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

Add intro-duction

Chapter 2

Background

In this chapter I address the foundations of this work. In Section 2.1 I introduce the concept of model-based systems engineering, which is a well known approach for complex system design. This includes SysML as well (Section 2.1.1). After this I talk about Formal Verification in Section 2.2, and introduce Petri nets (Section 2.2.2 and a mapping between activities and Petri nets (Section 2.2.3). Lastly, I introduce the Gamma Composite Statechart Framework, which is a tool for stating and verifying composite models using statechart behaviours Section 2.3, and a low-level formalism created for model checking used by Gamma in Section 2.4.

2.1 Model-based Systems Engineering

The INCOSE SE Vision 2020 [1] defines Model-based systems engineering (MBSE) as:

> the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases. MBSE is part of a long-term trend toward model-centric approaches adopted by other engineering disciplines, including mechanical, electrical and software. In particular, MBSE is expected to replace the document-centric approach that has been practiced by systems engineers in the past and to influence the future practice of systems engineering by being fully integrated into the definition of systems engineering processes.

Applying MBSE is expected to provide significant benefits over the document centric approach by enhancing productivity and quality, reducing risk, and providing improved communications among the system development team [2].

In MBSE, one of the most important concepts is the term "model" itself. Literature gives various definitions for models:

1. A physical, mathematical, or otherwise logical representation of a system, entity, phenomenon, or process.[5]
2. A representation of one or more concepts that may be realized in the physical world [7].

3. A simplified representation of a system at some particular point in time or space intended to promote understanding of the real system [4].
4. An abstraction of a system, aimed at understanding, communicating, explaining, or designing aspects of interest of that system [6].
5. A selective representation of some system whose form and content are chosen based on a specific set of concerns. The model is related to the system by an explicit or implicit mapping [10].

As one can see, choosing a definition is very much dependent upon how we wish to use our "model"; in this work I will use the 1st definition.

2.1.1 Systems Modeling Language

Systems Modeling Language (OMG SysML [9]) is a general-purpose modeling language that supports the specification, design, analysis, and verification of systems that may include harware and equipment, software, data, personnel, procedures, and facilities. SysML is a graphical modeling language with a semantic foundation for representing requirements, behaviour, structure, and properties of the system and its components [7].

This work focuses only on the *behavioural* modeling tools SysML provides. In the following section, I present two of the most used concepts; State Machines and Activity Diagrams.

2.1.1.1 State Machine

Reactive systems are all around us in our daily lives; in smart phones, avionics systems or event our calculators. Frequently, reactive systems appear in areas, where safety-critical operation is crucial, as even the slightest misbehaviour can have catastrophic consequences. This makes the verification of these systems a must during their design process.

The defining characteristic of reactive systems is their event-driven nature, which means that they continuously receive external stimuli (events), based on which they change their internal state and possibly react with some output[13]. Reactive systems can be verified using model checking techniques (see Section 2.2). Statecharts [14] are a popular and intuitive language to capture the behaviour of reactive systems [16, 24], and are at the same time formal and rigorous.

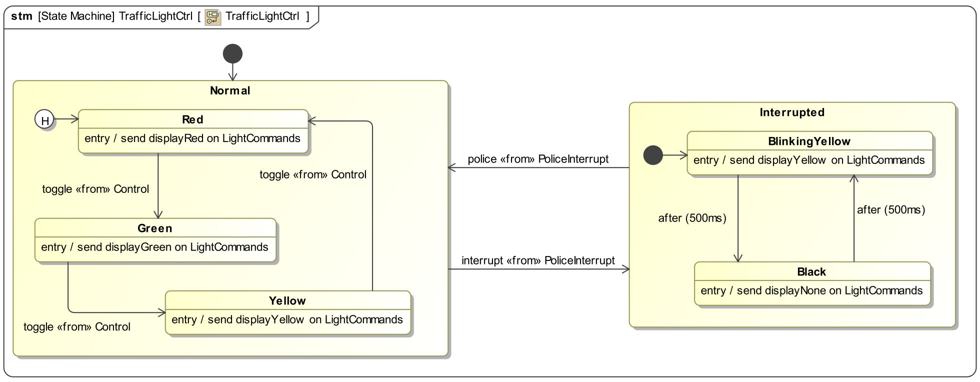

Figure 2.1: SysML State Machine describing the behaviour of a traffic light controller.

SysML state machines extend the concept of statecharts with hierarchical state-refinement, orthogonal regions, action-effect behaviour and state machine composition. These advanced constructs make state machines easy to use for engineers, but lead to the formal verification process being challenging. This challenge can be overcome using a transformation tool, such as Gamma, of which I will be talking in Section 2.3.

Figure 2.1 shows a state machine modeling the behaviour of a traffic light controller.

explain controller behaviour in words

2.1.1.2 Activity Diagram

As stated above, reactive systems are used frequently in the industry, however, they cannot describe the complicated semantics of distributed systems with concurrent, parallel behaviour, where the *interesting* thing is what the system *does* step-by-step. SysML activity diagrams are a primary representation for modeling process based behaviour [9] for distributed, concurrent systems. Figure 2.2 shows the set of interesting artifacts used in this work. In the following, I will introduce the different artifacts and show an example of a SysML activity diagram.

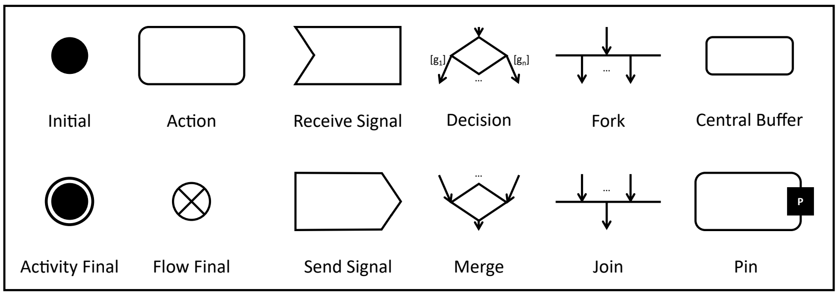

Figure 2.2: Artifacts of SysML activity diagrams.

SysML activity diagram is a graph based model, where the nodes are connected via flows. The dynamic behaviour of activity diagrams comes from *tokens* travelling from node to node; based on the given node's semantics a connected flow removes tokens from the source node and puts them onto the target node. Flows can also have *guards*, which are expressions specifying when the given flow is *enabled* or not; only enabled nodes can transfer tokens.

Tokens are a way of creating limitation over which node can do work, and which cannot; a given node is only considered *running*, when it contains a token. Tokens *flow* between nodes, carrying with them a given value - this value can be of type *void*, which makes it a *control* token.

The different nodes represent the different semantic 'tools' at our disposal; they can represent different actions, or introduce interesting token flow semantics. Simple **actions** represent a single step of behaviour that converts a set of inputs to a set of outputs. Both inputs and outputs are specified as pins, which get their data from connected flows - making that flow a *data flow*. The flow is started from the initial node, and is ended with a *Flow Final* or *Activity Final* node. *Fork* nodes generate tokens on all of their output flows, and *Join* nodes only forward the tokens, when all input flows contain one - thus making the kinds a *pair*. On the other hand, *Merge* nodes do not wait for all flows, they forward any token they receive instantly. Likewise, *Decision* nodes take one token from its input flows, and sends it out on its one and only one enabled output flows.

The detailed specification for SysML Activity Diagrams can be found in the OMG specification[9].

