Synthesis of Poka Yoke Activity Diagrams

October 30, 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, 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 at 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 nonatomic 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 2025-08-21) for the full UML2 specification.

also be an action exclusion invariant, which indicates that starting the execution of some action requires the system to be in a certain state as expressed by the 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 times some action can be started, or the number of times some activity can be called, in a to-be-synthesized activity. Note that occurrence constraints are temporary constructs and are intended to be removed later, which is future work.
  - A concrete UML activity is a finite directed graph consisting of *nodes* and *control flows*[2]. A node can be a:
    - *Control node*, i.e., an initial node, final node, fork node, join node, decision node, or merge node.
    - *Call behavior action node*, which either calls an activity, or executes an action by calling an opaque behavior.
    - *Opaque action node*, which is essentially an 'inlined' action consisting of a guard and zero or more effects. So likewise to opaque behaviors, also opaque action nodes are either deterministic or nondeterministic, and either atomic or nonatomic.

    Likewise to abstract activities, also concrete activities may contain preconditions, postconditions, and occurrence constraints. 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.)

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

The activity synthesis algorithm takes UML models as input, containing any number of (concrete and 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 one place and one 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 standard literature.

The activity synthesis algorithm uses a particular category of Petri Nets where possible, 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, among other systems, 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 2025-08-21) for further details. We use the CIF data-based synthesis tool 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 2025-08-21).

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 2025-08-21).

UML. We use the UML2 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 2025-08-21).

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 (free-choice, if possible) 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 than 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., elementary 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.

We will not explain supervisory controller 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 atomic 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 sets `bit` either to `true` or `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 CIF 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 := petrinet-synthesis(cifStatespace_min);
15 |   petriNet_red := rewrite-nonatomic-patterns(petriNet);
   |   // Transform the Petri Net to an activity without control flow guards.
16 |   umlActivity_concr := transform-to-activity(petriNet_red);
   |   // Compute the control flow guards for the synthesized activity.
17 |   umlActivity_guards := compute-guards(umlActivity_concr);
18 |   return umlActivity_guards;
```

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, concrete UML activities, 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 the CIF data-based synthesis tool. In case the to-be-synthesized 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 $umlActivity_{abstr}$ must be expressed over actions and/or activities in that UML model. If the to-be-synthesized activity is able to call other activities by its occurrence con-

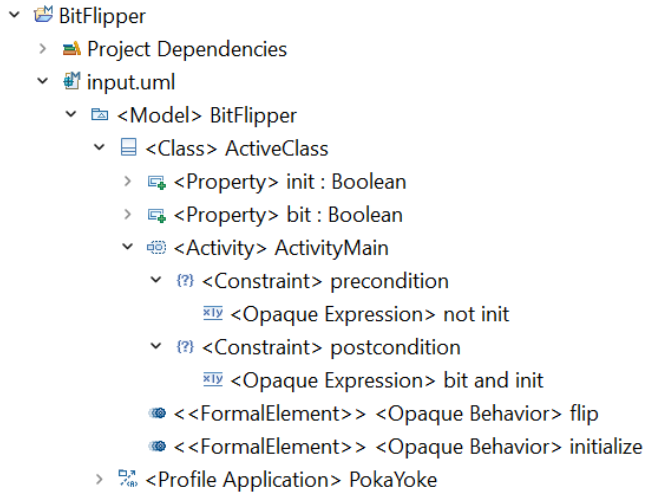


Figure 1: The `BitFlipper` model visualized within the PokaYoke UML Designer environment.

straints, then all these activities must be concrete and must not 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. In that case, at least one uncontrollable end event will be created, even if the opaque behavior has no user-defined effects. 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 *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 atomicity 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 atomicity 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 is 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 Boolean-typed algebraic variables for every translated precondition of the abstract activity. Activities may have multiple preconditions. Each of these preconditions are translated as algebraic variables in CIF, for better traceability. The conjunction of all these algebraic variables then forms the overall activity precondition, for which an algebraic variable is created as well, named `__precondition`.

- Contains Boolean-typed algebraic variables for every translated postcondition of the abstract activity. Activities may have multiple postconditions. Each of these postconditions are translated as algebraic variables in CIF, for better traceability. Apart from these, extra postconditions are generated as Boolean-typed algebraic variables in the following situations:

  - 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.
  - In case the abstract activity has occurrence constraints, extra postconditions are generated which express that occurrence constraints must be satisfied, i.e., the actions and activities that are subject to occurrence constraints must have happened the specified number of times.
  - In case concrete activities were translated, extra postconditions are generated expressing that none of their control flows must have a token.

  The conjunction of all these algebraic variables then forms the overall activity postcondition, for which an algebraic variable is created as well, named `__postcondition`.

- Contains an 'initial' predicate which is defined as `__precondition`, i.e., the conjunction of all translated preconditions of the abstract activity. This predicate then limits the number of initial states to only the ones satisfying the overall activity precondition.

