MŰEGYETEM 1782

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 introduction

Chapter 2

Background

In this chapter I present 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 section also goes into detail about SysML (Section 2.1.1), which is a general-purpose modeling language for system design. Next, I talk about the concept of 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 Statechart Composition Framework, which is a tool for modeling and verifying component-based reactive systems based on statecharts (Section 2.3), and a low-level formalism called XSTS (Section 2.4), used as an intermediary language for model checking by Gamma.

2.1 Model-based Systems Engineering

The INCOSE SE Vision 2020 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.” [1]

Applying MBSE is expected to provide significant benefits over the document document-centric 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 facilities SysML provides. In the following section, I present two of the most used diagram types in SysML; State Machine Diagrams and Activity Diagrams.

State Machine Diagram

Reactive systems are all around us in our daily lives; in smart phones, avionics systems or even our calculators. Oftentimes, 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], while also being formally defined.

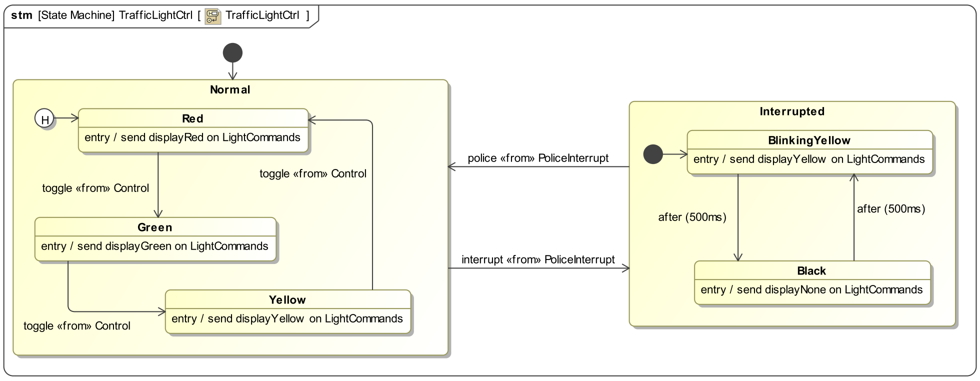

stm [State Machine] TrafficLightCtrl [TrafficLightCtrl]
Normal
H
Red
entry / send displayRed on LightCommands
toggle «from» Control
toggle «from» Control
Green
entry / send displayGreen on LightCommands
Yellow
entry / send displayYellow on LightCommands
toggle «from» Control
police «from» PoliceInterrupt
interrupt «from» PoliceInterrupt
Interrupted
BlinkingYellow
entry / send displayYellow on LightCommands
after (500ms)
after (500ms)
Black
entry / send displayNone on LightCommands

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 abstraction constructs support the concise modeling of state machines for engineers, but hinder their formal verification. This abstraction gap can be bridged using a transformation tool, such as Gamma (see Section 2.3) that maps these high-level constructs into low-level analysis formalisms.

Figure 2.1 shows a state machine modeling the behaviour of a traffic light controller. The *TrafficLightCtrl* component has three ports, two inputs (*Control* and *PoliceInterrupt*) for user input, and one output (*LightCommands*) for controlling (turning on and off) the specific light it is connected to. The state machine changes between the *red-green-yellow* lights, upon a *toggle* command (Normal state - main cycle). However, if a *police* signal is received, it starts turning on and off the yellow light every 500ms (Interrupted state - blinking cycle).

Activity Diagram

State machines cannot describe the complicated semantics of distributed systems with concurrent, parallel behaviour, where the *interesting* thing is what the system *does* step-by-step.(e.g., batch processing). 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 modeling elements used in this work. In the following, I introduce the different modeling elements of SysML activities and show an example of a SysML activity diagram.

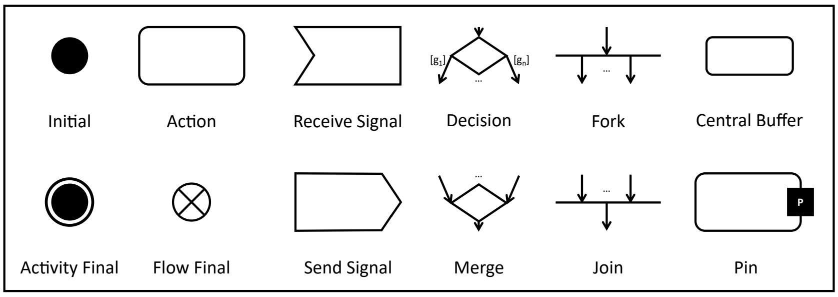


Figure 2.2: Artifacts of SysML activity diagrams.

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 limitations over which node can run, and which cannot; a given node is considered *running*, only when it contains a token. Tokens *flow* between nodes, carrying with them a given value - this value can also 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 convert 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 starts from the initial node, and ends with a *Flow*

Final or *Activity Final* node. *Fork* nodes generate tokens on all of their output flows, and *Join* nodes forward the tokens, only when all input flows contain one - thus, the two nodes complement each other. 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 send it out on its single enabled output flow.