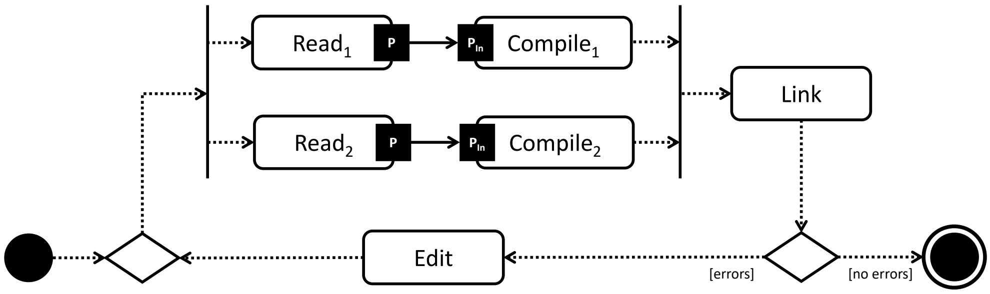

Figure 2.3: The activity of editing, compiling and linking two files.

Figure 2.3 shows an example activity diagram, modeling the process of editing, compiling and linking two files. First, the files have to be read, after which they are *transported* to the compiler module. We want to compile the two different files at the same time, thus we split the control flow using a *fork* node. Once the files are compiled, we link both of them - since linking requires both files, it is preceded by a *join* node. Finally, if the resulting code contains errors, we *edit* them and start over, otherwise we are done.

2.2 Formal Verification

In order to raise the reliability of system analysis, a system analysis technique is required that can have the precision of paper-and-pencil based mathematical proofs, and thus does not rely upon computer-arithmetic, and utilizes the computers for bookkeeping, to be able to handle complex systems without having to worry about human-errors. Formal verification methods, which are primarily based on theoretical computer science fundamentals like logic calculi, automata theory and strongly type systems, fulfil these requirements. The main principle behind formal analysis of a system is to construct a computer based mathematical model of the given system and formally verify, within a computer, that this model meets rigorous specifications of intended behaviour. Due to the mathematical nature of the analysis, 100% accuracy can be guaranteed [15].

2.2.1 Model Checking

Model Checking is a formal verification method to verify properties of finite systems, i.e., to decide whether a given formal model M satisfies a given requirement γ or not. The name comes from formal logic, where a logical formula may have zero or more models, which define the interpretation of the symbols used in the formula and the base set such that the formula is true. In this sense, the question is whether the formal model is indeed a model of the formal requirement: $M \not\models \gamma$?

Model Checker algorithms (see Figure 2.4), such as UPPAAL[1] [3] or Theta[2] [25] can answer this question, and can even return a *proof* (i.e., a part of the model) that M indeed does satisfy said requirement[3].

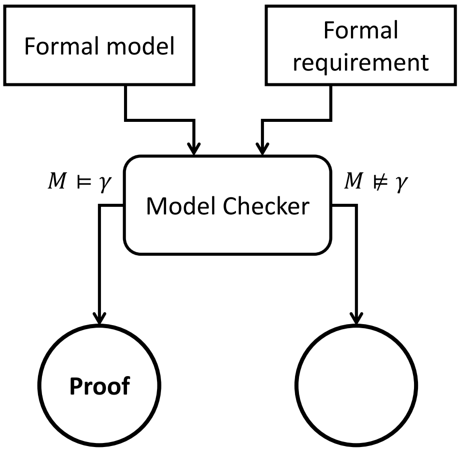

Figure 2.4: An illustration of model checking.

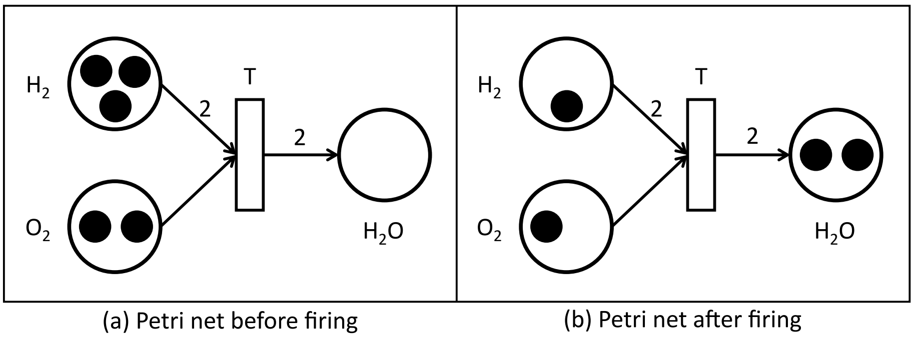

Figure 2.5: An example Petri net modeling the process of H_2O molecule creation.

2.2.2 Petri Net

Petri nets are a widely used formalism to model concurrent, asynchronous systems [22]. The formal definition of a Petri net (including inhibitor arcs) is as follows (see Figure 2.5 for an illustration of the notations).

Definition 1 (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: $\forall p \in P : M'(p) = M(p) - W^-(p, t) + W^+(t, p)$.

[1]https://uppaal.org/
[2]https://inf.mit.bme.hu/en/theta
[3]These proofs usually come in the form of an execution trace

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 $\forall p : M(p) \geq W(s, t)$.

2.2.3 Activities as Petri Nets

Formal verification requires models to be specified using *mathematical* precision, however SysML does not have precise semantics [21, 23, 18]. The paper called "Verifying SysML activity diagrams using formal transformation to Petri nets" [17] by Edward Huang, Leon F. McGinnis and Steven W. Mitchell proposes a way to *partially* map SysML activity diagrams to Petri nets. In the following, I will summarise their work.

2.2.3.1 Constrained Sub-set of SysML

Since SysML Activity Diagrams do not have exact execution semantics, the PN mapping can only be done for a limited subset of the modeling elements: *actions*, *initial nodes*, *final nodes*, *join nodes*, *fork nodes*, *merge nodes*, *decision nodes*, *pins* and *object/control flows*, which have precise execution semantics as defined in the Foundational Subset for Executable UML Models [11].

The paper also assumes the following constrains:

1. The value of tokens is not considered.
2. Control flows with multiple tokens at a time are not considered.
3. Optional object/control flows are not considered, ie, multiplicity lower bounds are strictly positive.

These constraints allow the mapping between activities and Petri nets, however, the constructed PN will not be semantically equivalent - data cannot flow between nodes. This fact is the motivation behind formalising a more-complete mapping (see Chapter 3).

2.2.3.2 Mapping Rules

Activity elements can be grouped into two sets: *load-and-send* (LAS) and *immediate-repeat* (IR).

LAS nodes are fired when all their inputs have tokens. When an LAS node fires, the number of tokens associated with the input arcs/pin is consumed and the number of tokens associated with an output arc/pin is added. In SysML activity diagrams, the execution semantics of fork nodes, join nodes, and basic actions are also LAS because these nodes are fired when all their input nodes have at least one token. As a result, these nodes can be mapped to *transitions* in the resulting Petri net.

In contrast, as soon as an IR node receives a token from any input, it immediately adds a token to its output nodes. For UML/SysML activity diagrams, activity final nodes, merge nodes, decision nodes and pins are IR nodes because they are fired immediately when any token is received. As a result, these nodes can be mapped to *places* in the resulting Petri net.

Given the set of assumptions in Section 2.2.3.1, control flows and object flows in an activity diagram can be mapped to arcs in a Petri net.

Finally, after mapping the elements, the resulting Petri net may contain transition-transition and place-place arcs, which are not valid; as final step, these arcs have to be split in two by inserting a transition or a place in the middle, making the model conform to the formalism.