- Contains a 'marked' predicate which is defined as `__postcondition`, i.e., the conjunction of all translated (user-defined and extra) postconditions of the abstract activity. This predicate then limits the number of marked states to only the ones satisfying the overall activity postcondition.

- Contains a requirement invariant for every created CIF event, which disables the CIF event in every state in which `__postcondition` is satisfied. In other words, if you would reach a system state where the overall 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 extra guards and effects on the 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 the abstract activity. 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 opaque behavior 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 opaque behavior, this variable is incremented every time that behavior is called. In case A is an activity, this variable is incremented every time the initial node of the activity has been executed. The occurrence constraint can then be expressed in this requirement automaton (using edge guards and marked predicates).

Example (Running Example)**.** The translation of the UML `BitFlipper` model to CIF is shown in Listing 1. 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 results of the `bit` initialization. The boolean property `bit` is transformed into a discrete boolean (`disc bool`) variable, which is nondeterministically initialized (denoted `in any`), whereas `init` is initialized as `false`. The activity precondition and postcondition are translated into algebraic boolean variables. Then we define the initial state predicate to be the algebraic `__precondition` variable, to indicate the initial states for performing synthesis. Moreover, we define the marked state predicate 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 atomic nondeterministic actions, in this case `initialize`. The atomic deterministic opaque behavior `flip` is translated into an edge with guards `init` and `__activeAction = 0`, and a single update `bit := not bit`. The atomic nondeterministic opaque behavior `initialize` is translated into three edges. First, an edge `initialize` is defined, with guard `not init and __activeAction = 0`, whose effect is `__activeAction := 1` (representing the atomic nondeterministic action starting). Next, two edges denote the two possible results of the nondeterministic action: `initialize__result_1` and `initialize__result_2` assign `true` and `false` to `bit`, respectively. At the same time, both edges assign `true`

to `init` and `0` to `__activeAction`, representing respectively the initialization of the system, and the end of the atomic nondeterministic action. Finally, four requirement invariants state that the overall postcondition disables all CIF events. In practice, these four requirements ensure that once the system reaches a state where the postcondition holds, no additional actions can be taken.

```
1  controllable flip;
2  controllable initialize;
3  uncontrollable initialize__result_1;
4  uncontrollable initialize__result_2;
5  plant automaton ActiveClass:
6    disc bool init = false;
7    disc bool bit in any;
8    disc int[0..1] __activeAction;
9    alg bool precondition = not init;
10   alg bool __precondition = precondition;
11   alg bool postcondition = bit and init;
12   alg bool __postcondition__activeAction = __activeAction =
     0;
13   alg bool __postcondition = postcondition and
     __postcondition__activeAction;
14   initial __precondition;
15   marked __postcondition;
16   location:
17     initial;
18     marked;
19     edge flip when init, __activeAction = 0 do bit := not
       bit;
20     edge initialize when not init, __activeAction = 0 do
       __activeAction := 1;
21     edge initialize__result_1 when __activeAction = 1 do
       init := true, bit := true, __activeAction := 0;
22     edge initialize__result_2 when __activeAction = 1 do
       init := true, bit := false, __activeAction := 0;
23 end
24 requirement invariant ActiveClass.__postcondition disables
   flip;
25 requirement invariant ActiveClass.__postcondition disables
   initialize;
26 requirement invariant ActiveClass.__postcondition disables
   initialize__result_1;
27 requirement invariant ActiveClass.__postcondition disables
   initialize__result_2;
```

Listing 1: CIF translation of the `BitFlipper` UML model.

3.2 Data-based synthesis with CIF

The operation `data-based-synthesis`(*cifSpec*) executes the CIF data-based synthesis tool. Data-based synthesis is thereby configured to do forward reachability (`--forward-reach=true`), and to start with forward reachability in the fixed-point order (`--fixed-point-order=reach-nonblock-ctrl`). Enabling forward reachability may lead to more readable results, in particular UML control flow guards. We start the fixed-point order with a forward reachability search, since in earlier activity synthesis examples, we experienced that this is computationally cheaper than starting with a backward reachability search. To see why, consider a to-be-synthesized abstract activity that is able to call various concrete activities. In that case, a backward reachability search may consider many behaviors that can never actually occur, like (combinations of) intermediate execution states of concrete activities that might hypothetically be executing. A forward reachability search may find out early that many of these behaviors can never occur, for example since concrete activities can only be called in certain situations—when their preconditions hold. We have seen concrete synthesis examples where this helps significantly. However, by now we have also seen cases where activity synthesis is significantly faster when starting with a backward search. We still need to properly investigate the best reachability strategy, which is probably dependent on the input specification.

Motivation. Our goal of performing data-based synthesis is to compute all extra restrictions on the execution of actions and the calls to activities 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 preconditions, to a state that satisfies the postconditions, without violating 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 3.9). After this transformation, we need to separately compute the incoming and outgoing guards of the control flows (Section 3.10).

Preconditions. All preconditions of the CIF data-based synthesis tool apply.