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

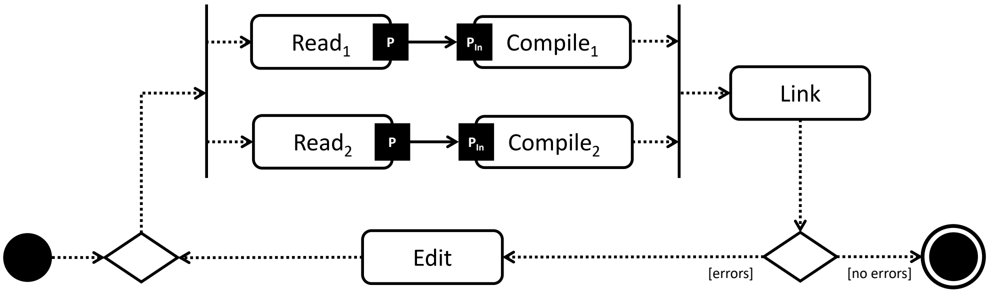


Figure 2.3: The activity of editing, compiling and linking two files.

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 *transferred* to the compiler module. We want to compile the two different files in *parallel*, thus we split the control flow using a *fork* node. Once the files are compiled, we *link* them - since linking requires both files, it is preceded by a *join* node. Finally, if the resulting code contains errors, we *edit* the source files and start over - otherwise we are done.

2.2 Formal Verification

In order to raise the reliability of system models, various analysis techniques are used to verify their correctness. Among these techniques are *unit tests*, *module tests*, *system tests* and *acceptance tests*. However, no matter how many kinds of tests we use, the test cases are ultimately constructed by people - rendering them as error prone as the model. Formal verification tools are meant to extend these analysis techniques by not specifying a given *sequence* of event, but rather specifying the undesirable state of the system. Formal verification methods are primarily based on theoretical computer science fundamentals like logic calculi, automata theory and strongly type systems. 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 the intended behaviour. Due to the mathematical nature of the analysis, 100% accuracy can be guaranteed.

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 the ones used in UPPAAL[1] [3] or Theta[2] [26] can answer this question, and can even return a *proof* (i.e., a diagnostic trace of the model) that M indeed does satisfy said requirement[3].

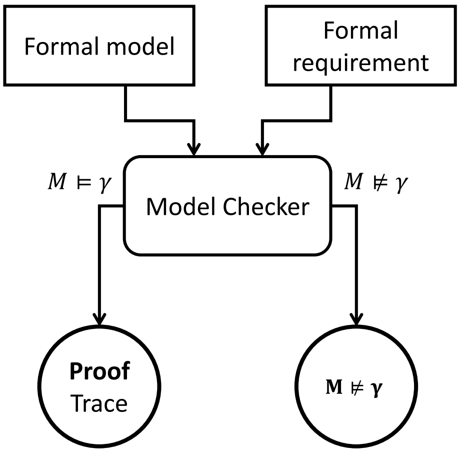
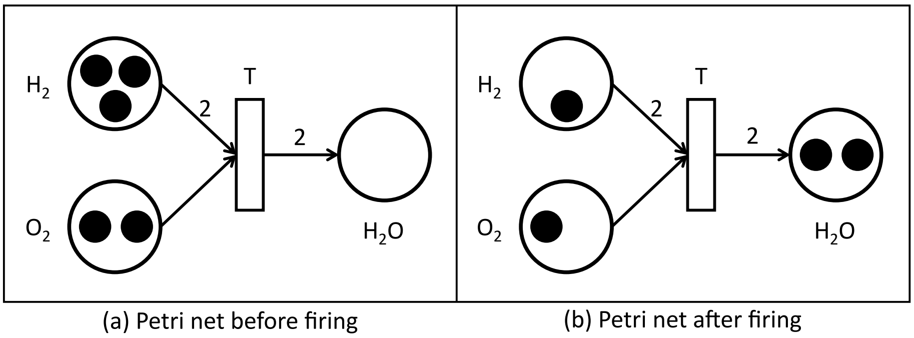


Figure 2.4: An illustration of model checking.

2.2.2 Petri Nets



(a) Petri net before firing

(b) Petri net after firing

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

Petri nets are a widely used formalism to model concurrent, asynchronous systems [22]. The formal definition of a Petri net [25] 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. ∎

[1] https://uppaal.org/

[2] https://inf.mit.bme.hu/en/theta

[3] These proofs usually come in the form of an execution trace

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

2.2.3 Activities as Petri Nets

Formal verification requires models to be specified using *mathematical* precision, i.e., the modeling language must have a formal syntax and semantics, however UML/SysML does not have precise semantics [21, 23, 18]. Huang et al. in [17] propose a way to *partially* map SysML activity diagrams to Petri nets, giving a denotational semantics to the subset of SysML Activity Diagrams. In the following, I will summarise their work, as my main contributions (see Chapter 3) build on it.

Constrained Subset of SysML

Since UML/SysML Activity Diagrams do not have precise execution semantics, the Petri net mapping can can be defined only 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, i.e., multiplicity lower bounds are strictly positive.

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

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 flows/pin is consumed and the number of tokens associated with an output flows/pin is added. In SysML activity diagrams, the execution semantics of all nodes - except *decision* and *merge* nodes - are 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 SysML activity diagrams, merge nodes, decision nodes 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, 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 the final step, these arcs must be split in two by inserting a transition or a place in the middle (with a weight of 1), making the model conform to the formalism.

