Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Formal Verification of Activity Diagrams Using the Gamma Framework

Scientific Students' Association Report

Author:

Ármin Zavada

Advisor:

dr. Vince Molnár

Contents

Kivonat

Jelen dokumentum egy diplomaterv sablon, amely formai keretet ad a BME Villamosmérnöki és Informatikai Karán végző hallgatók által elkészítendő szakdolgozatnak és diplomatervnek. A sablon használata opcionális. Ez a sablon LaTeX alapú, a *TeXLive* TeX-implementációval és a PDF-LaTeX fordítóval működőképes.

Abstract

This document is a LaTeX-based skeleton for BSc/MSc theses of students at the Electrical Engineering and Informatics Faculty, Budapest University of Technology and Economics. The usage of this skeleton is optional. It has been tested with the *TeXLive* TeX implementation, and it requires the PDF-LaTeX compiler.

Chapter 1

Introduction

Model-Based Systems Engineering is a widely used technique for designing, testing and executing otherwise too complicated systems in a well defined, *high-level* language (e.g. SysML). Such systems usually come from critical domains (e.g. transportation, flight) which have a very low tolerance for errors. As such, early detection of otherwise deadly mistakes is mandatory.

Formal model verification is a technique (among many) for detecting such errors. They use formal proving algorithms to check whether a given system property (usually an undesired state) is *reachable* and how to reach that state. However, these proving methods require low-level, *state based mathematical models*, which are far from the high-level languages we are used to. In order to verify such models, we need to *bridge* the gap between the two worlds.

Gamma is one of such frameworks. Right now Gamma only supports composite systems of only statecharts, however, real world models use activity diagrams heavily. For this reason, I propose to add an activity language to Gamma, and with it enable the systematic verification of a larger subset of SysML models.

TODO létező megoldások keresése és leírása (Beni pl)

TODO dolgozat felépítése

Chapter 2

Background

2.1 Model Verification

Kell ez egyáltalán? A cél az lett volna, hogy leírja a fontosságát, de ezt a bevezetőben megteszem..

2.2 Modeling Formalisms

2.2.1 Extended symbolic transition system

XSTS

2.2.2 Activities

SysML és általános activity leírás

2.2.3 Petri nets

Petri hálók leírása

2.3 The Gamma Statechart Composition Framework

The Gamma Statechart Composition Framework is a tool created to support platform-independent modeling and heterogeneous development of [system] components. For my purposes, it provides an interface to *design* composite systems and transform them to a verifiable model (XSTS).

Figure 2.1 shows an abstract representation of how a *high-level* SysML model could be verified using Gamma as a *middle man*. Important aspect of the transformations is to have back annotations, which provide the user with the information of *which* source model element the verifier returned.

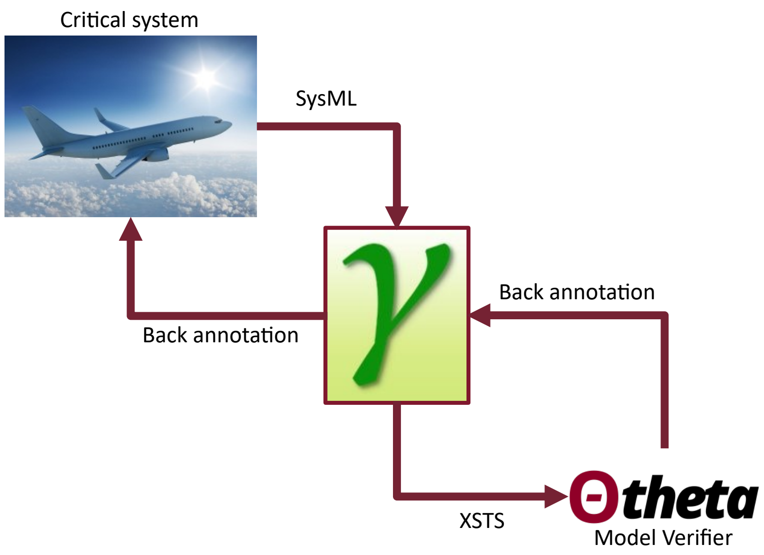

Figure 2.1: Using Gamma to verify high-level models

2.4 Related Work

Itt is jó, de kicsit a bevezetőnél is beszélek róla. Talán ott jobb lenne.

