Synthesis of Poka Yoke Activity Diagrams

October 3, 2024

1 Introduction

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

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

2 Preliminaries

2.1 Formalisms

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

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

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

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

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

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

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

- Any number of UML properties. UML properties have a type and optionally a default value. There are three supported types: Booleans, bounded integers, and enums. Default property values are then expressions of the appropriate type.
- Any number of UML opaque behaviors that have exactly one *guard* and zero or more *effects*. These opaque behaviors model *actions* to be performed in synthesized UML activities. Likewise to EFAs, action guards are state predicates, and action updates are state transformers, both of which are expressed over the UML properties as defined in the UML model. Any action is defined to be *deterministic* if the action has at most one effect. Any 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.
- A 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 also be an action exclusion invariant, which indicates that the execution of some action requires the system to be in a certain state as expressed by a state invariant.
- A single UML activity, which can either be *abstract* or *concrete*.
  - 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[2] are roughly of the form 'action A must happen at least M times and at most

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

[2] Note that occurrence constraints are temporary, and intended to be removed later.

N times' and thus limit the number of occurrences of some action in a to-be-synthesized activity. Abstract UML activities are 'empty' in the sense that they contain no nodes nor control flow.

- A concrete UML activity is a finite directed graph consisting of *nodes* and *control flows*[3]. A node can be a *control node* (initial, final, fork, join, decision, or merge node), be an *call opaque behavior node* which executes an action by calling an opaque behavior, or be an *opaque action node* containing a guard and zero or more effects, thereby essentially 'inlining' an action. Concrete UML activities cannot contain preconditions, postconditions, or occurrence constraints.

The activity synthesis algorithm takes UML models with an abstract UML activity as input. The goal of activity synthesis is then to synthesize a concrete UML activity for the specified abstract UML activity, and updating the UML model by replacing the abstract UML activity by the concrete synthesized ones.

We will only consider UML models that are *valid* with respect to the Poka Yoke validator. Being valid for example means that the UML model contains no abstract UML activities that have nodes or control flows.

Petri Nets. A Petri Net is a finite directed graph consisting of *places*, *transitions*, and *arcs*. We consider *elementary Petri Nets*, where (by definition) any place holds at most one *token*. Transitions in a Petri Net can be *fired* to move tokens around between adjacent places. If some transition can fire, we say that it is *enabled*. Any arc in a Petri Net must be connected to a place with a transition. That is, there cannot be an arc from a place to some other place, or from a transition to some other transition. Further details on Petri Nets and their semantics can be found in the standard literature.

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

2.2 Standard and tooling

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

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

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

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

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

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

2.3 Algorithms for synthesis

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

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

The field of Petri Net synthesis is developed around the *theory of regions*. Intuitively, the idea is to find groups of locations in the input state machine that can be turned into single places in the output Petri Net. These groups are then called regions. Depending on the type of Petri Net you aim to synthesize (e.g., general or free-choice ones), there are slightly different requirements of

Algorithm 1: Synthesis of Poka Yoke activity diagrams

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

what exactly constitutes a region. Petrify is essentially an implementation of the theory of regions, and is able to synthesize both general and free-choice Petri Nets (among other kinds of Petri Nets).

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

3 Activity Synthesis

Algorithm 1 shows the high-level activity synthesis algorithm. The algorithm is a sequence of operations performed on some input UML model, *umlModel*, that contains an abstract UML activity. The algorithm produces a UML model where the abstract UML activity is replaced by the synthesized concrete activity.

The remainder of this section explains all operations in Algorithm 1. A short explanation is given for every operation, followed by the motivation for having that operation, followed by its preconditions and postconditions.

3.1 Transforming the UML model to CIF

The `transform-uml-to-cif`(*umlModel*) operation translates a given UML model, *umlModel*, to a CIF specification to be used for supervisory controller synthesis.

Motivation. With respect to supervisory controller synthesis, the input UML model specifies the plant and requirements, i.e., the UML opaque behaviors and UML (occurrence) constraints. These plant and requirement constructs are translated one-to-one to CIF, to enable running its data-based synthesis tool.

Preconditions. This operation requires the input UML model to be valid and have an abstract UML activity (see also Section 2.1). The abstract UML activity is specified up front since it contains, e.g., the name of the to-be-synthesized activity, as well as its preconditions and postconditions.