Example Mapping

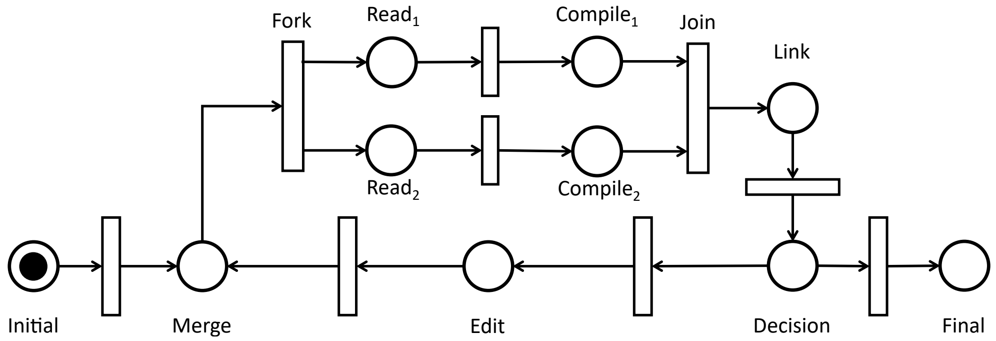


Figure 2.6: Example mapping from activity diagram to Petri net.

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 about the Petri net mapping, please refer to [17].

2.3 The Gamma Statechart Composition Framework

The Gamma Statechart Composition Framework[4] [8] is an integrated tool to support the design, verification and validation of, 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[5]. Gamma supports the formal verification of the assembled system 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 Expression Language (GEL)** is a lightweight expression language, created to describe value expression (addition, subtraction, etc.) in actions.

[4] https://inf.mit.bme.hu/en/gamma

[5] The composition language bears close similarities to the *ibd* language in SysML

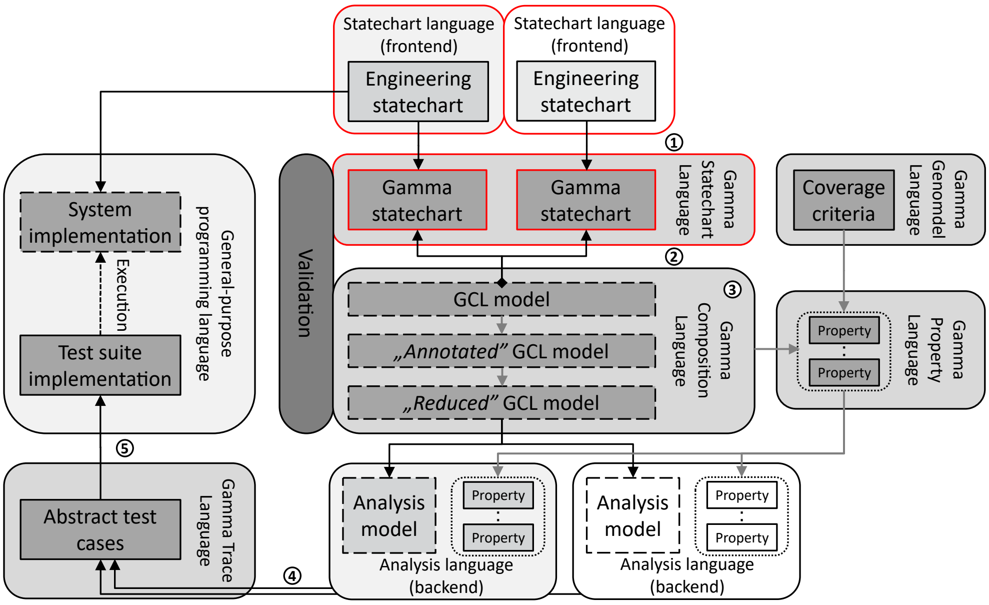


Figure 2.7: The overview of model transformation chains and modeling languages of the Gamma framework [8]. The parts relevant to this work have been marked with red outline.

- The **Gamma Action Language (GAL)** is a lightweight action language, created to describe actions on statechart transitions.
- 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.
- The **Gamma Property Language (GPL)** is a property language supporting the definition (CTL*) 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.

The component integration and verification workflow in Gamma is as follows. Optionally, statechart models defined in supported modeling tools (front-ends) can be imported into Gamma (Step 1), which can be integrated according to well-defined execution and interaction semantics (Step 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). For a more detailed description, see [8].

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

Listing 2.1: The traffic light controller state machine in the textual representation of Gamma's statechart language.

2.3.1 Example Statechart

Listing 2.1 shows the Gamma Statechart representation of the SysML State Machine introduced in Figure 2.1.

2.4 Extended Symbolic Transition System

In this section, I introduce the Extended Symbolic Transition System [20] language, which is a low-level modeling formalism designed to bridge the abstraction 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:

- $D = \{d_{v_1}, d_{v_2}, \ldots, d_{v_n}\}$ is a set of value domains;
- $V = \{v_1, v_2, \ldots, v_n\}$ is a set of variables with domains $d_{v_1}, d_{v_2}, \ldots, d_{v_n}$;
- $V_C \subseteq V$ 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 - operations only from the In set can be executed. Operations from the In set can fire only in the initial state and nowhere else. After that, operations from 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 basic and composite 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 a random value to a variable.
- Lastly, *assume* operations check a condition, and can be executed only if their condition evaluates to *true.*

Composite operations Composite operations contain other operations, and can be used to describe complex control stuctures.

- *Sequences* are essentially multiple operations executed one after the other.
- *Parallels* are like sequences, however they execute all operations in parallel.

- And lastly, *choices* model non-deterministic choices between multiple operations; one and only one branch of the choice operation is selected for execution.