Chapter 3

Activity Language

3.1 Language Design

Language design usually determines the domain and usability of the end result. For this reason, I knew from the beginning, how important it is to lay down the priorities of the language.

The aim of the language is to sit in the middle of the SysML and XSTS semantics – provide a *bridge* in between the two . To achieve that, first, it needs to be **semantically correct** – at least according to our *activity semantics* – to lift the burden of filtering out incorrect models. Secondly, it needs to be easily transformed to XSTS and from SysML.

The latter can be done by having a metamodel close to SysML, and the former by minimising the complexity of the language. Needless to say, this is not an easy thing to do, and requires an incremental design process.

3.2 Metamodel

The metamodel of the language was constructed to facilitate well defined semantics, but have as few model elements as possible.

3.2.1 Semantic part

This part provides elements needed to construct a semantically correct model. An Activity model has **Nodes** and **Flows** in between nodes. Flows can be connected to nodes directly, or to **Pins**, which gives us explicit control over *control* flows and *data* flows. See figure **??**

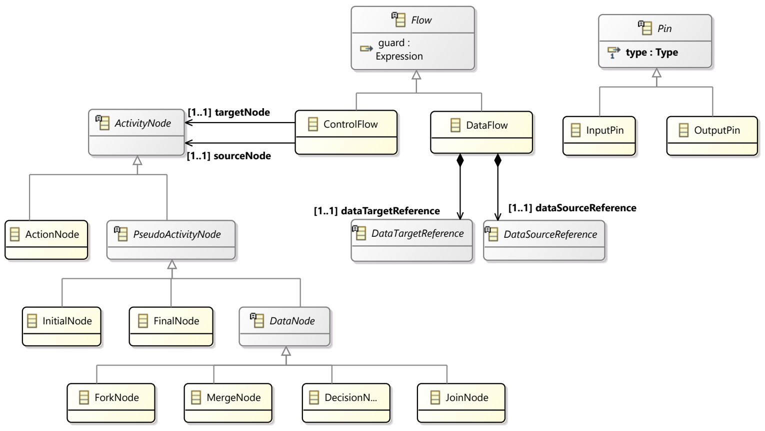

Figure 3.1: Using Gamma to verify high-level models

3.2.2 Syntactic part

The metamodel also has to provide elements for the language, such as **Declaration**s, **Definition**s, etc.

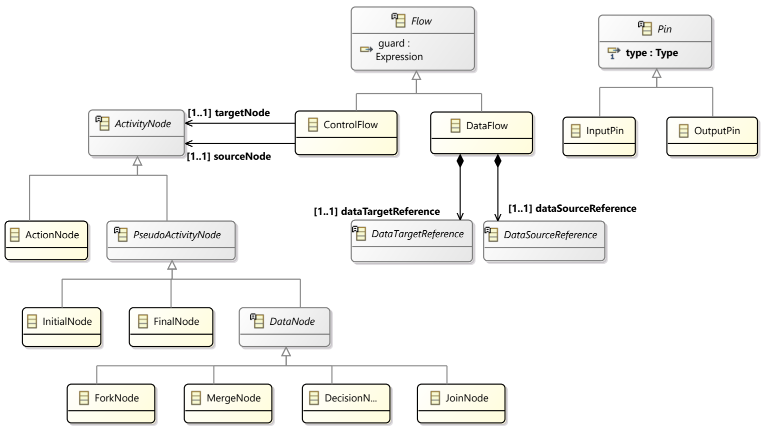

Figure 3.2: Using Gamma to verify high-level models

3.3 Domain-specific Language

In order to make it easier to test both the XSTS and SysML transformations, I created an **Xtext** domain-specific language. Easy readability and writing was not one of the main priorities, because the end goal is to have a higher-level systems modelling language as a source. As a result, many many constructs are inherently repetitive to write.

3.3.1 Language elements

The following section gives high level introduction into the syntax of the Gamma Activity DSL.

Inherited from Gamma

Some of the used constructs were reused from the other languages of the Gamma framework in order to provide tighter integration.

