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

Formal modeling and verification of process models in component-based reactive systems

Scientific Students' Association Report

Author:

Ármin Zavada

Advisor:

dr. Vince Molnár
Bence Graics

Contents

Kivonat

A biztonságkritikus rendszerek komplexitása folyamatosan növekedett az elmúlt években. A komplexitás csökkentése érdekében a modellalapú paradigma vált a meghatározó módszerré ilyen rendszerek tervezéshez. Modellalapú rendszertervezés során a komponensek viselkedését általában állapotalapú, vagy folyamatorientált modellek segítségével írjuk le. Az előbbi formalizmusa azt írja le, hogy a komponens milyen állapotokban lehet, míg az utóbbié azt, hogy milyen lépéseket hajthat végre, valamint milyen sorrendben. Gyakran ezen modellek valamilyen kombinálása a legjobb módja egy komplex komponens viselkedésének leírásához.

Formális szemantikával rendelkező modellezési nyelvek lehetővé teszik a leírt viselkedés (kimerítő) verifikációját. Formális verifikáció használatával már a fejlesztés korai fázisaiban felfedezhetőek a hibák: a módszer ellenőrzi, hogy a rendszer egy adott (hibás) állapota elérhető-e, és amennyiben elérhető, ad hozzá egy elérési útvonalat. A formális verifikációs eszközök emiatt gyakran csak alacsony szintű, állapotalapú modelleken működnek, melyek messze vannak az emberek által könnyen érthető nyelvektől. Ezért, hogy magas szintű viselkedési modelleket tudjunk verifikálni, implementálnunk kell egy olyan modell transzformációt, mely megtartja a folyamat- és állapotalapú modellek szemantikáját azok kombinációja után is.

Ebben a dolgozatban megvizsgálom a folyamatalapú modellek szemantikáját, valamint a kapcsolatukat egyéb hagyományos állapotalapú modellekkel. Emellett megoldásokat vetek fel a potenciális konfliktusokra a kombinált alacsonyszintű modellben. Munkám során a Gamma állapotgép kompozíciós keretrendszerre építek, mellyel komponensalapú reaktív rendszereket modellezhetünk és verifikálhatunk. Mivel a Gamma még nem támogatja az aktivitásokat, bevezetek egy új aktivitás nyelvet, melyhez a SysMLv2 szolgál inspirációként. Ezzel együtt implementálom hozzá a szükséges transzformációkat a Gamma alacsony szintű analízis formalizmusára. Végezetül pedig kiértékelem a koncepcionális és gyakorlati eredményeket esettanulmányokon és méréseken keresztül, valamint felvetek lehetséges fejlesztéseket és alkalmazásokat.

Abstract

The complexity of safety-critical systems has been increasing rapidly in recent years. To mitigate said complexity, the model-based paradigm has become the decisive way to design such systems. In model-based systems engineering, we usually define the behaviour of system components using state-based or process-oriented models. The former formalism describes what states the component can be in, while the latter describes what steps it can perform and in what order. Oftentimes, the best way to model the behaviour of a complex component is to combine these models in some way.

Modelling languages with formal semantics enable the (exhaustive) verification of the described behaviour. Formal verification may be used to detect errors early during development by checking if a given (erroneous) state of the system can be reached, and if so, providing a way to reach it. Formal verification tools often require low-level state-based mathematical models, which are far from human-understandable languages. Thus, to enable the verification of high-level behavioural models, a model transformation must be implemented that preserves the semantics of both process-oriented and state-based models, even when combined.

In this report, I analyse the semantics of process-oriented models, as well as its relation to traditional state-based models, and propose solutions for the possible conflicts in a combined low-level model. In my work, I build on the Gamma Statechart Composition Framework, which is a tool for modelling and verifying component-based reactive systems based on statecharts. Since Gamma does not support activities yet, I introduce a new activity language inspired by SysMLv2, and implement the necessary transformations to Gamma's low-level analysis formalism. Finally, I evaluate the conceptual and practical results through case studies and measurements then propose potential improvements and applications.

Chapter 1

Introduction

The complexity of safety-critical systems has been increasing rapidly in recent years. To mitigate said complexity, the model-based paradigm has become the decisive way to design such systems. In model-based systems engineering, we usually define the behaviour of system components using state-based or process-oriented models. The former formalism describes what states the component can be in, while the latter describes what steps it can perform and in what order. Oftentimes, the best way to model the behaviour of a complex component is to combine these models in some way.