Note that while these are composite operations, their execution is still atomic; i.e., a potential false evaluation of a containing assume operation prevents the execution of all operations in that particular branch. E.g., if a sequence's 2nd operation is an assume operation, which cannot execute, then the whole sequence operation will be prevented from execution.

2.4.2 Traffic Light Controller Example

Listing 2.2 is the textual representation of the traffic light controller statechart (Figure 2.1) transformed[6] to XSTS. For a more exact presentation please see Section A.1.

2.5 Related Work

Add related work

[6] The transformation was done using Gamma

```
type ActivityNodeState : { __Idle__, __Running__, __Done__ }
// ...
type Operating_Controller : { __Inactive__, Priority, Init, PriorityPrepares, Secondary,
     SecondaryPrepares }
var PoliceInterrupt_police_In_Controller : boolean = false
// ...
ctrl var main_region_Controller : Main_region_Controller = __Inactive__
ctrl var operating_Controller : Operating_Controller = __Inactive__
var SecondaryTimeout2_Controller : integer = 0

trans {
    // ...
    choice {
        assume (main_region_Controller == Interrupted);
        // ...
        choice {
            assume (operating_Controller == Priority);
            SecondaryTimeout2_Controller := 0;
            PriorityControl_toggle_Out_Controller := true;
            SecondaryControl_toggle_Out_Controller := true;
            // ...
        }
    } or {
        // ...
    }
    PoliceInterrupt_police_In_Controller := false;
}
init {
    SecondaryTimeout2_Controller := 2 * 1000;
    main_region_Controller := __Inactive__;
    // ...
    PriorityPolice_police_Out_Controller := false;
    choice {
        assume (operating_Controller == __Inactive__);
        operating_Controller := Init;
    } or {
        assume !(operating_Controller == __Inactive__);
    }
    // ...
}
env {
    havoc PoliceInterrupt_police_In_Controller;
    PriorityPolice_police_Out_Controller := false;
    SecondaryControl_toggle_Out_Controller := false;
    PriorityControl_toggle_Out_Controller := false;
    SecondaryPolice_police_Out_Controller := false;
}
```

Listing 2.2: Gamma XSTS Language representing the traffic light controller statechart.

Chapter 3

Gamma Activity Language

The high-level nature of SysML activities means they are easy to use for modeling complicated behaviours of distributed system, but their complexity encumbers formal verification. As discussed above in Section 2.2.3, we can define a semantic-preserving mapping between activities and Petri nets (which have a formal semantics), however, that mapping is not complete as it disregards the data contained in tokens.

The Gamma Statechart Composition Framework (Section 2.3) implements a model transformation pipeline (see Figure 2.7) for the formal verification of collaborating statecharts, however, it does not include activity diagrams. In order to support the definition and verification of activites in the Gamma framework, I propose the Gamma Activity Language (GATL) (Chapter 3), and integrate it (Chapter 4) into the transformation pipeline.

3.1 Language Design

The purpose of GATL is twofold: it should support as many features from SysML activity diagrams as possible, while also having formal semantics. In order to make the transformation easier, the language supports only a constrained subset of the SysML feature set, however, this restriction does not hinder its expressiveness.

3.1.1 Supported SysML Feature Subset

Compared to SysML activity diagrams, GATL supports the following language constructs:

- *control* and *data* flows;
- *Initial* and *activity final* nodes;
- *Decision* and *merge* nodes - without probability;
- *Fork* and *join* nodes;
- *Action* nodes - which can contain inner activities (Call behaviour in SysML);
- *Pins* on action nodes;
- *Action* nodes - action nodes may contain specific internal actions[1] that can calculate values, and send signals through specific ports.

[1] Written in the Gamma Action Language

3.2 Formal Definition

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 GATL formalism.

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

- $N = N_{IR} \bigcup N_{LAS}$ is a set of *Nodes*, where N_{IR} contains the *immediate-repeat* nodes and N_{LAS} contains the *load-and-sand* nodes (see Section 2.2.3). $N_{Action} \in N_{LAS}$ is a special set of nodes, which are the *Action* nodes;
- $P \subseteq P_{In} \bigcup P_{Out}$ is a set of two types of pins, where $P_{In} : N_{Action} \rightarrow \{p_1, \dots, p_n\}$ and $P_{Out} : N_{Action} \rightarrow \{p_1, \dots, p_n\}$ are the set of *InputPins* and *OutputPins*, respectively, with domains $\{d_1, \dots, d_n\} \subseteq D_{GA}$;
- $F \subseteq F_C \bigcup F_D$, where $F_C = \{f_{C_1}, \dots, f_{C_n}\}$ and $F_D = \{f_{D_1}, \dots, f_{D_n}\}$ 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;
- $D_{GA} = \{d_1, d_2, \dots, d_n\}$ is a set of value domains;
- $F_{Action} \subseteq d_1 \times d_2 \times \cdots \times d_n$ is a function mapping an *action* nodes input pin values to it's output pin values. ▪

Informally, Gamma Activities are composed of *nodes* and flows in between them. A given *Action* node may have any number of *pins* with domain $d \in D$, for which there must be one and only one flow connected to the node. Nodes $n \in N_{Action}$ may also contain internal activities denoted GA_n.

N_{IR} contains the nodes *decision/merge*, while N_{LAS} contains the nodes *fork/join*, *initial/final* and *action* nodes. For the most part, these elements have a similar behaviour as their SysML counterparts; 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. Upon token transfer, the data inside the pins are transferred instantly, without any intermediate state in between.