2.2.3.3 Example Mapping

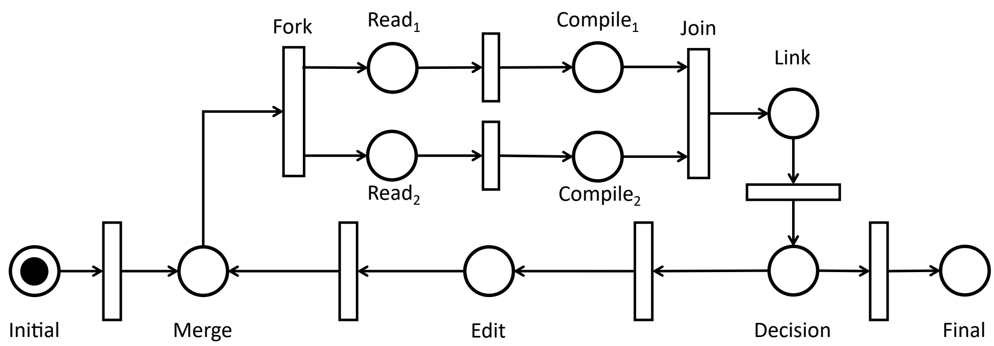

Figure 2.6: Example mapping from activity diagram to Petri net.

Figure 2.6 shows an example mapping from the activity Figure 2.3. The resulting elements are annotated with the names of their counter parts in the activity diagram.

For more information, please refer to the Paper!

2.3 The Gamma Statechart Composition Framework

The Gamma Statechart Composition Framework is an integrated tool to support the design, verification and validation as well as code generation for component-based reactive systems. The behaviour of each component is captured by a statechart, while assembling the system from components is driven by a domain-specific composition language. Gamma supports formal verification by mapping composite statecharts to a back-end model checker. Execution traces obtained as witnesses during verification are back-annotated as test cases to replay an error trace or to validate external code generators [19].

The workflow of Gamma builds on a model transformation chain depicted in Figure 2.7, which illustrates the input and output models of these model transformations as well as the languages in which they are defined, and the relations between them. The modeling languages are as follows.

- The **Gamma Statechart Language (GSL)** is a UML/SysML-based statechart language supporting different semantic variants of statecharts.
- The **Gamma Composition Language (GCL)** is a composition language for the formal hierarchical composition of state-based components according to multiple execution and interaction semantics.
- The **Gamma Genmodel Language (GGL)** is a configuration language for configuring model transformations.

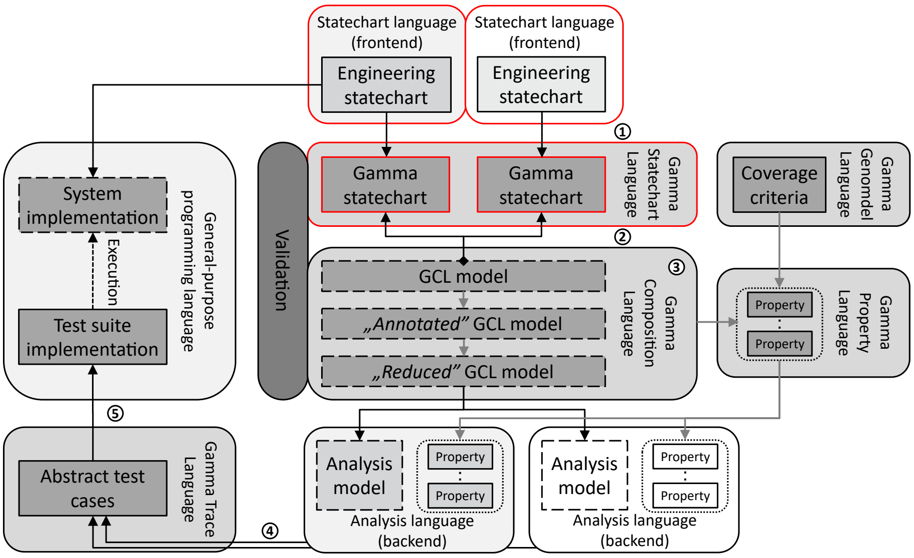

Figure 2.7: Overview model transformation chains and modeling languages of the Gamma framework. The parts relevant to this work have been marked with red outline.

- The **Gamma Property Language (GPL)** is a property language supporting the definition properties and thus, the formal specification of requirements regarding (composite) component behavior.
- The **Gamma Trace Language (GTL)** is a high-level specification language for execution traces of (composite) components.

Optionally, statechart models defined in supported modeling tools (front-ends) can be imported into Gamma (Step 0), which can be integrated according to well-defined execution and interaction semantics (Step 1-2). The resulting composite model is processed and transformed into the input formalisms of integrated model checker back-ends (Step 3). The model checker back-ends provide witnesses (diagnostic traces) based on specified properties, which are back-annotated, resulting in abstract traces (Step 4). Finally, the abstract traces are mapped into concrete (executable) traces tailored to the targeted execution environment (Step 5). Please note, that this is a summary of the workflow; for more detailed descriptions, see [8].

2.3.1 Example Statechart

Here you can see the Gamma Statechart representation of the State Machine introduced in Figure 2.1.

```
package TrafficLightCtrl
import "Interfaces"
statechart TrafficLightCtrl [
  port Control : requires Control
  port PoliceInterrupt : requires PoliceInterrupt
  port LightCommands : provides LightCommands
] {
  timeout BlinkingYellowTimeout3
  timeout BlackTimeout4
  transition from Yellow to Red when Control.toggle
  transition from Normal to Interrupted when PoliceInterrupt.police
  transition from Entry2 to Red
  transition from Entry1 to BlinkingYellow
```

```
  transition from Entry0 to Normal
  transition from Interrupted to Normal when PoliceInterrupt.police
  transition from BlinkingYellow to Black when timeout BlinkingYellowTimeout3
  transition from Black to BlinkingYellow when timeout BlackTimeout4
  transition from Green to Yellow when Control.toggle
  transition from Red to Green when Control.toggle
  region main_region {
    state Normal {
      region normal {
        shallow history Entry2
        state Green {
          entry / raise LightCommands.displayGreen;
        }
        state Red {
          entry / raise LightCommands.displayRed;
        }
        state Yellow {
          entry / raise LightCommands.displayYellow;
        }
      }
    }
    state Interrupted {
      region interrupted {
        initial Entry1
        state Black {
          entry / set BlackTimeout4 := 500 ms; raise LightCommands.displayNone;
        }
        state BlinkingYellow {
          entry / set BlinkingYellowTimeout3 := 500 ms; raise LightCommands.displayYellow;
        }
      }
    }
    initial Entry0
  }
}
```

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. SysML state machines and acitivty diagrams for example contain high-level constructs that make the modeling workflow more intuitive and enable the modeling of significantly more complex systems, however they are 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 [20] 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 2 (Extended symbolic transition system). An *Extended symbolic transition system* is a tuple $XSTS = (D, V, V_C, IV, Tr, In, En)$, where:

- $D = \{D_{v_1}, D_{v_2}, \dots, D_{v_n}\}$ is a set of value domains;
- $V = \{v_1, v_2, \dots, v_n\}$ is a set of variables with domains $D_{v_1}, D_{v_2}, \dots, D_{v_n}$;
- $V_C \subseteq V$ is a set of variables marked as *control variables*;
- $IV \in D_{v_1} \times D_{v_2} \times \cdots \times D_{v_n}$ 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 selected from the sets introduced above (Tr, In and En). 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 are selected 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. XSTS defines the following operations:

Basic operations Basic operations contain no inner (nested) operations. *Assignments* assign a given value v from domain D_n to variable V_n. *Havocs* behave likewise, except the value is not predetermined; giving a way to assign random value to a variable. Lastly, *assumptions* check a condition, and can only be executed if their condition evaluates to *true*.