Postconditions. All guarantees of the CIF data-based synthesis tool apply.

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. Therefore, 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. 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 start 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 SynthML 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.

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 these events. In our case, this means that the resulting CIF specification no longer contains the uncontrollable end events of atomic nondeterministic actions, and is trace-equivalent to the input modulo these 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`($cifStatespace_{proj}$) 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. Minimizing the DFA is not strictly required to be able to apply Petrify, but might help the Petri Net synthesis algorithm so that it does not have to reason about nondistinguishable DFA states. Note that Petrify also has a built-in option `--mints` to minimize the input graph modulo trace equivalence, as a pre-processing step. From the documentation of Petrify[4] this option seems to have some interaction with the `--hide` option. However, as explained in

[3] See https://eclipse.dev/escet/cif/tools/eventbased/projection.html (accessed 2025-09-24).

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

Section 3.5, `--hide` does not seem to always work. Therefore, in addition to event-based projection, we also do the DFA 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 can be no word in the automaton language that would distinguish m_1 and m_2. Hence they must be the same location since the automaton is minimal.

Example (Running Example)**.** The minimized CIF model is shown in Listing 2. 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 $s3$, after event `__end`.

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

```
1 controllable flip;
2 controllable initialize;
3 controllable __start;
4 controllable __end;
```

[5] See https://eclipse.dev/escet/cif/tools/eventbased/dfa-minimize.html (accessed 2025-09-24).

```
5  automaton minimal:
6    alphabet flip, initialize, __start, __end;
7    @state(projected = "s1")
8    location s1:
9      initial;
10     edge __start goto s2;
11   @state(projected = "s2")
12   location s2:
13     edge initialize goto s4;
14   @state(projected = "s4")
15   location s3:
16     marked;
17   @state(projected = "s3")
18   location s4:
19     edge __end goto s3;
20     edge flip goto s5;
21   @state(projected = "s5")
22   location s5:
23     edge __end goto s3;
24 end
```

Listing 2: Minimized CIF model.

3.7 Petri Net synthesis

The operation `petrinet-synthesis`($cifStatespace_{min}$) performs Petri Net synthesis to compute a minimal Petri Net whose behavior is trace-equivalent to the single automaton in the given CIF specification, $cifStatespace_{min}$. This operation attempts to synthesize a free-choice Petri Net, and if that fails, it synthesizes an ordinary Petri Net instead.

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 $cifStatespace_{min}$ to the input language of Petrify, and transforming the output of Petrify to PNML.

Motivation. By having computed a (projected and minimal) CIF state space, $cifStatespace_{min}$, 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, $cifStatespace_{min}$ 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 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 $cifStatespace_{min}$ (in this section), and translate that to a UML activity (in Section 3.9). 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, just translating a Petri Net to a UML activity structure is not yet sufficient: we then still need to account for the data which had been eliminated in the CIF state space generation step (Section 3.3). In particular, (incoming and outgoing) control flow guards are needed, for example for the decision nodes of the synthesized UML activity, since without such guards, any control flow could freely be taken, leading to executions that would not be allowed by the synthesized CIF supervisor. This step is 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.

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 $cifStatespace_{min}$ 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 tran-

sition 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 $cifStatespace_{min}$ 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.

Example (Running example)**.** The Petrify output is depicted in Figure 2. Notice that the end event `__end` has been split into two separate transitions due to label splitting. These two transitions will be merged together in the conversion step to a UML activity, and be turned into a UML merge node. Further, the 'final' place `p4` is equipped with a self-loop, which will be removed when converting this Petri Net to PNML format.

3.8 Reducing nonatomic patterns

The `rewrite-nonatomic-patterns` operation rewrites any nonatomic patterns in the Petri Net that can be merged. A *nonatomic pattern* in a Petri Net is defined as shown in Figure 3a, consisting of:

- A 'start' transition, corresponding to a start event of an nonatomic (possibly nondeterministic) action. In the figure, the A transition is the start transition.

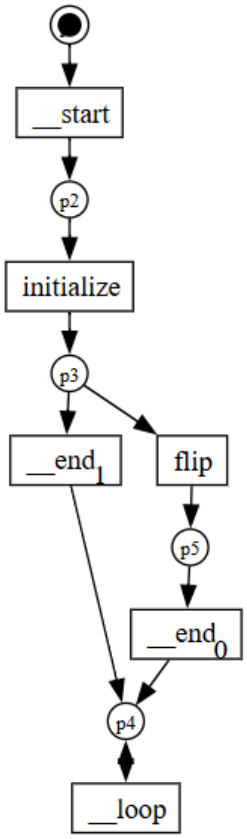
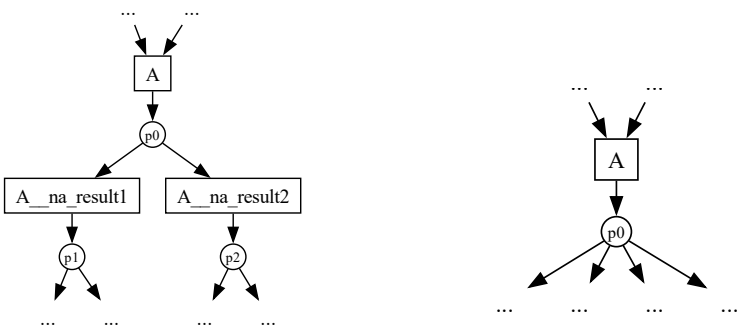


Figure 2: The Petrify output model.