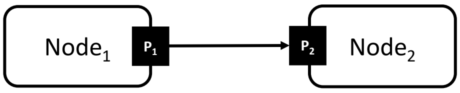


Figure 3.1: An example SysML data flow

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

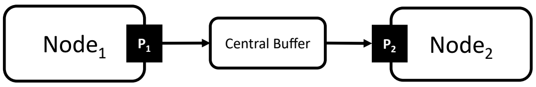


Figure 3.2: An example SysML data flow with central buffer

3.2.1 Behaviour Formalism

The behaviour of GATL is defined using the XSTS (Section 2.4) formalism:

Definition 4 (Gamma Activity Behaviour). $XSTS_{GA} = (D, V, V_C, IV, Tr, In, En)$, where:

- $D = D_{GA} \bigcup d_{NodeState} \bigcup d_{FlowState}$, where D_{GA} is the domain in GA, $d_{NodeState} = \{Idle, Running, Done\}$ and $d_{FlowState} = \{Empty, Full\}$ are the node and flow state domains representing their internal state. Node state *Idle* means a node is ready for a token, *Running* means it is currently executing and *Done* means it is done (but still contains a token). Flow states *Empty* and *Full* represent whether the flow contains a token;
- $V = V_{NS} \bigcup V_{FS} \bigcup V_{NV} \bigcup V_{FV} \bigcup V_{PV}$, where $V_{NS} = \{v_1, v_2, \ldots, v_n\}$ is the variable representing the *state* of the nodes with domain $d_{NodeState}$, $V_{FS} = \{v_1, v_2, \ldots, v_n\}$ is the variable representing the *state* of the flows with domain $d_{FlowState}$ and V_{NV}, V_{FV}, V_{PV} are the variables representing the values contained inside *flows*, *nodes* and *pins* respectively;
- IV sets all variables to their default value;
- $V_C = V_{NS} \bigcup V_{FS}$ are the *control variables*[2];
- $Tr = T$, where T is the single transition defined down below;
- In sets all values of variables associated with *InitialNodes* to *Running*;
- $En = \emptyset$ is the empty environment transition, as simple activities do not have environments. ▪

We also define the following helper functions:

- $S_N : N \to V_{NS}$ is a function that returns the *state variable* of a node;
- $S_F : N \to V_{FS}$ is a function that returns the *state variable* of a flow;
- $V_{N_{In}} : N \times F \to V_{NV}$ is a function that returns the *input value variable* of a node regarding a specific connected flow;
- $V_N : N \to V_{NV}$ is a function that returns the *value variable* of a node;
- $V_F : F \to V_{FV}$ is a function that returns the *value variable* of a flow;
- $PV_N : N \times P \to V_{PV}$ is a function returning the *value variable* of a pin inside a node.

[2] We don't have to set the value variables as control variables, because they only change when the given flow's or node's state variable changes

Informally, the behaviour of Gamma Activities is determined by the nodes' *state* (*Idle*, *Running*, *Done*), the nodes' and their pins' values, the flows' *state* (*Empty*, *Full*) and their values. For example, given an action node n and one of its pins p_1, $PV_N(n, p_1)$ would give us the exact value that p_1 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.

This definition gives us the structure to define the exact behaviour of a Gamma Activity. In the following, I incrementally build up the parts of transition T, and then propose an algorithm constructing the XSTS.

Let us define a shorthand function for finding a node's input variable associated with a given flow:

$$V_{N_{In}}(n, f) = \begin{cases} PV_N(n, \Phi(f)) & \text{if } n \in N_{Action} \\ V_N(n), & \text{otherwise} \end{cases}$$

And a shorthand function for finding a node's output variable associated with a given flow:

$$V_{N_{Out}}(n, f) = \begin{cases} PV_N(n, \phi(f)) & \text{if } n \in N_{Action} \\ V_N(n), & \text{otherwise} \end{cases}$$

These notations will help us easily transfer the data to and from the node value variables, as they hide the difference between the pin variables and the node variables.

Next, I define various functions that return operations, which will be used to construct the transition T. In the following, I use the following notations for XSTS operations:

- *sequences* are represented as operations after one another, or *sequence*(*op*);
- *parallels* are represented as *parallel*(*op*), where *op* is a set of operations;
- *assumptions* are represented as *assume*(*e*), where *e* is an expression;
- *assignments* are represented as *assign*(v, *value*), where v and *value* are a *variable* and a *value*, respectively, with the same *domain*.

Definition 5 (Flow Transition Operations). Let us denote the source/target node of flow f as $\theta(f)$ and $\Theta(f)$ respectively. $O_{FlowIn}(f)$ is a function returning a *sequential* operation, which takes a token from a flow to a node:

$$\begin{aligned} &\mathit{assume}(S_F(f) = \mathit{Full}) \\ &\mathit{assume}(S_N(\Theta(f)) = \mathit{Idle}) \\ &\mathit{assign}(S_F(f), \mathit{Empty}) \\ &\mathit{assign}(S_N(\Theta(f)), \mathit{Running}) \\ &\mathit{assign}(S_N(V_{N_{In}}(\Theta(f), f)), V_F(f)) \end{aligned}$$