Composite operations Composite operations contain other operations, and can be used to describe complex control stuctures. Note that while these are composite operations, their execution is still atomic; meaning that an *assume* operation prevents its containing operations from firing. *Sequences* are essentially multiple operations executed after each other, while *parallels* execute all operations at the same time. And lastly, *choices* model non-deterministic choices between multiple operations; one and only one branch of the choice operation is selected for execution, but that selected operation can still only be executed atomically - e.g., a single failing assume operation can prevent execution.

2.4.2 Simple example

Below is an example XSTS model defined in the language. For a more exact presentation please see Section A.2 in the Appendix.

```
itt majd egy szép xsts példa lesz a running example-re (vagy valami hasonlóra).
Akár lehetne egy Gamma által generált statechart is.
```

Statechart model xsts változata legyen

2.5 Related Work

Add related work

Chapter 3

Gamma Activity Language

The high-level nature of SysML activities means they are easy to model complicated behaviours of distributed system, but leads to an increasingly hard time trying to run formal verification on them. In the real world, SysML models often contain state machines that call activity diagrams using a state's *do behaviour*, because often times this makes the design process easier. However, models containing *do activity* cannot be verified directly; the activities have to be pre-processed and transformed into something else, like javascript or state machines (). As discussed above in Section 2.2.3, we can define a mapping between activities and Petri nets (which have exact semantics), however, that mapping is not exhaustive.

cite Pragmatic Verification and Validation of Industrial Executable SysML Models

The pre-existing Gamma Statechart Composition Framework (Section 2.3) implements this transformation-verification pipeline, however, it does not include activity diagrams. For this reason, I propose (Chapter 3) the *Gamma Activity Language*, and integrate it (Chapter 4) into the pipeline shown in Figure 2.7.

3.1 Language Design

During the design phase of the language SysML served as the basis when deciding what is should and should not contain. When deciding the scope of the language the following statement served as target: The language should be easy to transform into XSTS, while also following SysML activity semantics.

To conform to the previous statements, the language should resemble SysML, however, should be as *lightweight* as possible. For this reason, we determined a sub-set of the SysML activity feature set that the language will contain.

3.1.1 SysML Feature Sub-set

ez a rész csak jegyzet

Initial/final node, action node, composite activity node, decision/merge node, fork/join node, pins, control/data flows.

Do behaviour:

call activity action on transitions is not supported, would have to be inlined and flattened, atomic functions, interlaced with line execution semantics (future work)

- call activity context függő - hosszú (nem atomikus) dolgot nem lehet akárhol meghívni (transition, sima action, stb) - validáció: inline környezetben tilos várakozás és ciklus (vagy csak x mélységig) - sok köztes nem stabil állapot - nagy változás a trafóban

ezért csak do activity van

3.2 Formal Definition

Finalise formalism

In order to offer mathematical precision, formal verification methods require formally defined models with clear semantics. In this section I present the formal definition of the novel GAL formalism.

Definition 3 (Gamma Activity Language). A Gamma Activity is a tuple of $GA = (N, P, F, G, D, F_{Action})$, where:

- $N = N_{Initial} \bigcup N_{Final} \bigcup N_{Pseudo} \bigcup N_{Decision} \bigcup N_{Merge} \bigcup N_{Action}$ is a set of *Nodes*, where $N_{Initial}$ contains the *Initial* nodes, N_{Final} contains the *Final* nodes, N_{Pseudo} contains the *Pseudo* nodes, $N_{Decision}$ contains the *Decision* nodes, N_{Merge} contains the *Merge* nodes and N_{Action} contains the *Action* nodes;

- $P \subseteq P_{In} \bigcup P_{Out}$ is a set of two types of pins, where $P_{In} : N_{Action} \rightarrow \{p_1, \ldots, p_n\}$ and $P_{Out} : N_{Action} \rightarrow \{p_1, \ldots, p_n\}$ are the set of *InputPins* and *OutputPins*, respectively, with domains $\{d_1, \ldots, d_n\} \subseteq D$;

- $F \subseteq F_C \bigcup F_D$, where $F_C = \{F_{C1}, \ldots, F_{Cn}\}$ and $F_D = \{F_{D1}, \ldots, F_{Dn}\}$ are the control and data flows, respectively. Let us denote the input/output flows of node n as $\delta(n)$ and $\Delta(n)$, and the source/target pins of flow f as $\phi(f)$ and $\Phi(f)$. For any given node n, $\delta(n) \neq \Delta(n)$ and for any given action node n_a $\Phi(f) \in P_{In}(n_a) \forall f \in F_D \bigcap \delta(n_a)$ and $\phi(f) \in P_{Out}(n_a) \forall f \in F_D \bigcap \Delta(n_a)$ shall always hold. This means, that a flow cannot be input and output to the same node at the same time, and for a given action node, all input/output flows shall be associated with an input/output token, respectively;

- $G : F \rightarrow \{True, False\}$ is a function determining whether a given flow is enabled;

- $D = \{D_1, D_2, \ldots, D_n\}$ is a set of value domains;

- $F_{Action} : N \rightarrow D$ is a function running the contained action, and returning it's result values. ▪

Informally, Gamma Activities are composed of *nodes* (*Initial/Final*, *Decision/Merge*, *Action* and *Pseudo*) and flows (*Control* and *Data*) in between them. A given action can also have any number of *Pins* with a domain, for which, there must be one and only one *data* flow connected to the node. An action node also has a special F_{Action}, which implements its specific behaviour, and returns the values of its output pins.

For the most part, these elements have a similar behaviour as their SysML counter parts; however, there is a crucial difference regarding how a flow transmits a token. Figure 3.1 shows a simple data flow between two nodes, connected via their pins. When this flow is fired, the data inside the pins are transferred instantly, without any pseudo-state in between.

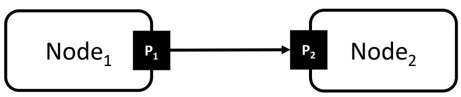

Figure 3.1: An example SysML data flow

Contrarily, in Gamma Activity Language, the equivalent data flow does indeed contain the given node, creating an intermediate state, where the token is in neither of the nodes. This would look like a *Central Buffer* in SysML (Figure 3.2). The reason for this is simplicity; by creating this intermediate state (and others), it is easier to state the set of transitions necessary to formally define the semantics of the language.

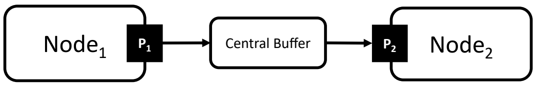

Figure 3.2: An example SysML data flow with central buffer

Definition 4 (Gamma Activity State). The state of a GA is defined the following:

- $S_N : N \to \{Idle, Running, Done\}$ is a function returning the *state* of a node;
- $S_F : F \to \{Empty, Full\}$ is a function returning the *state* of a flow;
- $V_N : N \to D$ is a function returning the current *value* contained in a node;
- $PV_N : N \times P \to D$ is a function returning the current *value* contained in a node's pin;
- $V_F : F \to D$ is a function returning the current *value* contained in a flow. ▪

Informally, the state of Gamma Activities are determined by the nodes' *state* (*Idle*, *Running*, *Done*) and their (or their pins') values, the flows' *state* (*Empty*, *Full*) and their values. For example, given an action node n and one of it's pins p_1, $PV_N(n, p)$ would give us the exact value that pin contains in this instance. This gives us the power - contrary to the Activity-PN mapping introduced in Section 2.2.3 - to formally define the values contained in tokens and nodes.