- One or more 'end' transitions, corresponding to the end events of the nonatomic action that got started by the start transition. There are multiple such end transitions if and only if the action is nondeterministic. The figure shows two end transitions: A__na_result1 and A__na_result2.
- An 'intermediate' place that connects the start and end transitions. In the figure, p0 is the intermediate place.
- An 'end' place for every end transition. In the figure, p1 and p2 are the end places.

This pattern definition is strict, in the sense that no other places, transitions, and arcs can be part of the pattern. That is, the start transition cannot have other outgoing arcs, the intermediate place cannot have other incoming/outgoing arcs other than the ones from/to the start and end transitions as shown in Figure 3a, the end transitions cannot have other incoming and outgoing arcs, and the end places cannot have other incoming arcs.



(a) A nonatomic pattern for a nonatomic nondeterministic action A with two nondeterministic effects.

(b) The rewritten nonatomic pattern shown on the left. The 'end' places and transitions have been merged into the 'intermediate' place.

Figure 3: A nonatomic pattern (left) and a rewritten pattern (right).

Every such nonatomic pattern in the Petri Net is rewritten, by replacing the start and end transitions in the pattern by a single transition for executing the nonatomic action. Figure 3b shows the rewritten pattern shown in Figure 3a.

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. The `rewrite-nonatomic-patterns` operation will attempt to merge the starts and ends of nonatomic actions into single transitions, on the intermediate Petri Net representation. By doing so, these merged transitions will be turned into single UML action nodes in later steps of the synthesis chain.

Note that, instead of rewriting nonatomic patterns on Petri Net level, it would also be possible to rewrite them on UML activity level. The advantage of rewriting on the level of Petri Nets, is that Petri Nets have a nicer structure than UML activities since they do not have fork/join/decision/merge nodes, making it easier to find and replace structural patterns in them.

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 nonatomic action A and 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 place in Figure 3a will have an extra outgoing arc to B, breaking the nonatomic pattern. In such cases, we do not merge the start and end transitions of nonatomic actions, but keep them separated. These transitions are later translated to separate UML action nodes, to be further handled during activity post-processing (**??**).

Preconditions. This operation requires a Petri Net resulting from the previous step. Moreover, this operation needs to know what Petri Net transitions start a nonatomic action, and what the corresponding end transitions are for ending these actions.

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

3.9 Transforming the Petri Net to an activity

The operation `transform-to-activity`(*petriNet*) transforms the given Petri Net *petriNet* in PNML format to a concrete UML activity. This operation takes care of unmerged nonatomic patterns, i.e., ones that were not rewritten by `rewrite-nonatomic-patterns`, by translating their start and end transitions to UML opaque actions with the proper guards and effects. Furthermore, this operation performs some post-processing: (1) it removes any internal actions, notably `__start` and `__end`, (2) it further simplifies the synthesized activity, and (3) it removes some names from UML control nodes and control flows, for better readability.

Motivation. The overall aim of the activity synthesis chain is to synthesize concrete 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 a concrete UML activity using `transform-to-activity`. This activity is then post-processed to improve its readability. Internal actions are removed since they were only needed to be able to synthesize the activity. Then the activity is further simplified. 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 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. The input *petriNet* must be in PNML format. This Petri Net may be, but does not have to be, free-choice. 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`. Finally, the Petri Net must not contain transitions corresponding to control nodes of any concrete activities that are called by the to-be-synthesized activity. Adding support for synthesizing hierarhical activities is planned as future work.

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

- Exactly one UML initial node and exactly one UML final node, which have been translated from the two places described above in the preconditions paragraph.
- For every transition in *petriNet* that performs either an atomic action or a nonatomic action for which the nonatomic pattern has been merged, a UML call behavior node that calls the UML opaque behavior of the action to perform. 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 to call.
- For every start transition of an unmerged nonatomic pattern in *petriNet*, an UML opaque action node is created to start the nonatomic action. This UML opaque action node is atomic, takes over the guard of the nonatomic action, and has no effects. The name of this opaque action node will be the nonatomic action name, followed by "`_start`". For example, `A_start`.
- For every end transition of an unmerged nonatomic pattern in *petriNet*, an UML opaque action node is created to end the nonatomic action. This UML opaque action node is atomic, has guard `true`, and takes over the effect of the nonatomic action. In case the nonatomic action was also nondeterministic, the right effect is taken over. The name of this opaque action node will be the nonatomic action name, followed by "`_end`", followed by the effect index. For example, `A_end_1` and `A_end_2`.