Postconditions. This operation produces a CIF specification that:

- For every UML enum declaration, contains a corresponding CIF enum declaration.
- Contains a CIF plant for the single UML class. This plant is a flower automaton, containing one location 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 deterministic actions are translated as single controllable events. All opaque behaviors that model nondeterministic actions are translated to multiple CIF events, namely a controllable one for starting the action, and uncontrollable ones for each of their nondeterministic effects to end the action. Such separate uncontrollable events must be defined since data-based synthesis in CIF disallows controllable events that can happen nondeterministically. So, for data-based synthesis, we need a controllable event to (controllably) start some nondeterministic action, and uncontrollable events to nondeterministically perform one of its effects. Moreover, in case the UML model contains nondeterministic actions, an internal *atomicity variable* is declared and maintained in the CIF specification, to ensure that no event may occur between the start and end event of a nondeterministic action.
- Contains an 'initial' predicate from the conjunction of all translated preconditions of the abstract UML activity. There may be multiple preconditions defined for this activity. Each of these preconditions are translated as Boolean algebraic variables in CIF, for better traceability. The conjunction of all these Boolean algebraic variables then forms the activity precondition, for which an algebraic variable is created as well. This algebraic variable is then used as the 'initial' predicate, which limits the number of initial states to only the ones satisfying the precondition.

- Contains a 'marked' predicate from the conjunction of all translated postconditions of the abstract UML activity. There may be multiple postconditions defined for this activity. Each of these postconditions are translated as Boolean algebraic variables in CIF, for better traceability. The conjunction of all these Boolean algebraic variables then forms the activity postcondition, for which an algebraic variable is created as well. Moreover, in case there are nondeterministic actions, an extra implicit postcondition is generated, stating that no nondeterministic action must be active (with respect to the atomicity variable explained earlier) for the postcondition to hold. This combined postcondition is then used as the 'marked' predicate.

- Contains requirement invariants stating that the postcondition disables any CIF event. In other words, if you would reach a system state where the activity postcondition holds, then no further actions would have to be taken as they will not contribute to coming closer to a postcondition state. These requirements can be seen as an optimization for synthesis and later state space generation, to avoid considering unnecessary steps. And they ensure that the synthesized UML activity will have no further actions after having reached a final node.

- Contains an edge in the flower automaton plant for every defined CIF event declaration. An edge is defined for every declared CIF events, both controllable and uncontrollable ones. Every controllable CIF event corresponds one-to-one to UML opaque behavior definitions, and thus their edge guards are the translated action guards. Moreover, in case of a deterministic action, the edge update is the translated action effect. In case of a nondeterministic action, the action effects are translated on the corresponding uncontrollable events instead. The edges for uncontrollable events do not have guards. We don't want conditions on any 'end' events, and by making them uncontrollable, any synthesized constraints are pushed back to the corresponding controlled 'start' events. These control flow guards and updates also correctly handle and maintain the atomicity variable.

- Contains requirement automata for the occurrence constraints defined in the UML model. Occurrence constraints are roughly of the form 'action A must happen at least M times and at most N times'. Such constraints are translated as requirement automata, containing a discrete variable that maintains how often the action has already occurred. This variable can then be incremented every time the action occurs. The occurrence requirement can then be expressed over this requirement (using edge guards and marked predicates).

3.2 Data-based synthesis with CIF

The operation `data-based-synthesis`(*cifSpec*) executes the data-based synthesis tool of CIF. Data-based synthesis is thereby configured to do forward

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

Motivation. Our goal of performing data-based synthesis is to compute all extra restrictions on actions that must be considered by the to-be-synthesized activity to never violate specified requirements. Data-based synthesis computes a minimally restrictive supervisor for going from a state that satisfies the activity precondition, to a state that satisfies the postcondition, without violating requirements, running into blocking situations, etc. We will later make the behavior of this supervisor explicit (Section 3.3), to be able to synthesize a compact Petri Net for it (Section 3.7) that is then transformed to a concrete activity (Section 3.8).

After this transformation, we need to separately compute the guards of the control flows that go out of UML decision nodes (Section 3.9). This computation make use of the extra restrictions that are synthesized by data-based synthesis. Moreover, this computation requires that these extra restriction predicates are *not* simplified, which CIF does by default, hence the extra configuration.