Definition 5 (State Transition Functions). The behaviour of the system is described by three functions (F_{In}, F_{Run} and F_{In}). At any one time, a node $n \in N$ is selected, and an *enabled* function is executed non-deterministically. These functions define the state-transitions for our system:

$$F_{In} : N \to \begin{cases} \text{select } f_s f_s \in \delta(n) \wedge S_F(f) = \mathit{Full} \\ V'_N(n) = V_F(f_s) \\ S'_F(f_s) = \mathit{Empty} \\ S'_N(n) = \mathit{Running}, & \text{if } n \in N_{Merge} \\[1ex] PV'_N(n, \Phi(f)) = V_F(f) : \forall f \in \delta(n) \bigcap F_D \\ S'_F(f) = \mathit{Empty} : \forall f \in \delta(n) \\ S'_N(n) = \mathit{Running}, & \text{if } n \in N_{Action} \\[1ex] V'_N(n) = V_F(f) : f \in \delta(n) \bigcap F_D \\ S'_F(f) = \mathit{Empty} : \forall f \in \delta(n) \\ S'_N(n) = \mathit{Running}, & \text{otherwise} \end{cases} \tag{3.1}$$

In words, the function F_{In} takes the tokens from the input flows, and puts it in the given node, thus starting it's execution. The rule how it selects when is enabled, and which flows to empty is determined by the node's type:

- if it is a *Merge* node, then it must only forward one, and only one token,
- if it is an *Action* node, then it must forward all tokens, along with their values into the associated pins,
- otherwise, it must forward all tokens.

This way we can ensure the behaviours of the simple 'LAS' nodes, as well as the 'IR' nodes.

$$F_{Run} : N \rightarrow \begin{cases} V'_N(n) = F_{Action}(n, V_N(n)) \\ S'_N(n) = Done, & \text{if } n \in N_{Action} \\ \\ S'_N(n) = Done, & \text{otherwise} \end{cases} \tag{3.2}$$

In words, the function F_{Run} stochastically sets the given node's state to *Done*. If the given state is of type *Action*, then it must have a deeper semantic associated with it, thus the associated F_{Action} function must also be called, which implements it's specific behaviour (more detail below).

$$F_{Out} : N \rightarrow \begin{cases} \text{select } f_s \in \Delta(n) \wedge S_F(f_s) = Empty \wedge G(f_s) = True \\ V'_F(f_s) = V_N(n) \\ S'_F(f_s) = Full \\ S'_N(n) = Idle, & \text{if } n \in N_{Decision} \\ \\ V'_F(f) = PV_N(n, \phi(f)) : \forall f \in \Delta(n) \bigcap F_D \\ S'_F(f) = Full : \forall f \in \Delta(n) \\ S'_N(n) = Idle, & \text{if } n \in N_{Action} \\ \\ S'_F(f) = Full : \forall f \in \Delta(n) \\ S'_N(n) = Idle, & \text{otherwise} \end{cases} \tag{3.3}$$

The function F_{Our} takes the token from node, and puts it in its output flows, thus ending it's execution. The rule how it selects when is enabled, and which flows to fill is determined by the node's type:

- if it is a *Decision* node, then it must only forward the token on one of its *enabled* flows,
- if it is an *Action* node, then it must forward all tokens, along with their values from the associated pins,
- otherwise, it must forward all tokens. ▪

Figure 3.3 depicts these three functions, and how they change the state of a given node.

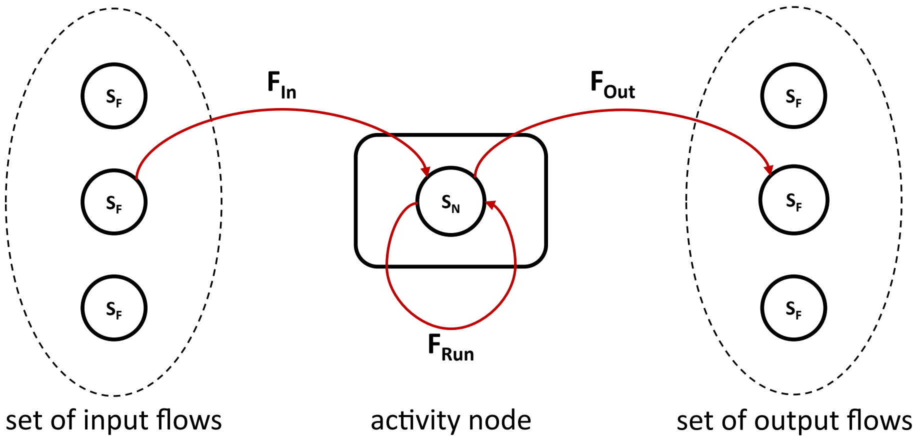

Figure 3.3: An illustration of activity node state and change functions

3.3 Domain-specific Language

Like the Gamma Statechart Language, the Gamma Activity Language is intended to be a first-class citizen in the Gamma Framework, thus it must have a domain-specific language to represent in a textual way. This implementation draws highly from the SysMLv2 [12] language design, while also fitting into the already existing language family of Gamma (Section 2.3).

3.3.1 Meta-model

Due to the complexity of the final meta-model of the language, I have split it into multiple parts for easier understanding.

3.3.1.1 Pins

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut vehicula turpis eget enim maximus, vel rutrum dui ullamcorper. Nulla enim ex, dapibus non aliquam vitae, molestie quis magna. Maecenas mattis turpis non ex feugiat, vitae pulvinar nisl vulputate.

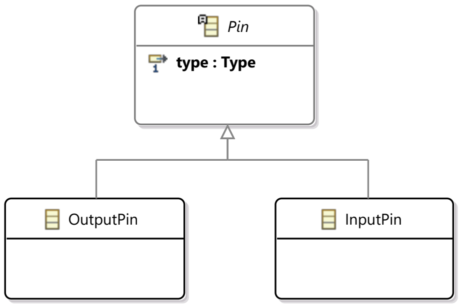

Figure 3.4: The Pins structure

Input Pin Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut vehicula turpis eget enim maximus, vel rutrum dui ullamcorper. Nulla enim ex, dapibus non aliquam vitae, molestie quis magna. Maecenas mattis turpis non ex feugiat, vitae pulvinar nisl vulputate.

Output Pin Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut vehicula turpis eget enim maximus, vel rutrum dui ullamcorper. Nulla enim ex, dapibus non aliquam vitae, molestie quis magna. Maecenas mattis turpis non ex feugiat, vitae pulvinar nisl vulputate.

3.3.1.2 Flows

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut vehicula turpis eget enim maximus, vel rutrum dui ullamcorper. Nulla enim ex, dapibus non aliquam vitae, molestie quis magna. Maecenas mattis turpis non ex feugiat, vitae pulvinar nisl vulputate.

