A MetaChecker for pyGK

Presented By Aaron Shui
Overview

• The Problem
• Solution Approach
• pyGK Intro
• Petri Net Example and Demo
Problem

• Does a model adhere to its metamodel?

• e.g. Is this a valid Petri Net?
Define (Meta) Model

• A model is a collection of model elements.
• Each element can have attributes.
• Each element can be connected to other elements (or itself).
Approach

• Function that checks a model against its metamodel.

• Has 3 Phases:
  – Phase I: Check types of elements.
  – Phase II: Check attributes of elements
  – Phase III: Check connections of elements

• Identifies major problems faster but runs slower than a single phase checker.
Phase I: Elements

• Ensure every element is an instance of something in the metamodel?
  – e.g. Does “MyClass” belong in a Petri Net?

• Checked automatically in the latest version of pyGK.
Phase I Check

- Implemented it anyways
- For each element check if its type exists as a meta element
  - e.g. Does the concept Class exist in the metamodel of Petri Nets?
Phase II: Attributes

• Use symbol table of attributed node in pyGK
  – Associate a predefined type with a name
• e.g. Number of tokens represented as an entry in symbol table of type Integer.
Phase II Checks

• Check:
  – Do all attributes have matching meta attributes (name and type)?
  – Are all uninstantiated meta attributes passed down as attributes?
  – Do any attribute override instantiated meta attributes?
Phase III: Connections

• Elements of a model are connected to other elements.
  – e.g. Place can be connected to a Place2Trans connector element.

• Are directed connections

• Note: these connections are not connectors in a model
  – e.g. Not: Association, Relationship, Trans2Place
Phase III Checks

• Are the meta elements of connected elements connected?

• e.g. If Place: p1 and Place2Trans: p2t1 are connected, are Place and Place2Trans connected?
pyGK

- Developed by Marc Provost
- Hi-Graph kernel for metamodeling
- Optimized for metamodeling
- Extended for graph rewriting/transformation system
Hi-Graph in pyGK
Defining (Meta)Models

- Text based description - pyGK has no GUI (yet)
- Build a foundational metamodel (ER) using Hi-Graph formalism supported by pyGK kernel.
- Build other models as instances of foundational metamodel.
Petri Net in ER (Graphical)
Petri Net in ER (pyGK)

\[
PN = \text{Graph(ID = "PetriNet“, typeld = "ER")}
\]
\[
PN.\text{addElement(SymbolTable(ID = "Place", typeld = "Entity", value = {}))}
\]
\[
PN.\text{addElement(SymbolTable(ID = "Transition", typeld = "Entity", value = {}))}
\]
\[
PN.\text{addElement(SymbolTable(ID="Place2Trans",typeld="Relationship",value={}))}
\]
\[
PN.\text{addElement(SymbolTable(ID="Trans2Place",typeld ="Relationship",value={}))}
\]
\[
PN.\text{getElement("Place")["Tokens"] = Int()}
\]
\[
PN.\text{getElement("Place2Trans")["Weight"] = Int()}
\]
\[
PN.\text{getElement("Trans2Place")["Weight"] = Int()}
\]
\[
PN.\text{connect("Place", "Place2Trans")}
\]
\[
PN.\text{connect("Place2Trans", "Transition")}
\]
\[
PN.\text{connect("Transition", "Trans2Place")}
\]
\[
PN.\text{connect("Trans2Place", "Place")}
\]
Petri Net Model in Petri Net

\[
\begin{align*}
\text{PNm1} & = \text{Graph(ID = "PNm1" , typeId = "PetriNet")} \\
p1 & = \text{SymbolTable(ID = "p1" , typeId = "Place" , value = {}}) \\
p2 & = \text{SymbolTable(ID = "p2" , typeId = "Place" , value = {}}) \\
t1 & = \text{SymbolTable(ID = "t1" , typeId = "Transition")} \\
p2t1 & = \text{SymbolTable(ID = "p2t1" , typeId = "Place2Trans" , value = {}}) \\
t2p2 & = \text{SymbolTable(ID = "t2p2" , typeId = "Trans2Place" , value = {}}) \\
p1["Tokens"] & = \text{Int(value = 2)} \\
p2["Tokens"] & = \text{Int(value = 0)} \\
p2t1["Weight"] & = \text{Int(value = 2)} \\
t2p2["Weight"] & = \text{Int(value = 2)} \\
\text{PNm1.addElement(p1)} & \ldots \ \text{PNm1.addElement(t2p2)} \\
\text{PNm1.connect("p1" , "p2t1")} & \ldots \ \text{PNm1.connect("t2p2" , "p2")}
\end{align*}
\]
 Faulty Petri Net

PNm1 = Graph(ID = "PNm1" , parent = root , typeld = "PetriNet")
p1 = SymbolTable(ID = "p1" , typeld = "Place" , value = {})
p2 = SymbolTable(ID = "p2" , typeld = "Place" , value = {})
t1 = SymbolTable(ID = "t1" , typeld = "Transition")
p2t1 = SymbolTable(ID = "p2t1" , typeld = "Place2Trans" , value = {})
t2p2 = SymbolTable(ID = "t2p2" , typeld = "Trans2Place" , value = {})
p1["Faulty Attribute"] = Int(value = 2)
p2["Tokens"] = Int(value = 0)
# p2t1["Weight"] = Int(value = 2)
t2p2["Weight"] = Float(value = 2)
PNm1.addElement(p1) ... PNm1.addElement(t2p2)
PNm1.connect("p1" , "p2t1") ... PNm1.connect("t2p2" , "p2")
PNm1.connect("p1", "p2")
Future Work

• Support more complex connections
  – e.g. Inheritance, Aggregation

• Difficulty: augment Graph or Interpreter?
  – Graph: flexible user defined checking
  – Interpreter: cleaner developer defined checking
Future Work (2)

- Extend attribute types
  - User defined attributes
  - Metamodel Elements (necessary?)
  - Indirectly supported via subclassing nodes
Questions?
References