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

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

Postconditions. All guarantees of the data-based synthesis tool apply. Moreover, since BDD predicate simplification is disabled, the resulting CIF supervisor contains no requirement invariants. These requirements are instead included in the synthesized conditions. The guards of the supervisor automaton, for each controllable event, are the full/complete conditions under which these controllable events are enabled in the controlled system.

3.3 State space generation

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

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

Petrify requires a DFA or NFA (i.e., a traditional supervisor in the form of a single DFA/NFA), rather than an EFA. Petrify does not support variables.

The state space that is generated by `generate-statespace` from the synthesized supervisor expresses all possible orderings of events, taking into account the extra synthesized guards and the original action guards as specified in the UML model. The goal of Petri Net synthesis is then to find a compact Petri Net representation of all these possible orderings, whose structure can then be translated to a concrete UML activity.

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

Preconditions. All preconditions of the CIF explorer tool apply.

Postconditions. All guarantees of the CIF explorer tool apply. The CIF explorer produces a CIF specification that contains exactly one automaton—the CIF state space. Noteworthy is that this state space will have state annotations `@state(...)` that indicate the values of every variable in every location, as well as the current location of each automaton that has at least two locations. This information will later be used for computing control flow guards (in Section 3.9).

Moreover, due to the way our UML/CIF input for synthesis is constructed (as result of Section 3.1), the resulting state space has the following properties:

- All initial locations in the state space correspond to states that satisfy the precondition of the to-be-synthesized activity.
- All marked locations in the state space correspond to states that satisfy the postcondition of the to-be-synthesized activity.
- 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.
- For every location it holds that all events on outgoing edges are either all controllable, or all uncontrollable. This property is a consequence of the atomic execution semantics of actions. If a nondeterministic action

is being executed in some location in the CIF state space, then by the atomicity constraint the only thing that could happen is an uncontrollable event to finish the atomic action. And conversely, if no nondeterministic action were being executed, then only controllable events can be executed to start some new action (given that the location is not marked).

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. This means that all uncontrollable events are projected away, i.e., *cifStatespace* is transformed to a DFA without keeping any uncontrollable events.

Motivation. Recall that uncontrollable CIF events were created for nondeterministic actions (in Section 3.1), to model the nondeterministic execution of their effects. However, these uncontrollable events and their corresponding edges are an internal, intermediate step that should not be visible in the synthesized UML activity. So at some point, these uncontrollable events have to be eliminated. This elimination should be done before Petri Net synthesis due to the atomicity constraint. To clarify this further, recall that the goal of Petri Net synthesis is to find a minimal Petri Net whose behavior is trace-equivalent to the CIF state space that was given as input to Petri Net synthesis. While doing so, Petri Net synthesis aims to reduce diamond 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 do not give perfect diamond patterns, since whenever some nondeterministic action is being executed, the atomicity constraint enforces that no other action can be performed, thus impacting interleaving. Therefore, we perform event-based projection on the CIF state space, to restore the diamond patterns that got disrupted by the atomicity constraint, before performing Petri Net synthesis.

Note that Petrify, which is the tool we use for Petri Net synthesis, also has a built-in option `--hide` to hide a list of given events. We used this option in

earlier versions of our implementation instead of doing event-based projection on the level of CIF. However, Petrify's hiding option seems broken, in the sense that we observed that Petrify does not always hide all events in the specified list. This further motivates doing hiding/projection on the level of CIF instead.

Preconditions. Since `event-based-projection` uses the automaton projection tool that comes with CIF[4], 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 those events. In our case, this means that the resulting CIF specification no longer contains the uncontrollable events, and is trace-equivalent to the input modulo those events. The resulting DFA is not guaranteed to be minimal. We will minimize it in the next step.

3.6 DFA minimization

The `dfa-minimization`($cifStatespace_{proj}$) 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, such that we get the smallest possible UML activity that is still optimal.

Moreover, note that Petrify, which is the tool we use for Petri Net synthesis, also has a built-in option `--mints` to minimize the input graph modulo trace equivalence. From the documentation of Petrify[5] this option seems to have some interaction with the `--hide` option. However, as explained in Section 3.5, `--hide` does not seem to always work. Therefore, in addition to event-based projection, we also do the minimization on the level of CIF.

Preconditions. Since `dfa-minimization` uses the event-based DFA minimization tool that comes with CIF[6], 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

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

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

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

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.

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

We use the Petrify tool for performing Petri Net synthesis. 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.