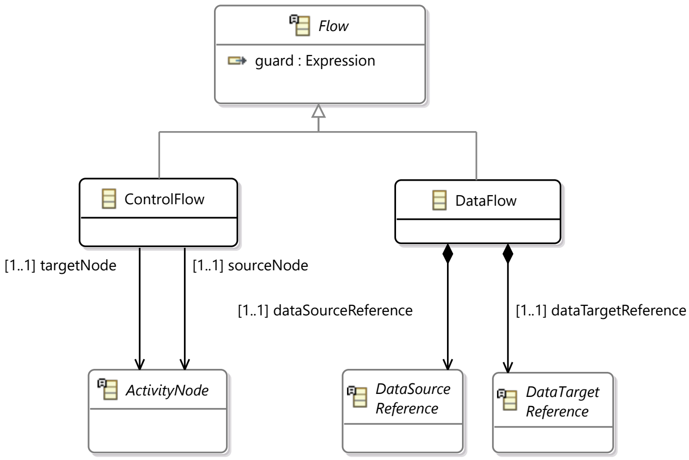

Figure 3.5: The Flows structure

Control Flow Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut vehicula turpis eget enim maximus, vel rutrum dui ullamcorper. Nulla enim ex, dapibus non aliquam vitae, molestie quis magna. Maecenas mattis turpis non ex feugiat, vitae pulvinar nisl vulputate.

Data Flow Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut vehicula turpis eget enim maximus, vel rutrum dui ullamcorper. Nulla enim ex, dapibus non aliquam vitae, molestie quis magna. Maecenas mattis turpis non ex feugiat, vitae pulvinar nisl vulputate.

3.3.1.3 Activity Nodes

In the following, I will talk about the different kinds of *ActivityNodes* and their special meanings. All nodes are considered *LAS* nodes by default. The meta-model described in this section can be seen in Figure 3.6.

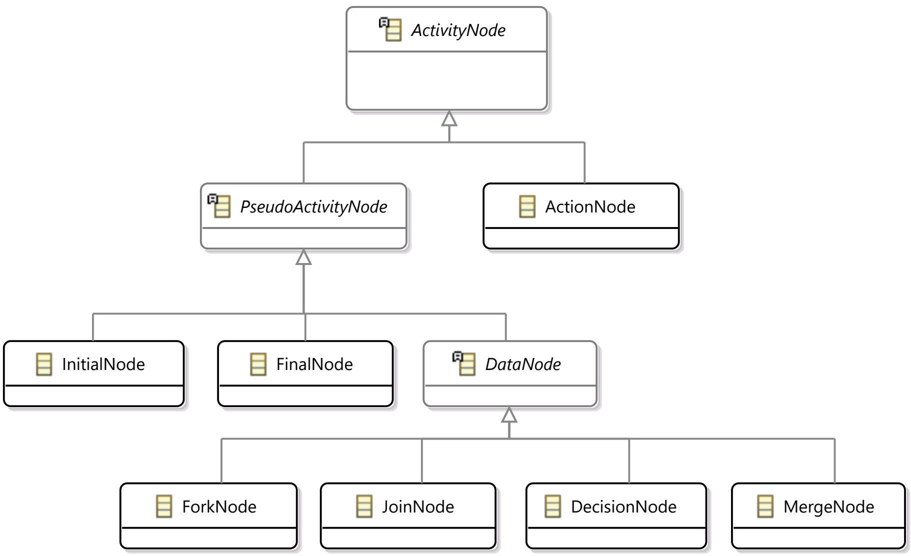

Figure 3.6: The Activity Node structure

Pseudo Activity Node *PseudoActivityNodes* are nodes, that do not represent a specific action, however are needed to convey specific meanings, e.g., the initial active node, or a decision between flows.

Initial Node *InitialNodes* have one token in them when the containing activity is started. They shall only have (one or more) outgoing flows, and no input flows.

Final Node When an activities *FinalNode* gets a token the containing activity is considered *Done*; after which the activity does not process any more tokens.

Data Node *DataNodes* encapsulate the meaning of *data* inside activities. A token may contain data (or value) of any kind, but that token can only travel to and from data *sources* and *targets*. More about this in Section 3.3.1.2.

Fork Node *ForkNodes* are used to model parallelism, by creating one token on each of its output flows when executed. Fork nodes shall only have one input flow.

Join Node *JoinNodes* are the pair of fork nodes; the additional created tokens are swallowed by this node, by only sending out one token, regardless of the number of input flows.

Decision Node *DecisionNodes* create branches across multiple output flows. An input flows token is removed, and sent out to one, and only one of its output flows - depending on which of the output flows are *enabled* (see Section 3.3.1.2).

Merge Node *MergeNodes*

3.3.1.4 Root Structure

Every model has to have a root element structure; Activity Language is not any different. The meta-model described in this section can be seen in Figure 3.7.

Activity Declaration All elements inside an activity are contained in a root *ActivityDeclaration* element. It contains *Pins* needed for value passing (see Section 3.3.1.1) and a *Definition*. A declaration can be *InlineActivityDeclaration*, which means they are declared in an other declaration, or *NamedActivityDeclaration*, which is a standalone activity declaration. The difference will be clarified in **??**.

Definition The definition *defines* how the activity is described; using activity nodes, or by the Gamma Activity Language[1]. *ActionDefinition* contains a single *Block*[2], which is executed as-is when the activity is executed[3]. *ActivityDefinition* contains *ActivityNodes* (Section 3.3.1.3) and *Flows* (Section 3.3.1.2).

[1]Gamma Activity Language is a lightweight programming language-like construct for writing simple algorithms

[2]A *Block* contains multiple *Actions* which are executed one after the other

[3]This fact means, that if one used Action to define an activity, that activity is executed atomically; it will not be interlaced with other XSTS transitions. See Chapter 4.

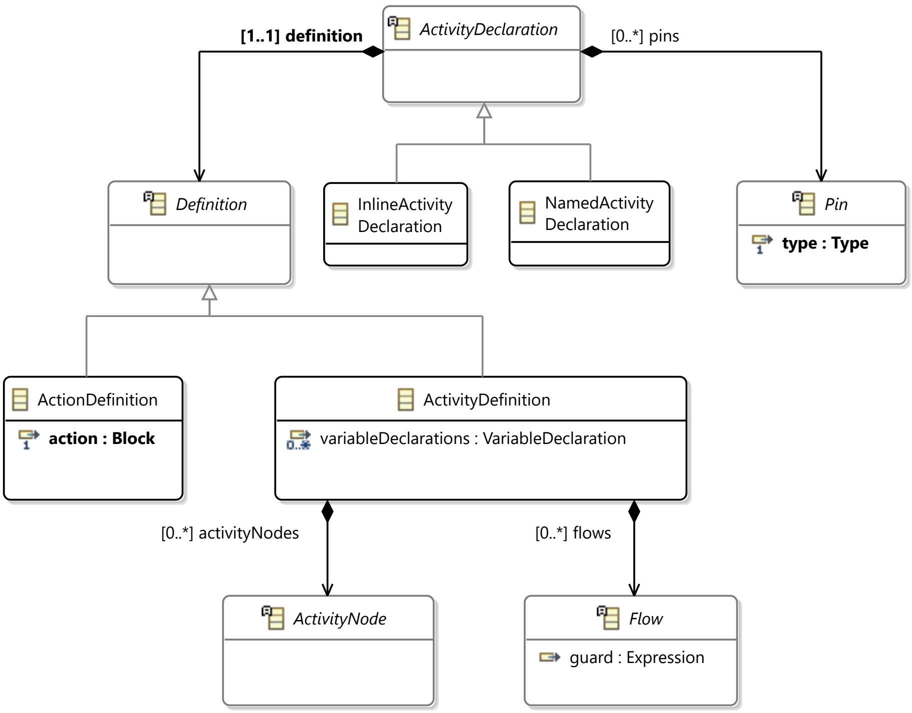

Figure 3.7: The root structure of the language

3.3.1.5 Composing Activities

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut vehicula turpis eget enim maximus, vel rutrum dui ullamcorper. Nulla enim ex, dapibus non aliquam vitae, molestie quis magna. Maecenas mattis turpis non ex feugiat, vitae pulvinar nisl vulputate.

