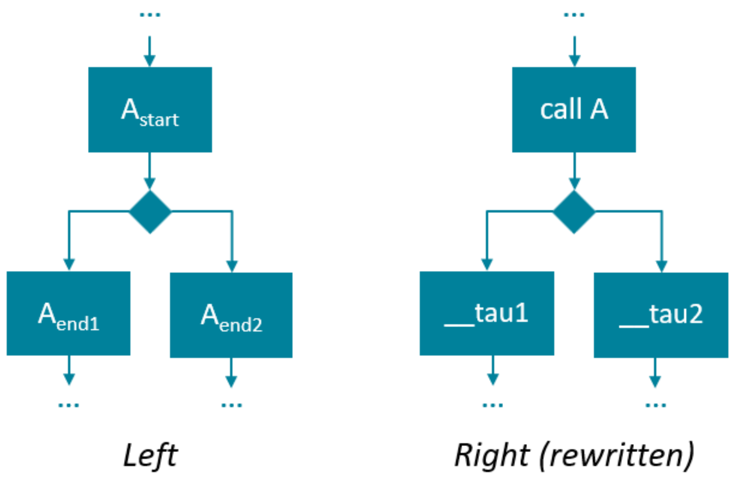

Figure 3: Left: a nonatomic pattern for a nonatomic nondeterministic action A with two nondeterministic effects, presented on UML activity level. Right: the same pattern, but rewritten by `reduce-non-atomic-patterns`.

constructions in the '...' parts below the decision node that we then would have pushed down[6]. These constructions make it more difficult (though not impossible) to gather all necessary state information for the guard computation.

Therefore, instead of directly merging the nonatomic pattern, we replace the $\mathsf{A}_{\mathsf{start}}$ node by a 'call A' node, and replace the end nodes by internal 'tau' nodes, which are later eliminated (in Section 3.11). The result is shown on the right in Figure 3. With this rewritten pattern, it becomes easier to later compute the control flow guards for the decision node, since the decision node structure is now fixed by means of the internal 'tau' nodes. This allows computing the guards for this decision node, while any (potentially complicated) structure that's below the 'tau' nodes can be handled separately. However, to compute the choice guards for this decision node, we do have to consider that, when comparing the left pattern with the right pattern in Figure 3, the nondeterministic effects are evaluated at different points. In the left pattern, the nondeterministic effect applies after the decision node, whereas in the rewritten pattern on the right, the effect applies before the decision node, as part of the 'call A' action. Section 3.10 explains how to deal with this when computing control flow guards.

[6]This is also the main reason why we actually do the guard computation on the Petri Net level rather than on UML activity level. On Petri Net level, there are only places and transitions, in contrast to the various different types of nodes that UML has for forking, joining, branching, merging, etc., which might also be composed in arbitrary ways. Nevertheless, the high-level synthesis algorithm is easier to explain on UML activity level, since then the explanation doesn't have to rely on details such as the connection/relation between the Petri Net and the translated UML activity.

Finally, merging nonatomic starts and ends is not always possible, since the nonatomic pattern may not always apply. This may, e.g., happen in case of imperfect concurrent interleaving. For example, consider some atomic action B such that, after data-based synthesis, the synthesized supervisor allows B to happen before A, during A (i.e., in between its start and end events), but not after A. Then there is no full diamond pattern with respect to the interleavings of A and B that can be reduced to a forking construct by Petri Net synthesis. As a result, the intermediate decision node in Figure 3 (left) will have an extra outgoing control flow to B, which breaks the nonatomic pattern. In such cases, we do not merge the starts and ends of nonatomic actions, but keep them separated. These cases are further handled during activity post-processing (Section 3.11).

Preconditions. This operation requires a concrete UML activity resulting from the previous step (the code implementation actually requires its earlier Petri Net representation). This model should not use the string `__tau` in any of its identifiers, as `reduce-non-atomic-patterns` will use it to represent rewritten end points of nonatomic patterns.

Postconditions. This operation returns the input activity, where all nonatomic patterns have been rewritten as shown on the right in Figure 3. Any starts and ends of nonatomic actions whose occurrences do not fit the nonatomic pattern will not have been rewritten, and are handled during activity post-processing.

