# A Process to Support Forensic Readiness

In this document we discuss the process steps by which we reach to identify assets, relationships, and actions that can be potential evidence for future investigations, i.e. identify what is required to proactively collect and store to be forensic-ready. The process consists of four steps: match assets, generate predicates, identify potential incidents, and analyse potential incidents. These steps and their inputs and outputs are shown in the figure 1, and discussed in the next sections.

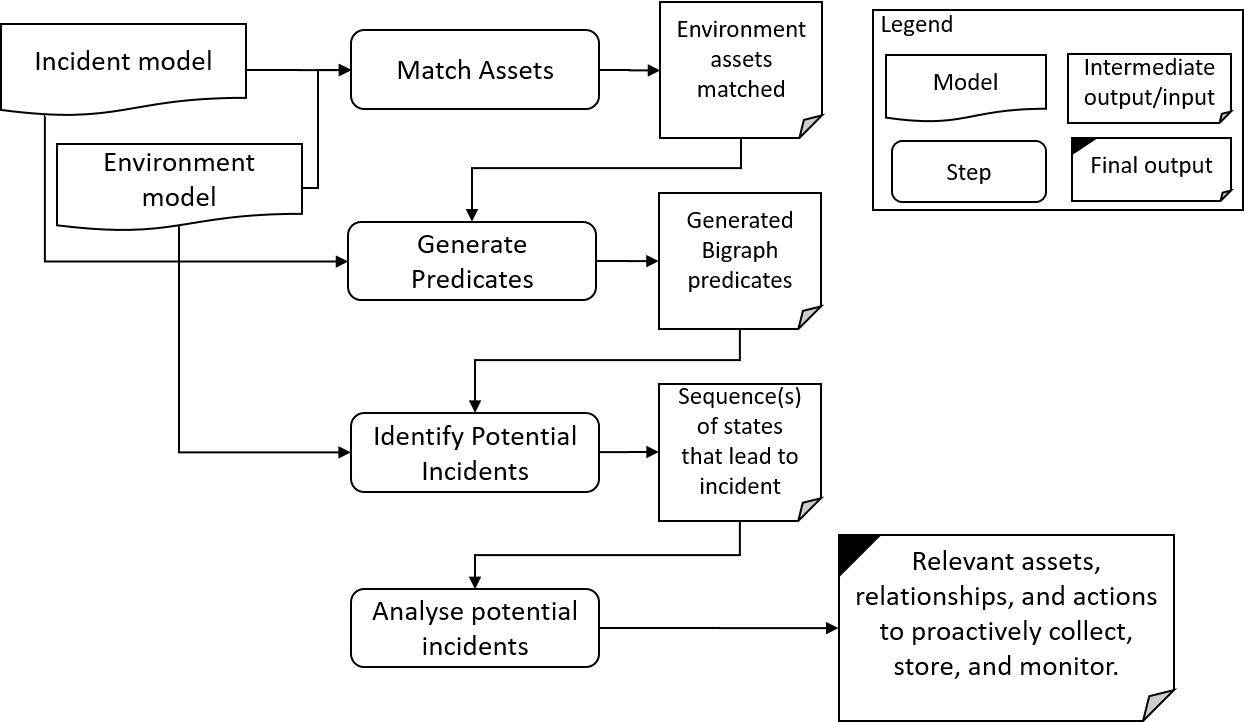


Figure 1 Process steps to identify assets, relationships, and actions for forensic readiness.

# Entity-Asset Matching

This is the first phase towards identifying assets, relationships, and actions of an environment that are relevant to potential incidents, and thus, potential investigations.

In this phase, entities found in a domain-independent incident are matched against assets defined in an environment model. The goal is to find all possible environment assets that match the incident entities. Matching is done based on several criteria, which are:

* Type of entity: matched environment assets should have the same type as the incident entities.
  + For example, an entity (entity1) that has a type “Smart Device” should be matched to all assets that have the same type.
* Type of entity’s parent: this refers to the parent entity that contains the entity to match to. Assets from the environment model should be contained in other assets that have the same type as the parent of the entity.
  + For example, entity1 *is contained in* entity2, and entity 2 has type “Room”.
* Entities contained: this refers to the entities that the target entity has. Match assets should contain the same number and type of assets.
  + For example, entity1 *contains* entity3 and entity4, both entities (3 & 4) are of type “Digital Data”.
* Status of entity: if found, the status of an entity is matched to statuses of assets.
  + For example, entity1 has status: OPEN.
* Properties of entity: if found, properties, which are defined as name-value pairs, are matched against assets’ properties, and assets that match these properties are selected.