Informally, the returned operation checks that the node is in *Idle* state and the flow is in *Full* state. Next, it transfers the token by changing their states. Let $O_{FlowOut}(f)$ be the function returning a *sequential* operation, which transfers a token from a node to a flow:

$$\begin{aligned}
&\mathit{assume}(G(f) = \mathit{true})\\
&\mathit{assume}(S_F(f) = \mathit{Empty})\\
&\mathit{assume}(S_N(\theta(f)) = \mathit{Done})\\
&\mathit{assign}(S_F(f), \mathit{Full})\\
&\mathit{assign}(S_N(\theta(f)), \mathit{Idle})\\
&\mathit{assign}(V_F(f), S_N(V_{N_{Out}}(\theta(f), f)))
\end{aligned}$$

The returned operation checks that the node is *Done* and the flow is *Empty* and *enabled*, in which case it transfers the token by changing their states. Note that both of the operations transfer the value of the node to the flow, or the value of the flow to the node using the previously defined helper notations. ▪

Definition 6 (Node Token In/Out Operations). Given the different behaviours of *IR* and *LAS* nodes, we must construct different operations for their in and out transitions. $O_{IRNodeIn}(n)$ is the function returning a *choice* operation that represents the "token intake" of an *IR* node:

$$O_{IRNodeIn}(n) = \mathit{choice}\left(\bigcup_{f \in \delta(n)} O_{FlowIn}(f)\right)$$

And $O_{IRNodeOut}(n)$ is the function returning a *choice* operation that represents the "token output" of an *IR* node:

$$O_{IRNodeOut}(n) = \mathit{choice}\left(\bigcup_{f \in \Delta(n)} O_{FlowOut}(f)\right)$$

On a similar note, the functions constructing the *LAS* node "token intake" and „token output" operations are the following:

$$O_{LASNodeIn}(n) = \mathit{parallel}\left(\bigcup_{f \in \delta(n)} O_{FlowIn}(f)\right)$$

$$O_{LASNodeOut}(n) = \mathit{parallel}\left(\bigcup_{f \in \Delta(n)} O_{FlowOut}(f)\right)$$

For the sake of simplicity, I define the following functions to merge all the node in/out operations:

$$\begin{aligned} O_{LASNodeIO}(n) &= \mathit{choice}\left(O_{LASNodeIn}(n) \bigcup O_{LASNodeOut}(n)\right) \\ O_{IRNodeIO}(n) &= \mathit{choice}\left(O_{IRNodeIn}(n) \bigcup O_{IRNodeOut}(n)\right) \\ O_{NodeIO}(n) &= \begin{cases} O_{IRNodeIO}(n) & \text{if } n \in N_{IR} \\ O_{LASNodeIO}(n), & \text{otherwise} \end{cases} \end{aligned}$$

In which case, $O_{NodeIO}(n)$ returns an operation transferring the tokens in and out of the specified node.

Informally, these operations realise the semantics of the *LAS* and *IR* nodes, by either putting the *flow in* and *flow out* operations inside a *parallel* or a *choice*, respectively[3].

Definition 7 (Node Run Operations). Node transition operations take the given node from *Running* state to *Done* state, while also executing the underlying operation. Generally, $O_{NodeRun}(n)$ function is defined as:

$$\begin{aligned} &\mathit{assume}(S_N(n), \mathit{Running}) \\ &\mathit{assign}(S_N(n), \mathit{Done}) \end{aligned}$$

However, some nodes have special behaviours. If the node is an *Action* node, it can contain either a *Gamma Action* expression, or a composite *Activity* (see Section 3.3 for more information).

- If the node is an *Action* node, the contained *Gamma action* is mapped to an operation $O_{Action}(n)$, and added to $O_{NodeRun}(n)$. Let us call this function $O'_{NodeRun}(n)$
- If the node is an *Action* node, the contained activity first has to be started, and the state of the node can change only when the contained activity is done. Let us call this function as $O''_{NodeRun}(n)$.

$O'_{Run}(n)$ function is defined as:

$$\begin{aligned} &\mathit{assume}(S_N(n), \mathit{Running}) \\ &O_{Action}(n) \\ &\mathit{assign}(S_N(n), \mathit{Done}) \end{aligned}$$

Let $In(n)$ and $Fin(n)$ be the sets of *initial* and *final* nodes in GA_n.

[3] Parallel operations execute *all* contained operations at the same time, while choices chose one at random

$$O_{Start}(n) = sequence\left(\bigcup_{n_{In} \in In(n)} \Big(assume(S_N(n_{In}) = Idle) \bigcup assign(S_N(n_{In}), Running)\Big)\right)$$

$$O_{Finish}(n) = sequence\left(\bigcup_{n_{Fin} \in Fin(n)} assume(S_N(n_{Fin}) = Done) \bigcup assign(S_N(n), Done)\right)$$

$$O_{ActivityRun}(n) = choice\Big(O_{Start}(n) \bigcup O_{Finish}(n)\Big)$$

$$O''_{NodeRun}(n) = sequence\Big(assume(S_N(n), Running) \bigcup O_{ActivityRun}(n)\Big) \qquad \blacksquare$$

Informally, the returned operation checks if the node is in state *Running*, and then chooses: it either sets the contained activity's initial nodes *Running* if they are currently *Idle*, or sets the node's state *Done* if the contained activity's final nodes are *Done*. Thus, the node is considered *Done* only when the contained activity is also done.