Modelling languages with formal semantics enable the (exhaustive) verification of the described behaviour. Formal verification may be used to detect errors early during development by checking if a given (erroneous) state of the system can be reached, and if so, providing a way to reach it. Formal verification tools often require low-level state-based mathematical models, which are far from human-understandable languages. Thus, to enable the verification of high-level behavioural models, a model transformation must be implemented that preserves the semantics of both process-oriented and state-based models, even when combined.

Gamma Statechart Composition Framework is a tool for bridging the gap between the two models. It is a tool for modelling and verifying component-based reactive systems based on statecharts. Since Gamma does not support activities yet, I introduce a new activity language inspired by SysMLv2, and implement the necessary transformations to Gamma's low-level analysis formalism.

ezt még át kéne írni

kicsit beszélni a létező implementációkró

Chapter 2

Background

In this chapter I address the foundations of this work. In **??** I introduce the theoretical bases of behaviour modeling, which is used widely in systems engineering, and their implementation in an existing language - SysML. After this I talk about Formal Verification in Section 2.3, which is a mathematically rigorous method for finding specific configurations of modelled systems. Lastly, I introduce the Gamma Composite Statechart Framework, which is a tool for stating and verifying composite models using statechart behaviours.

2.1 Behaviour Models

2.1.1 Finite State Machine

Finite state machines (FSMs) are widely used formalisms to model computation. They can be in exactly one of a finite number of *states* at any given time. FSM can change from one state to another in response to some *inputs* (usually called signals); which change is called a *transition* (see Figure 2.1 for an example finite state machine).

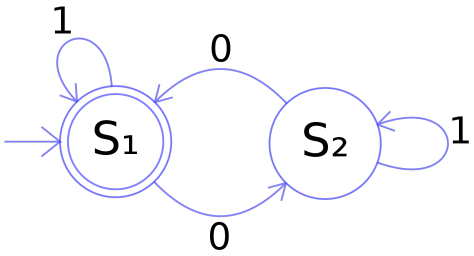

Figure 2.1: An example deterministic finite state machine, which checks a given binary numbers *evenness*.

The formal definition of *deterministic* finite state machines is as follows.

Definition 1 (Deterministic finite state machine). A DFSM is a tuple $SM = (\Sigma, S, s_0, \delta, F)$

- Σ is the set input *alphabet* (a finite non-empty set of symbols);
- S is a finite non-empty set of states;

- $s_0 \in S$ is an initial state;
- $\delta : S \times \Sigma \to S$ is the state-transition function;
- $F \subseteq S$ is the set of final states .

The current state of a DFSM is $s \in S$. A transition from state to state is performed, given the last input letter l, if and only if $s' = \delta(s, l) \in S$. In which case the current state will become s' (this work assumes, that if the given $\delta(s, l)$ is not defined, then the state machine remains in it's current state).

2.1.2 Petri Net

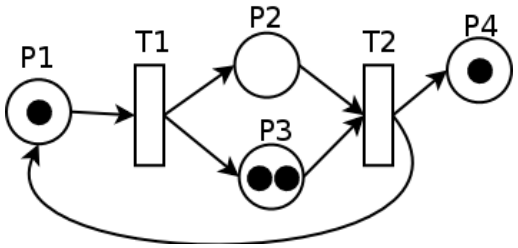

Figure 2.2: An example Petri net with 2 transitions and 4 places.

Petri nets are a widely used formalism to model concurrent, asynchronous systems [2]. The formal definition of a Petri net (including inhibitor arcs) is as follows (see Figure 2.2 for an illustration of the notations).

Definition 2 (Petri net). A Petri net is a tuple $PN = (P, T, W, M_0)$

- P is the set of *places* (defining state variables);
- T is the set of *transitions* (defining behaviour), such that $P \bigcap T = \emptyset$;
- $W \subseteq W^+ \bigcup W^-$ is a set of two types of arcs, where $W^+ : T \times P \to \mathbb{N}$ and $W^- : P \times T \to \mathbb{N}$ are the set of input arcs and output arcs, respectively ($\mathbb{N}$ is the set of all natural numbers);
- $M_0 : P \to \mathbb{N}$ is the *initial marking*, i.e., the number of *tokens* on each place. .