3.10 Computing control flow guards in the activity

The operation `compute-edge-guards`($umlActivity_{red}$, ...) computes guards for every control flow in the single concretized UML activity $umlActivity_{red}$ that goes out of a decision node. These guards are computed by using intermediate results from earlier steps in the activity synthesis algorithm, notably the extra conditions that have been synthesized earlier by data-based synthesis, the CIF state annotations in the generated/projected/minimized CIF state space, the region mapping, the tracing information from the earlier steps in the synthesis chain, etc. Rather than listing all items explicitly, we abbreviate them as '...'.

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 could not have yet been computed since Petrify (or any other known algorithm for Petri Net synthesis) cannot handle data. So we need to compute them separately, and add them to the concrete UML activity.

The three essential ingredients for computing control flow guards are:

- The *uncontrolled system guards*, as a mapping from actions to their corresponding guards as defined in the original input UML model. These guards can thus be obtained by querying the original UML input model.

- The *controlled system guards*, as a mapping from actions that were controllable on the CIF level, to their corresponding synthesized guards. The synthesized guards thus are the full conditions computed by data-based synthesis, under which the action is enabled in the controlled system.

- The *state information*, as a mapping from control flows (or, equivalently, Petri Net places), to the predicate that describes all states the system can be in whenever the control flow is enabled (i.e., whenever their is a token on the corresponding Petri Net place).

These three ingredients can be computed from the intermediate results of earlier steps in the synthesis chain, as indicated in the preamble of this section.

Constructing the state information mapping requires some backtracking. For every execution point (roughly corresponding to control flows) 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 3.3), through the various CIF event-based toolkit steps. So for every control flow we must: (1) determine the Petri Net place corresponding to it, (2) use the region mapping produced in Section 3.7 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 atomic nondeterministic actions, as we should not consider the state annotations of any 'intermediate' CIF locations in which an atomic nondeterministic action is being executed (e.g., since the atomicity variable is true in any such intermediate state, and choice guards should not depend on internal details). Therefore, any such state annotations must be filtered out first. The set of all system states can then be represented as a single predicate.

Note that all the predicates are internally represented in the code implementation as Binary Decision Diagrams (BDD). This is because BDDs allow easy manipulation of Boolean functions, like BDD simplification, i.e., `simplify`. CIF contains functionality for translating BDD representations of predicates, to CIF expressions, which can in turn be translated to strings.

Now let us explain how control flow guards are computed. To start, let us first determine an *uncontrolled guard* and a *controlled guard* for every UML action node, which form the basis for computing control flow guards. Let N be any action node in the UML activity for which we want to compute control flow guards. We define *uncontrolled*(N) to be the uncontrolled guard predicate for N such that:

- In case N is a call behavior action node, *uncontrolled*(N) is the uncontrolled system guard for that action.

- In case N is an internal `__start` action node, *uncontrolled*(N) is the activity precondition. That is, N is only executed in states where the activity precondition holds.

- In case N is an internal `__end` action node, *uncontrolled*(N) is the activity postcondition. That is, N is only executed in states where the activity postcondition holds.
- In case N is an internal `__tau` action node, *uncontrolled*(N) is the conjunction of the state info predicates of all outgoing control flows of the `__tau` node. That is, N is only executed from system states where the corresponding effect of the nonatomic action has applied. With *state info predicates* we mean the predicates from the state information mapping, in this case for all outgoing control flows of the `__tau` node. We compute the *conjunction* of all these state info predicates, since that predicate describes all states the system can be in right after the corresponding effect has been executed. Especially in case the activity forks after the `__tau` action node, computing the conjunction ensures that we get the state info predicate that applies right after the `__tau` node and right before the fork node. This computation of *uncontrolled*(N) is needed since we've rewritten the nonatomic pattern in the previous step, meaning that we have to calculate the new uncontrolled guards of the `__tau`-s for the new situation.
- In all other cases, *uncontrolled*(N) is the predicate `true`. That is, the execution of N is not restricted by any guard predicate, which may happen, e.g., if N is an nonatomic end event that's not reduced earlier.

Let us define *controlled*(N) to be the controlled guard predicate for N such that:

- In case N is a call behavior action node, *controlled*(N) is the controlled system guard for that action. That is, *controlled*(N) is the controlled-system condition resulting from synthesis, for the action called by N.
- In all other cases, *controlled*(N) is undefined. That is, no conditions were synthesized that further restrict N in the controlled system.