Based on the matching criteria defined earlier, we perform the matching using XQuery language. XQuery language is used since it provides efficient and fast way to perform selection operations from and between XML files, which is the structure of the incident and environment models.

The output of this phase is the set of all possible asset matches for each incident entity found. In other words, if an incident model has 3 entities, for example, and an environment model has 7 assets, then an output would be:

*Entity1: {Asset1, Asset2, Asset5}*

*Entity2: {Asset2, Asset6}*

*Entity3: {Asset3, Asset5, Asset7, Asset8}*

If an entity has no matches, then either execution of the rest of the phases are stopped, or the closest asset to match the entity can be returned and execution can continue.

From the output we can find all possible combinations of environment assets that match to the set of entities. For example:

*{Entit1, Entity2, Entity3}*

*=*

*{Asset1, Asset6, Asset7}*

*or*

*{Asset1, Asset2, Asset3}*

After finding all possible asset sets that match the entities, the output is fed to next phase to generate predicates for Bigrapher. The next phase is discussed in the next section.

# Predicate Generation

The aim of this phase is to generate Bigraph predicates and to insert them into a Bigrapher file to be executed in the next phase.

Generation of the predicates is conducted as follows:

* All entities found in the preconditions and postconditions of each activity in the domain-independent incident are replaced with matched assets from an environment model. Each condition is represented as a Bigraph statement.
* For each condition a new object is created (from *Predicate* class) that hold information about the condition such as name, Bigraph statement, and activity.
* After that, the conditions are inserted into a Bigrapher file that represents the environment. We call the conditions *predicates*, since they become predicates in the Bigrapher file.
  + Validation of predicates before inserting them into the Bigrapher file is required to assure that assets in the predicates have equivalent in the Bigrapher file. For example, if an asset is named “Smartlight”, there should be a control defined in the Bigrapher file with the name “Smartlight”.
  + Two changes are made to the Bigrapher file. These are:
    - Addition of predicates as Bigraph statements i.e. **big** predicate\_name = *Bigraph\_statement*.
    - Update of predicates in the definition of the Bigraph i.e. **pred** = {predicate\_name}.

A new Bigrapher file is created that contains the new predicates. After the successful creation of the new file, the next phase starts, which is discussed next.

# Potential incidents Identification

The aim of this phase to identify states and state transitions that satisfy the preconditions and postconditions of the incident activity, and hence, identify possible ways in which an incident can take place in an environment.

The identification of states and transition is conducted as follows:

* After creating a new Bigrapher file that holds generated predicates, the file is evaluated and executed using Bigrapher software tool.
  + A successful evaluation and execution will produce a folder that contains the following:
    - Transition system file: which contains state transitions (e.g., 2->3).
    - States’ files: each state of the system is exported to a file that has the name *state\_number*.json (e.g., 2.json, 3.json). Currently, states are exported in JSON format, as it is easier to handle in java. Other formats are SVG, DOT, and TXT.
    - Predicate file: this file has holds the names of the predicates and the states that satisfy these predicates.
      * For example: label “predicate\_1” = x = 1 | x = 5 | x = 7;
* After successful execution, the *transition system* file is read, and a Digraph is created that holds the states and the transitions. The Digraph helps in determining state transitions.
  + Depth first search is used to determine transitions between two states.
* Then, *predicate* file is read, and Predicate objects are updated with the states that satisfies each one.
* After that, possible state transitions are found between different predicates. This is done as follows:
  + First, transitions are found between a precondition and a postcondition within an activity (hereafter, pre-post transitions). It is assumed that an activity holds one precondition and one postcondition.
    - States the satisfy both are updated to those that have transitions from the precondition to the postcondition i.e. states with no transitions are dropped.
    - If there are no transitions found between states of the pre and post conditions, then the activity that holds these conditions cannot be satisfied in this environment. Thus, the incident cannot happen in this environment.
  + Second, transitions from one activity to the next are found. This is done by finding transitions from the postcondition of a current activity to the precondition of the next activity (hereafter, post-pre transitions).
    - Using updated states that satisfy conditions internally, transitions between activities are found. Based on these transitions the states are updated again to include only those that have transitions from pre to post within an activity and transitions between post of a current activity and pre of the next activity.
    - If there are no transitions found, then there is no way that the incident can proceed in execution in this environment. Thus, the incident cannot happen in this environment.
  + Third, we combine pre-post and post-pre transitions together to find transitions that satisfy conditions from the precondition of a current activity to the precondition of the next activity.
  + Fourth, we define possible incident paths in the environment by combining state transitions from the initial activity to the final activity. A state transition from a current activity is combined with another from the next activity if the last state in the transition of current activity matches the first state of the transition of next activity.