The state of a Petri net is defined by the current marking $M : P \to \mathbb{N}$. The behaviour of the systems is described as follows. A transition t is enabled if $\forall p \in P : M(p) \in W(p, t)$. Any enabled transition t may fire non-deterministically, creating the new marking M' of the Petri as follows: $\forall p \in P : M'(p) = M(p) - W^-(p, t) + W^+(t, p)$.

In words: W describes the *weight* of each flow from a transition to a place, or from a place to a transition. Firing a transition t in a marking M consumes $W^-(p_i, t)$ tokens from each of its input places p_i, and produces $W(t, p_o)$ tokens in each of its output places p_o. One such transition t is *enabled* (it may *fire*) in M if there are enough tokens in its input places for the consumptions to be possible, i.e., if and only if $\forall p : M(p) \geq W(s, t)$.

2.1.3 Differences

In real world applications, both behaviour models are useful, but for different use-cases. State machines provide a way of specifying what our system *react* with, when a given *environmental* event happens. This could be a user interaction (e.g., a keystroke) or a new temperature reading.

On the other hand, petri nets provide a way of modeling distributed systems with many interconnecting components, all running on their own accords. They have dependencies on each other (one calculates a value the other needs), or have a limited resource (a factory only has one worker).

2.2 SysML

SysML is a widely known and used Systems Modeling Language. Its aim is to provide a generic, extendable modeling language, capable of describing the most complicated systems from the specifications, all the way down to the actual behaviours of specific modules.

The previously introduced formal behaviour models have the power to describe many different behaviours a component may have, and are easy to reason about. However, such low-level models are not easy for users to write and maintain; that is why languages such as SysML exist - they provide high-level abstractions over low-level formal models.

In the following section I will present how SysML implements the two behaviour models introduced in Section 2.1.

2.2.1 State Machine

State Machines are a high-level abstraction over deterministic finite state machines. Please note, that the exact semantics of SysML State Machines are not needed for this work, I only introduce the concept briefly for the possibly use-cases it has. A given state machine contains many *states*, and *transitions* between said states. Transitions use *triggers* to enable them, and can also have an *effect* on the component itself. Such an effect is called the *action* of the transition. Figure 2.3 is a simple example.

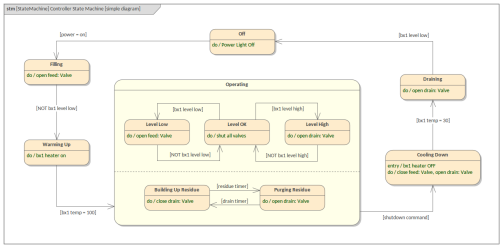

Figure 2.3: A SysML State Machine.

State Machines can also have *composite* states and regions, which help with simplifying the behaviour model.

Along with actions defined on transitions, states may also have multiple actions associated with them. *entry actions* defines a set of actions to be performed *before* the state becomes the current state, *exit actions* is a set of actions to be performed *after* the state is no longer the current state. On a similar logic, *do actions* is a set of actions to perform *while* the state machine is in the given state. This fact will be address more in depth in **??**.

2.2.2 Activity

Activities are a high-level abstraction over Petri nets.

– simple activity diagram –

Elements of SysML Activities

– metamodel of activities –

Action The specific operation an Action describes can be stated in various ways, usually in JavaScript or plain old English (Opaque action). In JavaScript, the engineer may access and change variables inside the containing object, send signals to connected objects, or just call existing functions.

– example of action with js –

There are many different kinds of Activity Nodes, which describe different behaviour for guiding tokens through them.

SysML Activities as Petri Nets

In this section, I will introduce the semantics of Activities using Petri nets as formal semantics. Note however, since not all elements in SysML activity diagrams have execution semantics, I will only consider a set of the modeling elements, including basic actions, initial nodes, final nodes, join nodes, fork nodes, merge nodes, decision nodes, pins and object/control flows.[3][1]

2.2.3 Combining Behaviours

In SysML the behaviours can be combined using the Call Behaviour sintax, which means the semantics here is combined with the called behaviour's semantics. For example, a state machine's state may have a do action, in which a Call Behaviour action is specified.