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

Motivation. By having computed a (projected and minimal) CIF state space, *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 3.8). We thus use Petri Nets as an intermediate formalism in our activity synthesis algorithm.

Moreover, there are particular classes of Petri Nets, *free-choice* Petri Nets, that have shown to lead to more intuitive UML activities (although we haven't yet done an extensive study or comparison to see whether this is always the case). Therefore we attempt to synthesize a free-choice Petri Net whenever possible.

However, translating a Petri Net to a UML activity structure is not yet sufficient: we then still need to compute the control flow guards for the decision nodes of the synthesized UML activity. These control flow guards are needed, since without them any UML decision node could freely follow any of its outgoing control flows, leading to executions that are not allowed by the synthesized CIF supervisor. This step must be done separately (in Section 3.9), since to the best of our knowledge, Petri Net synthesis with data/state is an open research field for which tooling is not available. Therefore, we must be able to relate the output of Petri Net synthesis to the input specification, $cifStatespace_{min}$. Such a relation is (or should be) produced by the Petri Net synthesis algorithm, and is called a *region mapping*. The reason for this, is that Petri Net synthesis is based on the *theory of regions*. Without going too deep into this theory; the main idea is to group locations from $cifStatespace_{min}$ together so that every such group (roughly) corresponds to a Petri Net place. These groups are then called *regions*. So the Petri Net algorithm can produce a mapping from Petri Net places to the regions from which they are formed, thereby providing an input-output relation. With this relation, we can later find the relevant data/state and synthesized guards on the level of CIF, for the Petri Net places that will become UML decision nodes, and from those calculate the control flow guards. Section 3.9 explains this further.

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

identifiers containing a dot.

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

Furthermore, the event name `__loop` must not be in the automaton alphabet. This is because `petrinet-synthesis` will create an auxiliary self-loop edge named `__loop` before Petrify is invoked, which is removed later from the output of Petrify. This auxiliary self-loop is added to make Petrify work. This is because Petrify seems unable to do Petri Net synthesis in case the input automaton has deadlock locations, i.e., locations without outgoing edges. And recall that $\mathit{cifStatespace}_{min}$ has a single marked location which is a sink location, and thus when you get into this location during some execution, the execution deadlocks. Hence, to resolve this, we add an extra self-loop edge that connects the single marked location to itself, to still allow 'progress' from this sink location. After having invoked Petrify, the synthesized Petri Net will contain exactly one transition for `__loop` (since the input automaton has exactly one marked location and we added exactly one `__loop` edge), so it's easy to remove it again.

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

Postconditions. All guarantees from Petrify apply. In particular, the output is an optimal (possibly free-choice) Petri Net that is trace-equivalent to the input automaton. The synthesized Petri Net is returned in PNML format.

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

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

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

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

3.8 Transforming the Petri Net to an activity

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

Motivation. The overall aim of the activity synthesis chain is to synthesize 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 UML activity using `transform-to-activity`.

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

The input *umlModel* must be the original UML model that was given as input to the activity synthesis algorithm. See also the preconditions described in Section 3.1. Notably, it must have a single abstract UML activity.

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

- Exactly one initial node and exactly one final node, which have been translated from the two places described above in the preconditions paragraph.
- One auxiliary opaque action named `__start`, which has been created for the start event introduced earlier, as explained in Section 3.4. Then the initial node has exactly one outgoing control flow to this `__start` action.
- An auxiliary opaque action named `__end` for every end event introduced earlier, as explained in Section 3.4.
- No other opaque action, apart from the `__start` and `__end` opaque actions. Then all these opaque `__end` actions have exactly one outgoing control flow that goes to the final node.
- Fork/join/decision/merge 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.

- Call behavior nodes for all transitions in *petriNet* that perform an action. These nodes then call the appropriate UML opaque behavior of the action. Any duplication that was introduced by label splitting (recall from Section 3.7) has been removed during the process of finding the right UML opaque behavior corresponding to a Petri Net transition.

- No guards on any control flows. Control flow guards are computed later, in Section 3.9.