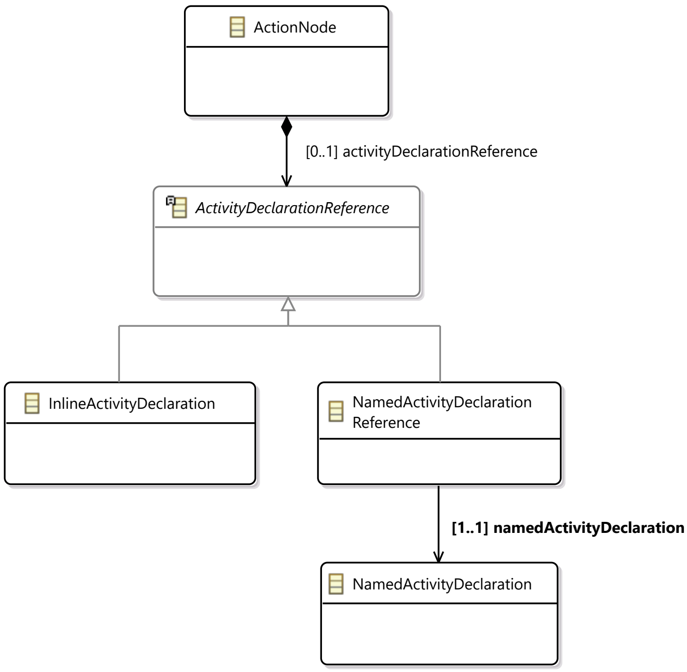

Figure 3.8: The

3.3.1.6 Data Source-Target Reference

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut vehicula turpis eget enim maximus, vel rutrum dui ullamcorper. Nulla enim ex, dapibus non aliquam vitae, molestie quis magna. Maecenas mattis turpis non ex feugiat, vitae pulvinar nisl vulputate.

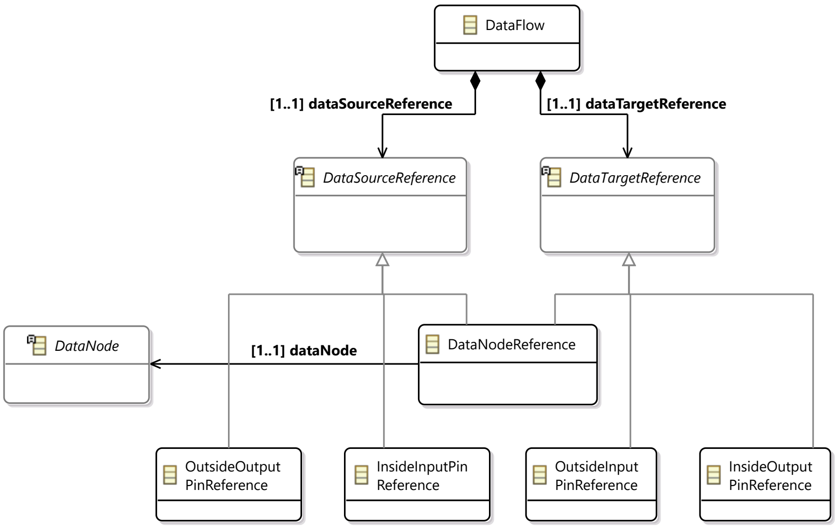

Figure 3.9: The Data node reference structure

3.3.1.7 Pin Reference

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut vehicula turpis eget enim maximus, vel rutrum dui ullamcorper. Nulla enim ex, dapibus non aliquam vitae, molestie quis magna. Maecenas mattis turpis non ex feugiat, vitae pulvinar nisl vulputate.

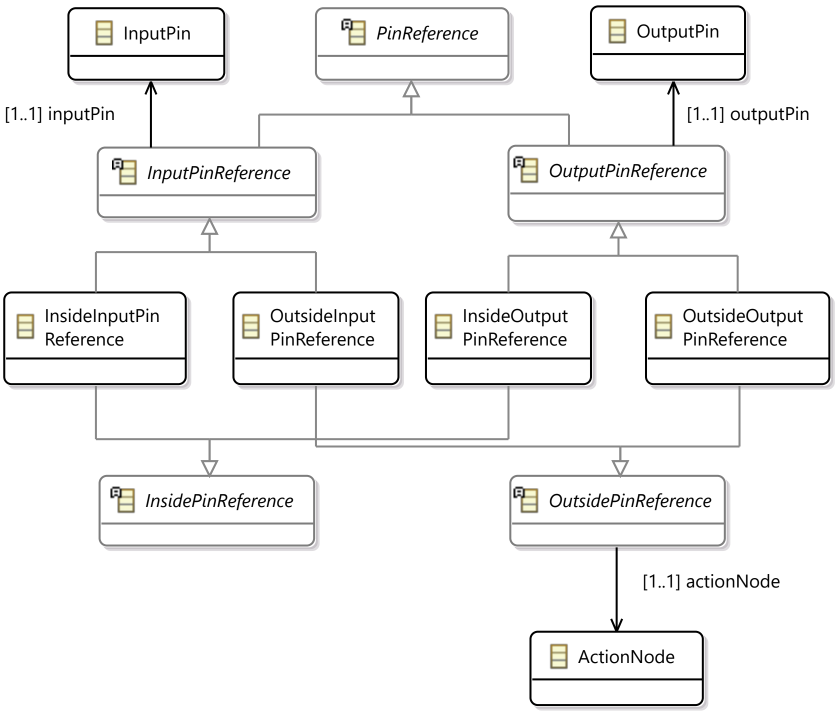

Figure 3.10: The Data node reference structure

3.3.1.8 Gamma Extension

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut vehicula turpis eget enim maximus, vel rutrum dui ullamcorper. Nulla enim ex, dapibus non aliquam vitae, molestie quis magna. Maecenas mattis turpis non ex feugiat, vitae pulvinar nisl vulputate.

3.3.2 Concrete Syntax

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.

Gamma extension metamodel részlet, leírni, hogy mivel és miért kellett kiegészíteni

3.3.2.1 Example

In this section, you can see the Gamma Activity Language representation of the compilation activity (Figure 2.3)

```
activity CompilationProcess {
  var errors : boolean := false

  initial Initial

  merge Merge

  fork Fork

  action Read1 : activity(
    out p : integer
  )
  action Compile1 : activity(
    in p : integer
  )

  action Read2 : activity(
    out p : integer
  )
  action Compile2 : activity(
    in p : integer
  )

  join Join

  decision Decision

  action Edit

  final Final

  control flow from Initial to Merge
  control flow from Merge to Fork
  control flow from Fork to Read1
  data flow from Read1.p to Compile1.p
  control flow from Fork to Read2
  data flow from Read2.p to Compile2.p
  control flow from Compile1 to Join
  control flow from Compile2 to Join
  control flow from Join to Decision
  control flow from Decision to Edit [errors]
  control flow from Edit to Merge [!errors]
  control flow from Decision to Final
}
```

Chapter 4

Integrating Activity Language Into Gamma

We defined the language, however many unanswared questions how to transform side by side to to verify what semantics etc

how to integrate into gamma, action, do action, etc, transition gamma használja az xsts-t -> xsts-re kell trafózni