2.3 Formal Verification

Formal verification is a method for proving or disproving the correctness of a system with mathematical precision. Correctness is checked with respect to certain properties or specifications given by the user. Model checking is a formal verification technique that explores the behaviour of the given model exhaustively, i.e., all relevant behaviours of the model are analysed.

Formal verification tools require the formal model and a formal requirement as input, and return either an ok state, a counter example, or that it can't decide (see Figure 2.4). The counter example is the most important of all, as with its help, the engineers may have a chance of correcting the model.

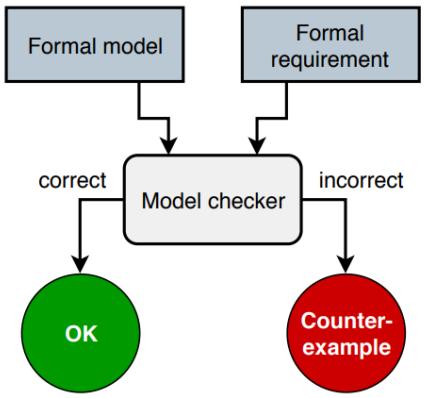

Figure 2.4: An illustration of model checking.

2.4 Extended Symbolic Transition System

The high-level nature of engineering models means they are easy-to-use for engineers, but leads to difficulties during the formal verification process. In case of statecharts for example, orthogonal regions, hierarchical state-refinement or broadcast communication are all high-level constructs that make the modeling workflow more intuitive and enable the modeling of significantly more complex systems. They are however difficult to process using formal methods that are defined on low-level mathematical formalism and verified using SMT solvers. In this section, I introduce the XSTS language, which is a low-level modeling formalism designed to bridge the aforementioned gap between engineering models and formal methods.

2.4.1 Formal definition

Definition 3 (Extended symbolic transition system). An *Extended symbolic transition system* is a tuple $XSTS = (D, V, V_C, IV, Tr, In, En)$, where:

- $D = D_{v1}, D_{v2}, \ldots, D_{vn}$ is a set of value domains;
- $V = v_1, v_2, \ldots, v_n$ is a set of variables with domains $D_{v1}, D_{v2}, \ldots, D_{vn}$;
- $V_C \subseteq V$ is a set of variables marked as *control variables*;
- $IV \in D_{v1} \times D_{v2} \times \cdots \times D_{vn}$ is the *initial value function* used to describe the initial state. The initial value function IV assigns an initial value $IV(v) \in D_v$ to variables $v \in V$ of their domain D_v;
- $Tr \subseteq Ops$ is a set of operations, representing the *internal transition relation*; it describes the internal behaviour of the system;
- $In \subseteq Ops$ is a set of operations, representing the *initialisation transition relation*; it is used to describe more complex initialisation, and is executed once and only once, at the very beginning;
- $En \subseteq Ops$ is a set of operations, representing the *environmental transition relation*; it is used to model the system's interactions with its environment. ▪

In any state of the system a single operation is executed, which is selected from the sets introduced above. The set from where the operation can be selected depends on the current state. In the initial state - (which is described by the initialization vector IV) - only operations from the In set can be executed. Operations from the In set can only fire in the initial state and nowhere else. After that, En and Tr can be fired in an alternating manner.

Operations $op \in Ops$ describe the transitions between states of the system, where Ops is the set of all possible transitions. All operations are atomic in the sense that they are either executed in their entirety or none at all.

2.4.2 Language constructs

In the following section, I will introduce the language constructs and their semantics of XSTS language.

2.4.2.1 Types

XSTS contains two default variable types, logical variables (*boolean*) and mathematical integers (*integer*). Types defined this way make up the domains $D_{v1}, D_{v2}, \ldots, D_{vn}$ introduced in Extended symbolic transition system 2.4.1. XSTS also allows the user to define *custom types*, similarly to enum types in common programming languages.

A custom type can be declared the following way:

```
type <name> : { <literal_1>, . . . , <literal_n> }
```

Where $D_{name} = \{literal_1, \ldots, literal_n\}$.

2.4.2.2 Variables

Variables can be declared the following way, where `<value>` denotes the value that will be assigned to the variable in the initialization vector:

```
var <name> : <type> = <value>
```

Where v_{name} will be in domain D_{type}.