# Potential Incidents Analysis

The aim of this phase is to identify assets, relationships, and actions that are relevant to the incident being mapped to the environment. This involves analysing possible state transitions that satisfy the preconditions and postconditions of each activity in an incident.

From previous phases, we did the following:

1. identified potential assets that match entities in the incident model.
2. Generated predicates (preconditions and postconditions): preconditions and postconditions of each activity are replaced with matched assets from the environment.
3. Extracted states that satisfy these predicates through executing a bigraph representation of the environment using Bigrapher that included the generated predicates.
4. Identified state transitions between states that satisfy the predicates.

Currently, we have identified all possible state transitions that satisfy the pre/postconditions of an activity and also transitions that link one activity to the next.

What is left is to analyse these transitions to reach our aim. To do so, we need to know what a state in Bigrapher contains. This is discussed in the next subsection.

## Bigrapher State

A state file generated by Bigrapher tool contains the following information:

* Nodes: A Bigrapher state has a definition of the nodes, where each node is defined by a node id, and a control and its arity. For example:

*{"node\_id": 4,*

*"control": {"control\_id": "M", "control\_arity": 2}}*

*{"node\_id": 3,*

*"control": {"control\_id": "Snd", "control\_arity": 0}}*

* Directed graph (place graph) which represents the containment. A containment is represented by the source -> target using nodes ids defined in the nodes above. The place graph also holds information about the number of regions, nodes, and sites. For example:

*{"source": 4, "target": 3}*, numbers refer to nodes ids

* Link graph representing the connectivity graph between nodes in that state. A connectivity is represented by inner and out interfaces. An interface such as outer interface is defined by a name and ports. Each port has a node id and a port arity (i.e. how many connections to that node are there).

*{"inner": [], "outer": [{"name": "v\_b"}], "ports": [{"node\_id": 4, "port\_arity": 1}]}*

Any change to state contents (e.g., dropping/adding a node/edge) will lead to the generation of a new state. Transitioning is done using reaction rules in Bigraph. However, currently Bigrapher tool does not label state transitions with the actions that caused the transition. Therefore, we implemented a way to do so, which is discussed in the next subsection.

## Identifying Actions for State Transitions

For actions, we can identify actions through state transitions, which I implemented using the *LabelExtractor* class since currently Bigrapher does not label transactions. However, the assumption made is that each reaction rule should have a unique keyword that is placed in the redex and dropped in the reactum. A file containing all reaction rules keywords separated by semicolon should also be provided. For example:

*react snd = A{a}.****Snd****.(M{a1, v} | id) | Mail -> A{a} | Mail.(M{a1, v} | id);*

*react ready = A{a}.****Ready*** *| Mail.(M{a, v} | id) -> A{a} | Mail | {v};*

keywords used = {Snd; Ready}

I have implemented another approach that does not require having unique keywords but it is computationally more expensive and it can return more than one possible action for a transition. The approach is to use the redex and reactum of a reaction rule as predicates in the Bigrapher file then extract states that satisfy them and identify states that are immediately a transition to each other depending on the transition system or the digraph object that represent the transition system.

## Identifying Relevant Parts of a State

We need to identify part(s) of a state that are relevant to the incident being investigated. We can extract assets and their relationships that concerns us from a given state depending on earlier generated predicates (phase-2), which can tell us what assets and relationships we are interested in.

Identifying which action took place helps in identifying relevant part(s) of stats. It can help identify nodes and their relationships at a state and the change that happened to that part in the next state.

There could be different parts that evolve/change over a state transition.