Next we define *extra*(N) as a predicate describing the extra synthesized condition for executing N, such that:

- If *controlled*(N) is not defined, then *extra*(N) = *uncontrolled*(N).
- If *controlled*(N) is defined, then

  $$\mathit{extra}(N) = \texttt{simplify}(\mathit{controlled}(N), \mathit{uncontrolled}(N)),$$

  where `simplify`(p, q) is an operation that simplifies some given predicate p with respect to some predicate q, and returns the simplified predicate. This simplification is based on heuristics. Various algorithms exist for simplifying (symbolic representations of) predicates. We can simplify here since any restrictions covered by the controlled guard for N are already considered by the execution semantics of activities.

Now, given any control flow cf in the UML activity that goes from some decision node to some action node N, we may use $\mathit{extra}(N)$ as the control flow guard for cf. This would be a valid guard, but it can also be quite lengthy since we configured data-based synthesis to not perform any simplifications. It is known that data-based synthesis may synthesize long control conditions that are difficult to read and understand by engineers. To make the guard (hopefully) human readable, we can further simplify it by state information. Another reason for further simplification is that $\mathit{extra}(N)$ may be expressed over internal state, like internal CIF variables for encoding occurrence 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 $\mathit{extra}(N)$ by using any contextual information we have at hand, if $\mathit{extra}(N)$ happens to be non-trivial.

Also note that, in UML activities, the control flows that go out of decision nodes do not have to always target an action node. The target node could in principle also, e.g., be a fork or join node introduced by `transform-to-activity`. This is why the code implementation actually computes choice guards on the Petri Net level, and then adds them to the UML activity. Any UML decision node corresponds to a Petri Net place, and by the structure of Petri Nets, all outgoing arcs of that place will have a Petri Net transition as target, corresponding to some action. This makes guard computation easier to do on the Petri Net representations as then you don't have to deal with any additional structure introduced by `transform-to-activity`. Yet the high-level ideas can well be explained on the level of UML activities.

To allow further simplification, we first define the state information predicate $\mathit{stateinfo}(D)$ for any decision node N, so that:

- If D is an intermediate decision node of a rewritten nonatomic action pattern, then $\mathit{stateinfo}(D)$ is defined to be disjunction of the predicates $\mathit{uncontrolled}(\texttt{\_\_tau}_i)$ of all the internal $\texttt{\_\_tau}_0, \texttt{\_\_tau}_1, \ldots$ nodes connected to D by an outgoing control flow. In other words, the system states you can be in while executing D, are all the system states you can be in directly after the nonatomic action has taken effect. Note that this computation of $\mathit{stateinfo}(D)$ is needed since we've rewritten the nonatomic pattern, meaning that we have to recompute the relevant state information for D.

- Otherwise, if D is not part of any nonatomic pattern, then $\mathit{stateinfo}(D)$ is defined to be the state information predicate of its incoming control flow. (Recall that `transform-to-activity` guarantees that D has a single incoming control flow.)

Now we can further simplify the extra guard predicates defined earlier, which finishes guard computation. Given any control flow cf in the UML activity that goes from some decision node D to some node N, we define the guard for cf to be $\texttt{simplify}(\mathit{extra}(N), \mathit{stateinfo}(D))$. 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 always be eliminated.

Preconditions. The input UML activity $\mathit{umlActivity}_{red}$ should be a synthesized concrete UML activity, with its nonatomic patterns reduced wherever possible (as result of Section 3.9). Moreover, all intermediate information from earlier steps of the activity synthesis algorithm must be available, like the original UML input model, the synthesized Petri Net, the region mapping, the state annotations of the generated/projected/minimized CIF state space, the CIF supervisor, etc.

As indicated earlier in Section 3.2, the CIF data-based synthesis tool should have been configured to disable all BDD simplification. This is needed for computing choice guards, since if synthesized conditions were simplified, they may no longer capture restrictions that are imposed by CIF requirements as those would have been simplified away. Instead, we want the synthesized conditions to completely capture all imposed restrictions, including requirements.

Moreover, all nonatomic patterns must have been reduced, since their reduction has an impact on the computation of choice guards, as motivated earlier.

Postconditions. The output is a UML activity that is equal to $\mathit{umlActivity}_{red}$, but with proper choice guards added to all control flows that go out of decision nodes. These added choice guards are Boolean predicates that express extra guard conditions for taking the control flow, that are not yet captured by the original action guards nor other known system state information.