If the user wishes to declare a variable without an initial value, this is possible as well:

```
var <name> : <type>
```

A variable can be tagged as a control variable with the keyword `ctrl`:

```
ctrl var <name> : <type>
```

In which case the variable v will also be added to V_C (the set of control variables).

2.4.2.3 Basic operations

Operations make up the set *Ops* introduced in Extended symbolic transition system 2.4.1. These operations, and their compositions define the behaviour of the XSTS model.

Assume An assumption operation can only be executed, if and only if its *expression* evaluates to *true*. This fact means, that if a composite operation contains a *falsy* assumption, the whole composite operation will not fire.

A simple assumption operation can be stated like the following:

```
assume <expr>
```

Assignment Assignments have the following syntax, where `<varname>` is the name of a variable and `<expr>` is an expression of the same type:

```
<varname> := <expr>
```

An assignment operation overwrites the variables value upon execution.

Havoc Havocs give the XSTS models randomness, by randomly *assigning* a value.

The syntax of havocs is the following, where `<varname>` is the name of a variable:

```
havoc <varname>
```

2.4.2.4 Composite operations

Composite operations give way to building up more complicated transition trees, by providing nesting the already introduced simple operations.

Choice Non-deterministic choices work by randomly executing one and only one of its composed operations.

Non-deterministic choices have the following syntax, where `<operation>` are arbitrary basic or composite operations:

```
choice { <operation> } or { <operation> }
```

Sequence Sequences execute all composed operations one-by-one from top to bottom.

Sequences have the following syntax:

```
<operation>
<operation>
<operation>
```

2.4.2.5 Transitions

Each transition is a single operation (basic or composite). We distinguish between three sets of transitions, *Tran*, *Init* and *Env* - associating to the three different operation sets introduced in Extended symbolic transition system 2.4.1. Transitions are described with the following syntax, where `<transition-set>` is either `tran`, `env` or `init`:

```
<transition-set> {
    <operation>
} or {
    <operation>
} or
...
or {
    <operation>
}
```

2.4.2.6 Simple example

Below is a simple example given in the textual representation of the XSTS DSL. The XSTS has two variables, x and y, both with the inital value 0. The init transition sets both variables' values to 1. After this, the environment repeatedly increments the value of y, to which the systems reacts by either incrementing x, or leaving the value of x unchanged.

```
var x: integer = 0
var y: integer = 0
tran {
    x:=x+1
} or {
    x:=x
}
init {
    x:=1
    y:=1
}
env {
    y:=y+1
}
```

2.5 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).

részletesebben, komponensek felírása, mert később még bele fogunk nyúlni

2.5.1 Statechart

2.5.2 Activity

Chapter 3

Gamma Activity Language

3.1 Language Design

The Gamma Activity Language aims at providing a medium in which one can describe Activities like in SysML, but also be semantically consistent.

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.

magasszintű leírása a nyelvnek, egy-egy példa, elképzelés. nem mit, hanem miért - de fontos, hogy mi alapján

tervezés menete, alfejezetekkel

one-one alapján simán lehetne

szemantikai utánanézés, scope, limitációk bevállalása

3.2 Formal modelling

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

kövér helyett texttt vagy textsc. Tök jó lenne, ha lennének példák a metamodel használatára, pl egy activity diagram - metamodel pár, ahol a kettő között húzódnának traceability csíkok

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.