- 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 UML control nodes that can have multiple incoming control flows are join and merge nodes. The only UML control nodes that can have multiple outgoing control flows are fork and decision nodes.

- 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 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.

Finally, `transform-to-activity` ensures that the translated UML activity no longer contains internal actions and is simplified. Moreover, all names of control nodes (initial/final/fork/join/decision/merge nodes) and control flow have been removed for better readability.

3.10 Computing control flow guards in the activity

The `compute-guards`($umlActivity_{concr}$) operation computes incoming and outgoing guards for the control flows of the given concrete activity $umlActivity_{concr}$. These guards are computed by first translating $umlActivity_{concr}$ to a CIF specification, and then using CIF data-based synthesis to compute extra guards for the controllable events of that specification, which are the start events for executing the nodes of $umlActivity_{concr}$. Thus by doing so, for every activity node, we compute the extra guard that must hold to be able to safely execute that node. These extra guards are then translated to incoming and/or outgoing guards of the incoming and/or outgoing control flows of the activity nodes. Whether an incoming or outgoing guard is computed, or whether this computed guard is put on an incoming or outgoing control flow of an activity node, is dependent on the type of the activity node. This is further elaborated upon below.

Motivation. At this point in the activity synthesis algorithm we managed to synthesize the structure of the concrete UML activity. However, this activity does not contain any data (other than the guards and effects of actions). This is primarily because Petri Net synthesis is unable to deal with data. As a result, the (incoming and outgoing) guards that determine which control flow should be followed are missing at this point. Without control flow guards, the synthesized activity may have more behavior than allowed by the requirements, including potentially wrong or deadlocking behavior. Hence, we must compute the control flow guards and add them to the activity, which is what `compute-guards` does.

The `compute-guards` operation starts by translating $umlActivity_{concr}$ to a CIF specification in a similar way as described in Section 3.1. In fact, the same transformation `transform-uml-to-cif` is used for this, with some minor differences:

- We now give a concrete UML activity as input, rather than an abstract one. So instead of translating the UML opaque behaviors in the input model, we now translate the nodes and control flows of $umlActivity_{concr}$ (as described already in Section 3.1 for activities that are being called). However, we do not translate the initial node and final node, since execution of these nodes only has meaning when $umlActivity_{concr}$ is being called from some other activity, which is not the case here as $umlActivity_{concr}$ is considered in isolation. The UML-to-CIF translation ensures that there is a token on the single initial control flow—the one that goes out of the initial node of $umlActivity_{concr}$—by letting the corresponding CIF variable have `true` as the default value. Moreover, we do not translate any other concrete activities that might be in the input model, since $umlActivity_{concr}$ will not call any other activities since it is flattened.

- Two variants of the activity postcondition are now translated to CIF: one "without structure" and one "with structure". The "without structure" variant is the same as the postcondition translation as explained in Section 3.1. The "with structure" variant is extra, and is equivalent to the "without structure" variant plus the extra condition that only the final control flow—the one that goes into the final UML node of $umlActivity_{concr}$—holds a token. Both postcondition variants are translated to Boolean-typed algebraic variables in CIF. The algebraic variable for the "with structure" variant is used to indicate the marked states of the CIF specification.

- Extra CIF requirements are generated expressing that, whenever the activity postcondition holds, no further action nodes can be executed. Similar requirements were already generated before, as explained in Section 3.1. Here we slightly change them for the purpose of guard computation, since we must still allow the execution of control nodes whenever the activity postcondition holds, e.g., to execute join/merge/final nodes of $umlActivity_{concr}$ to finalize the execution. To achieve that, we require that the start events of all action nodes are disabled whenever the "without structure" variant of the activity postcondition holds, and that the start events of all control nodes are disabled whenever the stronger "with structure" variant of the postcondition holds.

- Every CIF event for starting the execution of some node in $umlActivity_{concr}$ is translated as a controllable event. This is different from before: Section 3.1 describes that, when calling a concrete activity from a to-be-synthesized activity, only the start events of initial nodes are controllable, and the others are all uncontrollable. However, $umlActivity_{concr}$ does not call other activities, as it is flattened. Our goal now is computing the extra conditions needed to safely execute the nodes of $umlActivity_{concr}$, hence we make all their start events controllable. By doing so, we are able to compute control flow guards as locally as possible.

- The translation of occurrence constraints is slightly different, to be able to correctly handle unmerged nonatomic patterns. In particular, for every nonatomic action that is subject to an occurrence constraint, the translated CIF requirement automaton for the constraint should consider: (1) the CIF events created for the call behavior nodes in $umlActivity_{concr}$ that call the UML opaque behavior of that action, as well as (2) the CIF start events of the unmerged nonatomic patterns involving that action.

