|  |  |
| --- | --- |
| Question | Answer |
| 3.1 | |  |  | | --- | --- | | Purpose | Petri nets are a graphical modeling language used to describe the behavior of concurrent, distributed systems. They excel at representing systems with:  Concurrency: Multiple events happening simultaneously.  Synchronization: Events needing to occur in a specific order.  Resource sharing: Limited resources being used by different parts of the system. | | Use Cases | System designers: They use Petri nets to model and analyze system behavior during the design phase.  System analysts: They use Petri nets to understand existing systems and identify potential problems.  Software developers: They use Petri nets to verify the correctness of concurrent software. | | Concepts | Places: Represented by circles, they denote states or conditions within the system. They can hold tokens.  Transitions: Represented by rectangles, they represent events or actions that cause the system to change from one state to another.  Tokens: Represented by black dots within places, they signify resources, data items, or control signals flowing through the system.  Arcs: Directed arrows connecting places and transitions. They define the flow of tokens and how events are triggered. | | Relations | Arcs connect places to transitions (input) and transitions to places (output). A transition can only fire (execute) if all its input places have sufficient tokens. Firing a transition removes tokens from input places and adds them to output places.  Tokens can be simple or complex, carrying additional information about the system state.  Multiple transitions can be enabled simultaneously, representing concurrent events in the system.  Arcs connecting multiple places to a transition enforce synchronization, requiring all those places to have tokens for the transition to fire. | | Examples | Widely used in embedded systems, cyber-physical systems. | |
| 3.2  (Solved in Rust instead of Scala and Ecore) | enum Color {  Black,  White,  Red,  }  impl Color {  fn from\_str(value: &str) -> Option<Color> {  match value {  "black" => Some(Color::Black),  "white" => Some(Color::White),  "red" => Some(Color::Red),  \_ => None,  }  }  }  struct Property {  name: String,  value: Color,  }  impl Property {  fn new(name: &str, value: &str) -> Result<Property, String> {  Color::from\_str(value)  .map(|color| Property { name: name.to\_string(), value: color })  .ok\_or\_else(|| format!("Invalid color value: {}", value))  }  }  struct StyleRule {  selector: String,  properties: Vec<Property>,  }  impl StyleRule {  fn new(selector: &str, properties: Vec<Property>) -> StyleRule {  StyleRule { selector: selector.to\_string(), properties }  }  }  struct StyleSheet {  rules: Vec<StyleRule>,  }  impl StyleSheet {  fn new(rules: Vec<StyleRule>) -> StyleSheet {  StyleSheet { rules }  }  } |
|  |  |