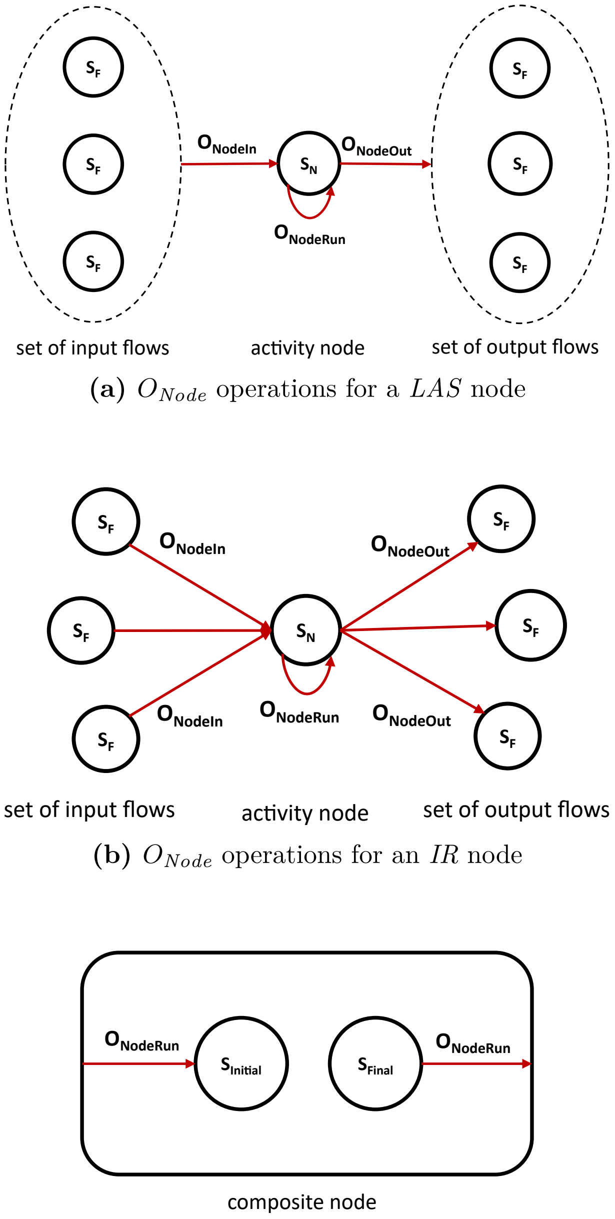
Definition 8 (Composite Node Operation). Putting all of the work together, we can define a function returning the operation representing all behaviour of a given node:

$$O_{Node}(n) = choice\Big(O_{NodeIO}(n) \bigcup O_{NodeRun}(n)\Big) \qquad \blacksquare$$

Informally, this is a choice operation, that either moves the tokens in/out, or executes the internal action of the node. The various operations defined in the previous sections can be seen in Figure 3.3.

As the final step, we construct the transition T, by applying the $O_{Node}(n)$ function on each node:

$$T = \bigcup_{n \in N} O_{Node}(n)$$



(a) O_{Node} operations for a LAS node

(b) O_{Node} operations for an IR node

(c) $O_{NodeRun}$ operations for a *composite* action node

Figure 3.3: An illustration of the state change operations

3.3 Language Gramar

Similarly to 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 grammar to support the representation of models in a textual way. This grammar definition incorporates elements from the SysMLv2 [12] language design, while also fitting into the already existing language family of Gamma (Section 2.3). In the following, I define the *metamodel* and the *grammar* created to realise the formalism specified in Section 3.2.

3.3.1 Metamodel

Due to the complexity of the final metamodel of the language, I have split it into multiple parts for easier understanding: Pins (3.3.1), Flows (3.3.1), Activity Nodes (3.3.1), Structure (3.3.1), Composite Activities (3.3.1) and Data-, Pin-reference (3.3.1-3.3.1).

Pins

Pins are categorised into two sets, *InputPins* and *OutputPins*. All pins have a *Type*[4] associated with them, which define their domain. Figure 3.4 shows this section of the metamodel.

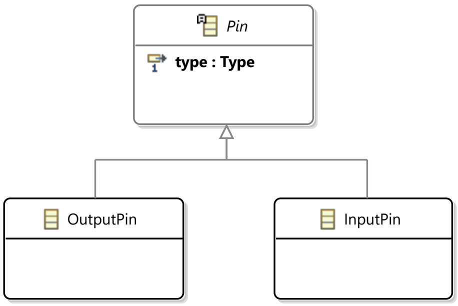


Figure 3.4: The pin metamodel diagram

Flows

Flows have two kinds, *ControlFlows* and *DataFlows*. All flows have a guard of type *Expression*[5], which can evaluate to a *boolean* value; if it evaluates to *True*, the flow is considered *enabled*. Figure 3.5 depicts the relation of flows in the metamodel.

Control Flow *ControlFlows* have a reference to their source and target nodes.

Data Flow *DataFlows* contain a *DataSourceReference* and a *DataTargetReference*, which can either be a *Pin*, or a *DataNode*. A data token may contain a value of any kind, but that token can travel only to and from data *sources* and *targets*. This will be explained in more detail below.