After translating $umlActivity_{concr}$ to a CIF specification, extra guards are computed for the CIF start events for the activity nodes, using CIF data-based synthesis. Data-based synthesis is configured to do forward reachability (`--forward-reach=true`) and to start with forward reachability in the fixed-point order (`--fixed-point-order=reach-nonblock-ctrl`). So far, this configuration seems to perform best in practice. However, we have not yet properly validated this, so we are unsure whether this is the best fixpoint order strategy in general. We also configure data-based synthesis to not free certain Binary Decision Diagrams (BDDs) that we need for guard computation, which are otherwise freed by the synthesis procedure. BDDs are used by data-based synthesis, but also by guard computation, to symbolically represent and conveniently manupulate sets of system states. Since guards are predicates over the system state, they can be represented as BDDs. We will therefore compute control flow guards on BDD level, after which we will convert them to CIF expressions.

At this point, we have used CIF data-based synthesis to compute the controlled-behavior predicate as a BDD, as well as extra guards for the controllable CIF events, also as BDDs. The next step is using these BDDs to compute the extra guards needed for executing the activity nodes, after which we turn these extra guards into incoming and/or outgoing control flow guards. The extra guards for activity nodes are computed in the following way. Let *edge* be the CIF/BDD edge for starting the execution of some activity node. Then *edge* must have a controllable event. The extra guard `extra`(*edge*) for *edge*, i.e., the extra guard for executing the activity node, is essentially[6] defined as:

`extra`(*edge*) = `simplify`(`and`(`guard`(*edge*), *ctrlBeh*), `and`(`origGuard`(*edge*), *ctrlBeh*))

where:

- `origGuard`(*edge*) is the original guard for *edge* that was used as input for CIF data-based synthesis, as a BDD.
- `guard`(*edge*) is the strengthened guard for *edge* that was produced as output of CIF data-based synthesis, as a BDD.
- *ctrlBeh* is the controlled-behavior predicate computed by CIF data-based synthesis, as a BDD.
- `and` is a binary BDD operation that computes the conjunction of the two given BDDs.

[6]This definition is slightly simplified. The actual definition additionally eliminates internal variables using existential quantification. We come back to that later.

- `simplify` is a binary BDD operation that attempts to simplify the structure of the first given BDD by assuming the second BDD. This simplification is done to produce more readable control flow guards.

To understand this computation, let S be the set of all controlled system states in which the uncontrolled system guard of *edge* holds. This set S is captured exactly by the BDD `and`(`origGuard`(*edge*), *ctrlBeh*). Now we must find the subset $S' \subseteq S$ of these system states from which the application of *edge* always ends up in a controlled system state. Finding this subset S' is important since the activity node corresponding to *edge* must not be executed in any system state $S \setminus S'$, and `extra`(*edge*) should capture exactly that. To compute S' as a BDD, we have to reason about applications of *edge*. Luckily, CIF data-based synthesis has already done this for us, as it computed `guard`(*edge*), which ensures that any application of *edge* ends up in a controlled system state. Thus, the set S' is captured exactly by the BDD `and`(`guard`(*edge*), *ctrlBeh*). Note that the construction of this BDD still requires a conjunction with *ctrlBeh*. This is to ensure that the subset relation $S' \subseteq S$ indeed holds, which is needed for ensuring that `simplify` produces a correct extra guard[7]. Now that we have BDDs representing the sets S and S', we can compute `extra`(*edge*) by means of BDD simplication: assuming we are in a system state from S, what is the extra condition needed to ensure that we are also in a system state from S'? This extra condition is thus defined as the 'simplication' of the BDD representation of S' with respect to the BDD representation of S.

Note that, due to the way `extra`(*edge*) is defined, we assume that *edge* is only taken from controlled system states, i.e., states from *ctrlBeh*. The correctness argument of `extra`(*edge*) is inductive: assuming we execute the activity in a controlled manner up to the execution of the activity node represented by *edge*, we now have to compute the extra guard that ensures that the execution of *edge* ends up in a controlled system state again. If we do this consistently, then every activity execution will be safe (under the control flow guards that we compute for the activity, via `extra`) by induction on the length of the execution trace. We thus stay within the controlled system behavior. As an alternative, we could have defined `extra`(*edge*) in a way that does not rely on this assumption. Then `extra`(*edge*) would have been defined as `simplify`(`guard`(*edge*), `origGuard`(*edge*)). This definition would be correct in theory, but in practice it would lead to much duplication in computed control flow guards. In particular, we have seen that, if some control flow in the synthesized activity would get some non-trivial guard, then all subsequently reachable control flows would also get that guard. We now prevent this duplication.

Moreover, the definition of `extra`(*edge*) as given earlier is slightly simplified. The actual definition also ensures that the computed extra guard does not de-

[7] CIF data-based synthesis guarantees that any execution of *edge* from a controlled system state in which `guard`(*edge*) holds end up in a controlled system state. However, the predicate `guard`(*edge*) may itself describe more system states than just the controlled ones. In other words, the set of states described by `guard`(*edge*) is not necessarily a subset of *ctrlBeh*. The extra `and` computation is to ensure this subset relation. Without it, the `simplify` operation may assume too many states, which has shown to lead to incorrectly computed extra guards.