Lehet ez is rossz helyen van.. Végülis visszamehet a background részre

Pins

Pins can be declared the following way, where `<direction>` can be either `in` or `out`, and `type` a valid Gamma Expression type.

```
<direction> <name> : <type>
```

For example:

```
in examplePin : integer
```

where the direction is `in`, the name is `examplePin` and the type is `integer`.

Nodes

Nodes can be declared by stating the `type` of the node and then it's name. The type determines the underlying meta element. See figure 3.1

The available node types:

```
initial InitialNode
decision DecisionNode
merge MergeNode
fork ForkNode
join JoinNode
final FinalNode
action ActionNode
```

Flows

The behaviour of the Activity can be described by stating `data` or `control` flows between two nodes. Flows may have guards on them, which limits when the flow can fire. Activities may only be from the current activity definition's children. Pins can be accessed using the `.` accessor operator, the activity on the left hand side, and the pin's name on the right. The enclosing activity's name is `self`.

Flows can be declared the following way, where `<kind>` is the kind of flow, `<source>` and `<target>` is the source/target node or pin, and `<guard>` is a Gamma Expression returning boolean:

```
<kind> flow from <source> to <target> [<guard>]

control flow from activity1 to activity2
control flow from activity1 to activity2 [x == 10]
data flow from activity1.pin1 to activity2.pin2
data flow from self.pin to activity3.pin2
```

Declarations

Activity declarations state the *name* of the activity, as well as its *pins* 3.3.1.

```
activity Example (
  ..pins..
) {
  ..body..
}
```

You can also declare activities inline by using the `:` operator:

```
activity Example {
  action InlineActivityExample : activity
}
```

Definitions

Activities also have definitions, which give them bodies. The body language can be either `activity` or `action` depending on the language metadata set. Using the `action` language let's you use any Gamma Action expression, including timeout resetting, raising events through component ports, or simple arithmetic operations.

An example activity defined by an action body:

```
activity Example (
  in x : integer,
  out y : integer
) [language=action] {
  self.y := self.x * 2;
}
```

Inline activities may also have pins and be defined using action language:

```
activity Example {
  action InlineActivityExample : activity (
    in input : integer,
    out output : integer
  ) [language=action] {
    self.output := self.input;
  }
  }
```

3.3.2 Integration into Gamma

3.4 Examples

3.4.1 Adder Example

The following is a basic example of an adder activity. It ’reads’ two numbers, adds them, and then ’logs’ the result.

```
package hu.mit.bme.example

activity Adder(
  in x : integer,
  in y : integer,
  out o : integer
) [language=action]  {
  self.o := self.x + self.y;
}

activity Example {
  initial Initial

  action ReadSelf1 : activity (
    out x : integer
  )
  action ReadSelf2 : activity (
    out x : integer
  )
  action Add : Adder
  action Log : activity (
    in x : integer
  )

  final Final

  control flow from Initial to ReadSelf1
  control flow from Initial to ReadSelf2
  control flow from Initial to Add
  data flow from ReadSelf1.x to Add.x
  data flow from ReadSelf2.x to Add.y
  data flow from Add.o to Log.x
  control flow from Log to Final
}
```

Chapter 4

Activity Model Verification

4.1 Activities as State-based Models

4.2 Transforming Individual Elements

4.3 Equivalence with Petri Nets

Chapter 5

Implementation

TODO: rövid leírása az implementációnak; kb milyen méretű a változtatás. nem szeretném hosszúra, annyira nem érdekes, viszont jó lenne valahogy mutatni, hogy nem volt azért triviális.

Chapter 6

Evaluation

Chapter 7

Conclusion and Future Work

Acknowledgements

Ez nem kötelező, akár törölhető is. Ha a szerző szükségét érzi, itt lehet köszönetet nyilvánítani azoknak, akik hozzájárultak munkájukkal ahhoz, hogy a hallgató a szakdolgozatban vagy diplomamunkában leírt feladatokat sikeresen elvégezze. A konzulensnek való köszönetnyilvánítás sem kötelező, a konzulensnek hivatalosan is dolga, hogy a hallgatót konzultálja.

16

Bibliography