Moreover, the added choice guards do (hopefully) not use internal variables, like the atomicity variable. We are not sure that the computed choice guards will indeed never use internal variables. On the other hand, we have not found a counterexample yet where the computed guards use internally generated variables. To be sure, we have added an assertion in the code to check for use of internal variables, and if such use is detected, the `compute-edge-guards` operation will terminate exceptionally. So if the operation terminates, then the choice guards do not use internal variables.

3.11 Post-processing of the activity

The operation `postprocess-activity`($\mathit{umlActivity}_{guards}$) post-processes the synthesized activity $\mathit{umlActivity}_{guards}$, to:

- Rewrite any leftover nonatomic actions that were not merged earlier by `reduce-non-atomic-patterns`.
- Remove any internal actions have been introduced in earlier steps, like the opaque `__start`, `__end`, and `__tau` actions.
- Further simplify the synthesized activity.
- Remove some names from UML control nodes and control flows, for better readability in UML Designer.

Motivation. The `postprocess-activity` operation first rewrites any leftover nonatomic actions that were not merged earlier. Recall from Section 3.9 that not all occurrences of nonatomic actions can be reduced as part of a nonatomic pattern. This means that the activity might still contain separate actions for the starts and ends of nonreduced nonatomic actions. We rewrite these to ensure that the synthesized activity is valid. We do this by turning all such starts to opaque actions that have the original action guard and no effects, and turning all ends to opaque actions that have guard `true` and the corresponding effect of the nonatomic action as their (single) effect. We also rename all these opaque actions, to ensure that they are no longer considered to be internal actions.

Next, we remove any internal actions. Any auxiliary actions that were not part of the original input UML model should be removed from the synthesized result, as these should not be visible to the user.

After that, we further simplify the synthesized activity. This is done for two reasons: (1) we know that translating a minimal Petri Net to a UML activity does not imply that the translated activity is minimal, and (2) the removal of internal actions may lead to patterns in the UML activity that may further be reduced. This simplification step searches for several predefined patterns in the UML activity, and rewrites any found instances of these patterns.

Finally, activity post-processing gives the opportunity to do some other minor improvements, e.g., to improve readability a bit by removing some names.

Preconditions. This operation requires a concrete synthesized UML activity $umlActivity_{guards}$ resulting from the earlier steps.

Postconditions. This operation ensures that the synthesized UML activity no longer contains internal actions, is simplified, and has any leftover nonatomic starts and ends rewritten in order to get a valid activity. Moreover, all names of control nodes (initial/final/fork/join/decision/merge nodes) and control flow have been removed for better readability.

Example (Running Example)**.** Finally, the synthesized activity is shown in Fig. 4. The initial node is marked in blue, the place `p3` has been turned into a decision node, and the two outgoing control flows represent the nondeterministic initialization of `bit`. If `bit` is true, the flow can go directly to the final node (shown with a blue border); otherwise, the flow goes through the `flip` action. The two `__end` transitions have been translated into a merge node, and finally place `p5` is transformed into the final node.

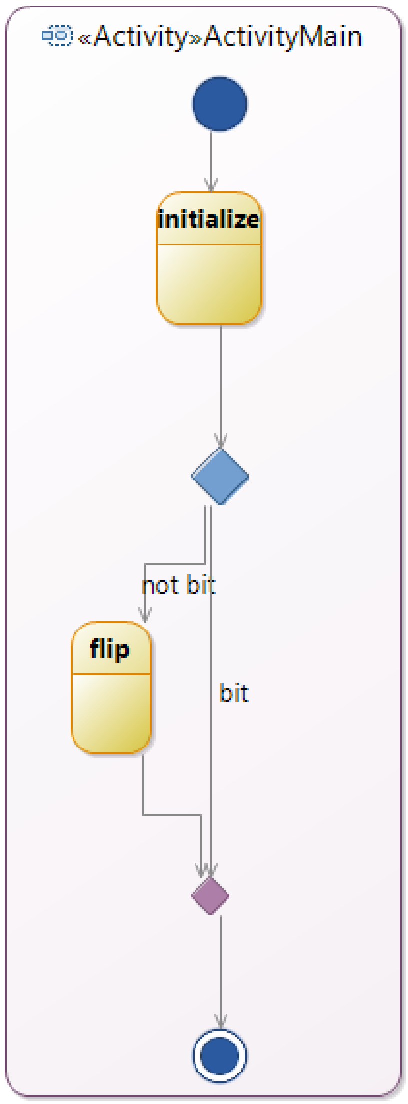

Figure 4: The synthesized activity.