pend on any internal CIF variables that were introduced by the UML-to-CIF transformation, like the atomicity variable. Such a thing can be ensured because there might be multiple valid ways to `simplify` a BDD, leading to different results which are all correct. In other words, a correct simplification result is not necessarily unique. In practice it might happen that multiple extra guards exist for *edge* that are all correct, some of which are expressed over internal CIF variables and some of which are not. And as a result of the `simplify` operation, it might happen that `extra`(*edge*) becomes a predicate that does depend in internal state. We could have avoided that, by eliminating all internal CIF variables from the BDD representations of S and S' using existential quantification. By doing so, `extra`(*edge*) is guaranteed not to depend on internal state. However, as a downside to this, it might then not always be possible to compute extra guards. Sometimes the only way of expressing the extra guard of *edge* is via internal state. The code implementation checks for such internal state dependencies. The activity synthesis algorithm terminates exceptionally when such a case is detected, resulting in an error message for the user. With the addition of existential quantification, the full definition of `extra`(*edge*) now becomes:

$$\begin{aligned}\texttt{extra}(\mathit{edge}) = \texttt{simplify}(&\\ &\texttt{exists}(\texttt{and}(\texttt{guard}(\mathit{edge}), \mathit{ctrlBeh}), \mathit{internalVars}),\\ &\texttt{exists}(\texttt{and}(\texttt{origGuard}(\mathit{edge}), \mathit{ctrlBeh}), \mathit{internalVars}))\end{aligned}$$

where *internalVars* is a BDD representing the set of internal variables to eliminate, and `exists` is the BDD operation that computes the existential quantification of a given BDD over a set of variables also given as a BDD.

Finally, now that we have computed extra guards `extra`(*edge*) for every CIF/BDD edge *edge* with a controllable event, we must now translate these extra guards to incoming and/or outgoing control flow guards in the activity. Recall that *edge* that been created by the UML-to-CIF translator to start the execution of some activity node. Let *umlNode* be the UML activity node for which the CIF/BDD edge *edge* has been created. Then we can translate `extra`(*edge*) to incoming/outgoing guards of the incoming/outgoing control flows of *umlNode*. This translation is done by case distinction on the activity node type:

- If *umlNode* is a fork node, then `extra`(*edge*) will become the outgoing guard of the single incoming control flow of *umlNode*. By doing so, *umlNode* can only be executed if it can take a token from its single incoming control flow, to which we have added the extra guard.

- If *umlNode* is a fork node, then `extra`(*edge*) will become the outgoing guard of the single incoming control flow of *umlNode*. Note that an execution of *umlNode* is atomic and has no guards nor effects. Therefore, *umlNode* can only be executed if it can take a token from its single incoming control flow, to which we have added the extra guard.

- If *umlNode* is a join node, then `extra`(*edge*) will become the incoming guard of the single outgoing control flow of *umlNode*. Note that an execution of *umlNode* is atomic and has no guards nor effects. Therefore,

umlNode can only be executed if it can put a token on its single outgoing control flow, to which we have added the extra guard.

- If *umlNode* is a decision node, we may have multiple extra guards, since the UML-to-CIF translator created a controlled CIF event for every decision branch, i.e., for every outgoing control flow. Then `extra`(*edge*) will become the incoming guard of the outgoing control flow of *umlNode* for which *edge* was created. Note that an execution of *umlNode* is atomic and has no guards nor effects. Therefore, the execution of *umlNode* can only follow a certain decision branch if a token can be put on that outgoing control flow, to which we have added the extra guard.

- If *umlNode* is a merge node, we may have multiple extra guards, since the UML-to-CIF translator created a controlled CIF event for every merge branch, i.e., for every incoming control flow. Then `extra`(*edge*) will become the outgoing guard of the incoming control flow of *umlNode* for which *edge* was created. Note that an execution of *umlNode* is atomic and has no guards nor effects. Therefore, the execution of *umlNode* can only follow a certain merge branch if a token can be taken from that incoming control flow, to which we have added the extra guard.

- If *umlNode* is the initial node, then the situation is slightly different, since we have not created a controllable CIF event for its execution[8]. Instead, in the CIF translation, we made sure via the initials predicate that initially there is a token on its outgoing control flow. So guard computation works slightly different here, but still follows the same principles. Revisiting our earlier explanation of `extra`; we now let S be the set of all (controlled or uncontrolled) system states in which the the activity precondition holds and in which there is a token on the single outgoing control flow of the initial node. And we let S' be the set of all controlled system states that are in S. Then clearly $S' \subseteq S$, thus we can compute an extra guard, which expresses the extra condition needed on top of the activity precondition for calling the activity, i.e., for putting a token on the single outgoing control flow of the initial node. The extra guard thus becomes an incoming guard of that control flow. Note that S may contain uncontrolled system states. This has to do with the inductive correctness argument: the extra incoming guard should ensure that we start activity execution in a controlled system state, so that the inductive correctness argument applies.