fekete-fehér, kevés szín

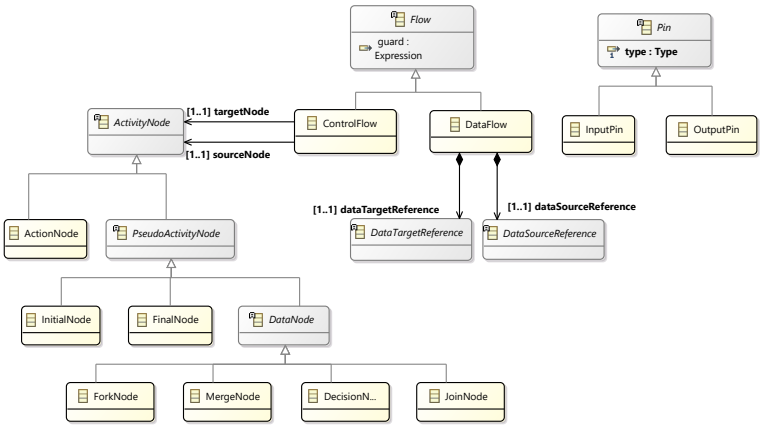

Figure 3.1: Using Gamma to verify high-level models

3.2.2 Syntactic part

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

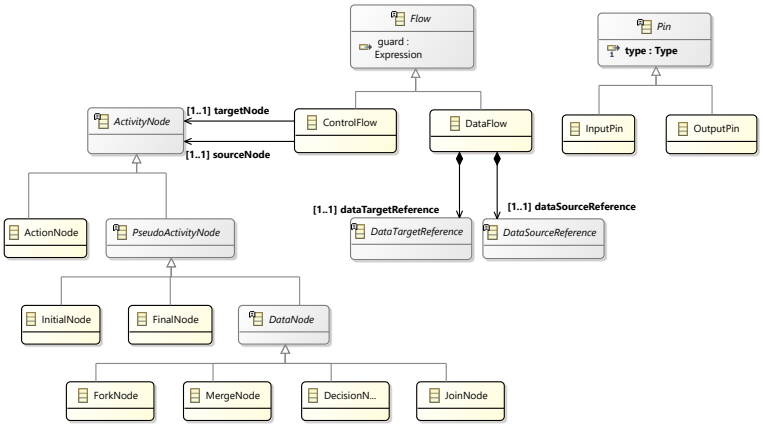

Figure 3.2: Using Gamma to verify high-level models

3.3 Domain-specific Language

ide még jöhet később xtext kód

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.

diagram, opaque action

Ehelyett egyébként lehetne a running example, amit még az egész elején definiálunk

```
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

activity vagy proces-model?

ahogy fentebb írtam, tokenek haladnak, és tranzíciók vannak.

A process-orionted model előnye itt a hátrányunk; mivel sok process mehet teljesen függetlenül egymástól, ezért nagyon nehéz szépen leírni őket

tegyük fel, hogy a node-ok és csatlakozók token-tartalma egy állapot; vagy van benne, vagy nincs. ehhez még hozzátesszük azt, hogy a node éppen fut, kész, vagy nem csinál semmit.

ezek alapján le tudunk írni bármilyen process-oriontált modelt állapot alapú modellben, feltételezve, hogy bármikor bármelyik tranzíció tüzelhet.

itt leírom az activity különböző node-jait, és azoknak a különböző szabályait (merge, choice, stb)

majd leírom a különböző csatlakozók segítségével is

4.2 Activities Alongside Statecharts

fentebb leírtam, hogyan lehet leképezni proces modelleket, de így csak magukban futhatnak. Itt most leírom, hogy milyen nehézségeket okozhat, amikor egymás mellett futtatjuk őket:

do activity:

amikor a state aktív, futhat az aktivity, de ez pár problémával jön: párhuzamossági és szinkornizációs problémák

action:

egy activity értékét ki kell lapítani, hogy a tranzíció tüzelése után azonnal értelmezhető legyen (nem felezhetjük el az adott tranzíciót, mert akkor érvénytelen állapotaink lesznek)

unblock:

még nagy kérdés, hogy milyen szinten bontsuk fel a lépéseket, mennyire legyenek atomikusak, mik futhatnak egymás mellett, stb

alap elképzelésben akár egy activity akár egy állapotgép definiálhat egy komponsens viselkedését, de jelenleg ezt is egyszerűsített módon implementáltuk; az activity az állapotgép része lehet

megoldás a kérdésekre:

megoldásként a legegyszerűbb módszert választottuk, mert első sorban az a kérdés, hogy ez a módszer egyáltalán alkalmazható-e; a másik mindenképpen bonyolítja az implementációt és a létrejövő modellt is

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

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.

Bibliography

[1] Edward Huang, Leon F. McGinnis, and Steven W. Mitchell. Verifying sysml activity diagrams using formal transformation to petri nets. *Systems Engineering*, 23(1):118–135, 2020. DOI: https://doi.org/10.1002/sys.21524. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/sys.21524.

[2] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989. DOI: 10.1109/5.24143.

[3] OMG. Semantics of a foundational subset for executable uml models. 2018. URL https://www.omg.org/spec/FUML/1.4/.