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

3.9 Computing control flow guards in the activity

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

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

Given any control flow *cf* in the UML activity that goes from some decision node D to some node N, the guard for this control flow *cf* can be computed as `simplify`(`simplify`(N_{cond}, N_{guard}), D_{pred}), where:

- N_{cond} is the original condition for executing N as specified in the input UML model. In case N is an action node, N_{cond} is the original guard predicate of the action as defined in the input UML model. In case N is a final node, N_{cond} is the activity postcondition, i.e., the condition needed to 'finalize' the activity. For all other types of activity nodes, the N_{cond}

predicate is simply `true`, since no enabling condition is explicitly defined in the input UML for fork/join/decision/merge/initial nodes.

- N_{guard} is the full synthesized condition predicate for N, under which N can be executed in the controlled system. In case N is an action node, N_{guard} is the full condition that is computed by data-based synthesis, under which the action is enabled in the controlled system. For all other types of activity nodes, the N_{guard} predicate is simply `true`, since data-based synthesis only computes extra conditions for controllable events, which correspond only to actions.

- D_{pred} is the predicate describing all states the system may be in at the point of taking/following cf.

- `simplify`(p, q) is an operation that simplifies some given predicate p with respect to some predicate q, and returns the simplified predicate. Simplification can be performed by assuming q, and removing parts of p that are covered by this assumption (i.e., if q were true, what remains of p?). This simplification is based on heuristics. Various algorithms exist for simplifying (symbolic representations of) predicates.

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

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

We still need to gather the necessary information from earlier steps. Firstly, for every action defined in $umlModel_{concr}$ we need to know its action guard, as well as the condition that was synthesized for it during data-based synthesis (Section 3.2). The former can directly be obtained from the UML model. The latter can be obtained from the CIF supervisor, by finding the synthesized condition for the controllable event that was defined for the action (Section 3.1).

Secondly, for every node in the synthesized UML activity, we need to know the set of all states in which the system may be at that point. These sets of states can be found by tracing back the CIF state annotations from the earlier generated CIF state space (see Section 3.3), through the various CIF event-based toolkit steps. So for every UML node we must: (1) determine the Petri Net place corresponding to it, (2) use the region mapping produced in Section 3.7 to determine which CIF locations correspond to that Petri Net place, and (3) collect the state annotations from these locations in the generated CIF state space. Step (3) is slightly tricky in case there were non-deterministic actions, as we should not consider the state annotations of any 'intermediate' CIF locations in which a non-atomic action is being executed (e.g., since the atomicity variable is true in any such intermediate state, and choice guards should not depend on internal details). Therefore, any such state annotations must be filtered out first. The set of all system states can then be represented as a single BDD, i.e., D_{pred}.

Preconditions. The input UML model $umlModel_{concr}$ should be the original UML model that was used as input to the activity synthesis algorithm, with its single abstract UML activity replaced by a concrete synthesized one (as result of Section 3.8). Moreover, all intermediate information from earlier steps of the activity synthesis algorithm must be available, like the synthesized Petri Net, the region mapping, the state annotations of the generated/projected/minimized CIF state space, the CIF supervisor, etc.

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

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

Moreover, the added choice guards do not use internal variables, like the atomicity variable. Actually, in truth, we are not entirely sure that the computed choice guards will indeed never use internal variables. However, we have added an assertion in the code to check for use of internal variables, and if such use is detected, the `compute-edge-guards` operation will terminate exceptionally. So if the operation terminates, then the choice guards do not use internal variables.

3.10 Post-processing of the activity

The operation `postprocess-activity`($umlModel_{guards}$) post-processes the synthesized activity in $umlModel_{guards}$, to remove any leftover auxiliary constructs,

notably the opaque `__start` and `__end` actions. It also removes some names from UML control nodes and control flows, for better readability in UML Designer.

Motivation. All auxiliary constructs that were not part of the original input UML specification should be removed from the synthesized result. These should not be visible to the user. Moreover, post-processing gives the opportunity to do some other minor improvements, for example to improve readability a bit.

Preconditions. This operation requires a UML model $umlModel_{guards}$ with a concrete synthesized UML activity, resulting from the earlier steps. The concrete UML activity in this model should have exactly one opaque action named `__start`, and at least one opaque action named `__end`.

Postconditions. This operation ensures that the synthesized UML activity no longer contains the auxiliary `__start` and `__end` opaque actions. In fact, since these were the only opaque actions in the UML activity (recall from Section 3.8), `postprocess-activity` ensures that the activity has no opaque actions.

Moreover, all names of control nodes (initial/final/fork/join/decision/merge nodes) and control flow have been removed for better readability.