- If *umlNode* is the final node, then the situation is slightly different as well, since we have not created a controllable CIF event for its execution[9]. Instead, in the CIF translation, we made sure via the marked predicate that activity execution is truly finished when (1) there is a token on the

[8] This is because the execution of an initial node also involves another activity that calls it, to transfer the token to the called activity.

[9] This is because the execution of a final node also involves another activity that called it, to transfer the token back to the calling activity.

incoming control flow of *umlNode*, (2) the activity postcondition holds, and (3) no other control flow holds a token. So guard computation works slightly different here, but still follows the same principles. Revisiting our earlier explanation of `extra`; we now let S be the set of all controlled system states in which the single incoming control flow of *umlNode* has a token. And we let S' be the set of all system states in S in which the activity postcondition holds. Note that S now contains only controlled system states, which is unlike the explanation for initial nodes above. This is because of the inductive correctness argument: every execution step ends in a controlled system state, including the step to finalize the execution of this activity. Yet we may only 'safely' finalize the activity in a system state where the activity postcondition holds, which is captured by S'. Thus $S' \subseteq S$. So we can compute an extra guard, which expresses the extra condition needed to be able to safely finalize activity execution. However, due the extra state/event exclusion invariants that we generated to prevent the activity to perform any more actions when its postcondition holds, this extra guard will always be trivially `true`. This expectation is also checked in the code implementation with an assertion. So the incoming control flow of *umlNode* will never get a non-trivial extra guard.

As a last step, to prevent BDD memory leaks, we free all BDDs that have not been freed by data-based synthesis, due to our configuration. Moreover, a final check is done over the synthesized activity to see whether it has nondeterministic choices. An activity has nondeterministic choices when it has decision nodes where multiple decision branches can become enabled in the same system state. We aim to synthesize deterministic controllers, i.e., UML activities with deterministic behavior. So a user warning will be given for every found nondeterministic choice.

Preconditions. The input UML activity $umlActivity_{concr}$ should be a synthesized concrete UML activity, with its nonatomic patterns reduced wherever possible as result of Section 3.8. The input activity must not call other activities, i.e., it must be flattened, which is ensured already by the previous steps.

Moreover, tracing information must be available indicating how this concrete activity relates to the input UML model, as well as to all the intermediate steps/representations that were needed to synthesize this concrete activity. This tracing information is needed, e.g., for finding all CIF events that have been created for the nodes of $umlActivity_{concr}$.

Postconditions. The output is a UML activity that is equal to $umlActivity_{concr}$, but with proper incoming and outgoing control flow guards added to it. These control flow guards ensure correctness: any execution of $umlActivity_{concr}$ is guaranteed to not deadlock nor to reach a system state that violates the requirements. The added incoming and outgoing control flow guards are Boolean conditions that express extra conditions for placing a token on a control flow, or for taking a token from a control flow, respectively. These conditions are

extra in the sense that they are not yet captured by the guards of any actions performed by $umlActivity_{concr}$.

Moreover, the added control flow guards are guaranteed not use internal variables that were used for synthesizing $umlActivity_{concr}$, like the atomicity variable or occurrence constraint variables. However, it might happen that $umlActivity_{concr}$ requires control flow guards that cannot be expressed in a way that is independent of internal state. In that case, the synthesis algorithm will terminate exceptionally. We have shown that this can happen with some rather artificial synthesis examples. So far we have not experienced such problems with more realistic synthesis examples.

User warnings are given for any nondeterministic choices in the synthesized activities with control flow guards.

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.

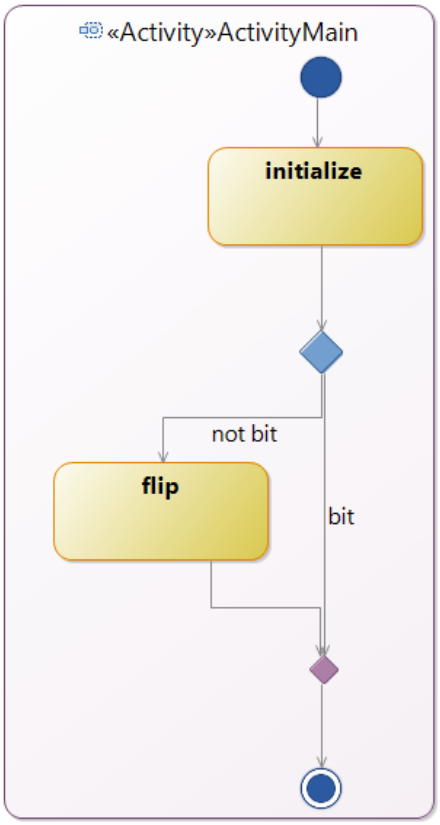


Figure 4: The synthesized activity.