4.1 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

4.2 Activities as State-based Models

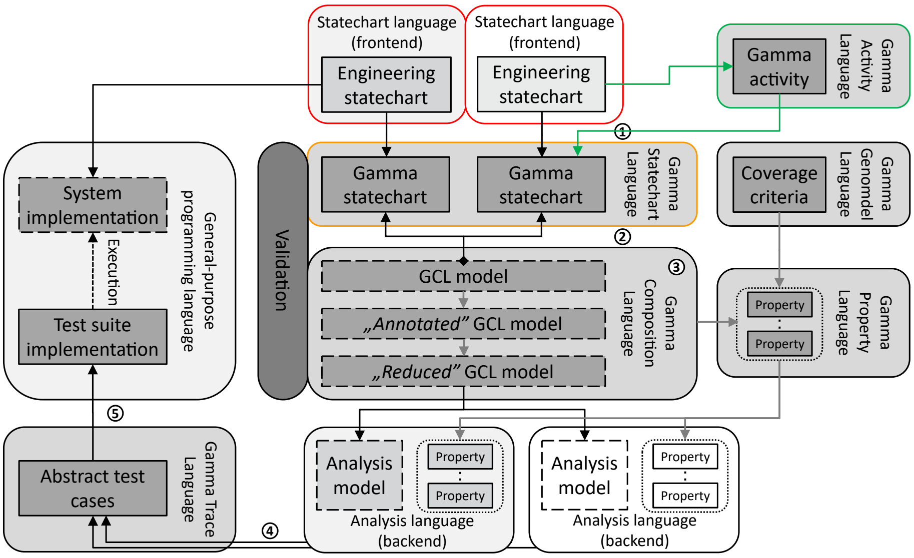

Figure 4.1: Overview of my modification to the Gamma framework. Added parts are outlined with green, while edited parts are outlined with orange.

formalizmus trafózása statechart mellett xsts-re. a végeredmény miért ekvivalens

4.3 Implementation Regards

implementációs nehézségek, kérdések, stb (commitok, line change, projektek magas szinten)

Chapter 5

Evaluation

spacecraft model bevezetése, leírása statechart elemek referálása, activity-k megírása kézzel (modell mehet apendix-be) mérések végzése és leírása

Chapter 6

Conclusion

ami nem ment evaluationbe - lassú - nem skálázódik - de egy első lépés, hogy unified verifikációt lehessen csinálni komplikált rendszereken

Future Work - sysml activity nagyobb mappelése - signalok bevezetése - activity külön komponensként - activity inline-olása transition action-re

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.

List of Figures

Todo list

Bibliography

[1] Technical operations international council on systems engineering incose. incose systems engineering vision 2020. technical report. URL https://sebokwiki.org/wiki/INCOSE_Systems_Engineering_Vision_2020.

[2] Mbse wiki. URL https://www.omgwiki.org/MBSE/doku.php?id=start.

[3] David A. Larsen K. G. Håkansson J. Pettersson P. Yi W. Hendriks M. Behrmann, G. Uppaal 4.0. 2006.

[4] G Bellinger. Modeling & simulation: An introduction. 2004. URL http://www.systems-thinking.org/modsim/modsim.htm.

[5] DoD. Dod modeling and simulation (m&s) glossary. *DoD Manual 5000.59-M. Arlington, VA, USA: US Department of Defense*, 1998. URL https://apps.dtic.mil/sti/pdfs/ADA349800.pdf.

[6] D. Dori. Object-process methodology: A holistic system paradigm. *New York, NY, USA: Springer.*, 2002.

[7] A. Moore R. Steiner Friedenthal, S. and M. Kaufman. A practical guide to sysml: The systems modeling language, 3rd edition. *MK/OMG Press.,*, 2014.

[8] Bence Graics, Vince Molnár, András Vörös, István Majzik, and Dániel Varró. Mixed-semantics composition of statecharts for the component-based design of reactive systems. *Software and Systems Modeling*, 19(6):1483–1517, Nov 2020. ISSN 1619-1374. DOI: 10.1007/s10270-020-00806-5. URL https://doi.org/10.1007/s10270-020-00806-5.

[9] Object Management Group. Omg system modeling language. URL https://www.omg.org/spec/SysML/.

[10] Object Management Group. Mda foundation model. omg document number ormsc/2010-09-06. 2010.

[11] Object Management Group. Semantics of a foundational subset for executable uml models. 2018. URL https://www.omg.org/spec/FUML/1.4/.

[12] Object Management Group. Systems modeling language version 2 (sysmlv2). 2020. URL https://www.omgsysml.org/SysML-2.htm.

[13] D. Harel and A. Pnueli. On the development of reactive systems. In Krzysztof R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg. ISBN 978-3-642-82453-1.

[14] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987. ISSN 0167-6423. DOI: `https://doi.org/10.1016/0167-6423(87)90035-9`. URL `https://www.sciencedirect.com/science/article/pii/0167642387900359`.

[15] Osman Hasan and Sofiène Tahar. Encyclopedia of information science and technology, third edition. pages 7162–7170., 2015. DOI: `https://doi:10.4018/978-1-4666-5888-2.ch705`.

[16] Benedek Horváth, Bence Graics, Ákos Hajdu, Zoltán Micskei, Vince Molnár, István Ráth, Luigi Andolfato, Ivan Gomes, and Robert Karban. Model checking as a service: Towards pragmatic hidden formal methods. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381352. DOI: `10.1145/3417990.3421407`. URL `https://doi.org/10.1145/3417990.3421407`.

[17] 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`.

[18] Balcer MJ. Mellor SJ. Executable uml: A foundation for model- drivenarchitecture. *The Addison-Wesley Object TechnologySeries: Addison-Wesley Professional*, 2002.

[19] Vince Molnár, Bence Graics, András Vörös, István Majzik, and Dániel Varró. The Gamma statechart composition framework: design, verification and code generation for component-based reactive systems. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, pages 113–116. ACM, 2018. DOI: `10.1145/3183440.3183489`.

[20] Milán Mondok. Extended symbolic transition systems: an intermediate language for the formal verification of engineering models. *Scientific Students' Association Report*, 2020.

[21] Zoltán Micskei Márton Elekes. Towards testing the uml pssm test suite. 2021.

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

[23] OMG. Precise semantics of uml state machines (pssm). *formal/19-05-01.*, 2019.

[24] Gianna Reggio, Maurizio Leotta, and Filippo Ricca. Who knows/uses what of the uml: A personal opinion survey. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems*, pages 149–165, Cham, 2014. Springer International Publishing. ISBN 978-3-319-11653-2.

[25] Hajdu A. Vörös A. Micskei Z. Majzik I. Tóth, T. Theta: a framework for abstraction refinement-based model checking. *Stewart, D., Weissenbacher, G. (eds.), Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design.*

Appendix

A.1 Gamma Language

ide berakhatom ugyan azt az appendix-et ami a cikkben van? vagy inkább ne, cite pont erre van (ezesetben ez nem is lesz itt)

A.2 XSTS Language

In this appendix I introduce the exact language constructs for the XSTS language

A.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 State Transition Functions 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 $D_{name} = \{literal_1, \ldots, literal_n\}$.

A.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).

A.2.2.1 Basic operations

Operations make up the set *Ops* introduced in State Transition Functions 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>
```

A.2.3 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>
```

A.2.4 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 State Transition Functions 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>
}
```

A.3 Gamma Activity Language

A.3.1 Language elements

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

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 **??**

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* A.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;
  }
}
```

A.4 Spacecraft Model