[4] Types come from the Gamma Expression Language
[5] Written in the Gamma Expression Language

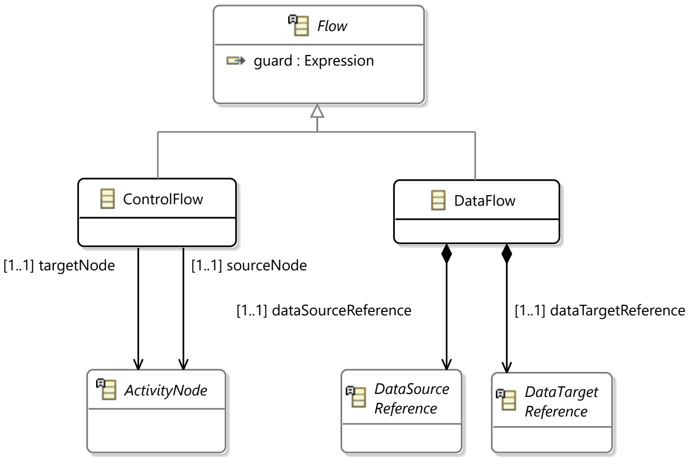


Figure 3.5: The Flows structure

Activity Nodes

In the following, I will talk about the different kinds of *ActivityNodes* and their special meanings. The metamodel described in this section can be seen in Figure 3.6.

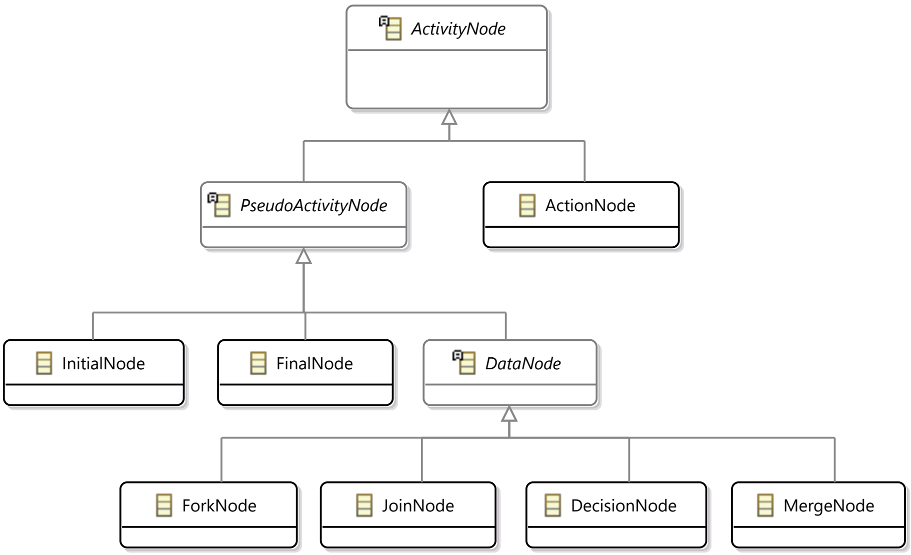


Figure 3.6: The Activity Node structure

Action Node *ActionNodes* represent a specific action the activity may execute. This action can be defined in multiple ways, see Section 3.3.1 for more detail.

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* represent the entry point of the activity.

Final Node A node representing the final node of the activity.

Data Node *DataNodes* encapsulate the meaning of *data* inside activities.

Fork Node *ForkNodes* are used to model parallelism by creating one token on each of its output flows when executed.

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

Decision Node *DecisionNodes* create branches across multiple output flows. An input flow's token is removed, and transfered out to a single output flow - depending on which of the output flows are *enabled*[6].

Merge Node *MergeNodes* forward all incoming tokens as soon as they arrive, one by one. They are used to *merge* different flow paths (created using *decisions*).

Root Structure

The root structure defines where the elements of the activity reside in the model. The metamodel described in this section is depicted in Figure 3.7.

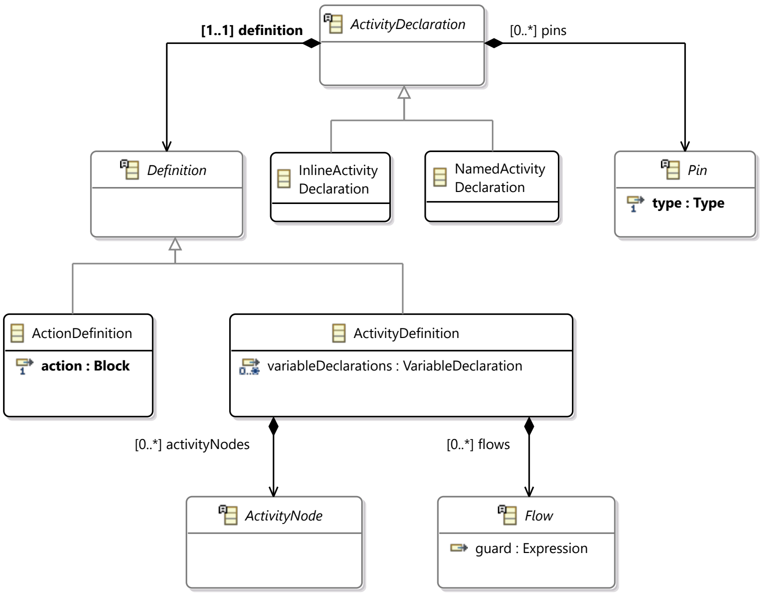


Figure 3.7: The root structure of the language

Activity Declaration All elements inside an activity are contained in a root *ActivityDeclaration* element. It contains *Pins* needed for value passing and a *Definition*. Declarations can be *InlineActivityDeclarations*, which means they are declared in an other declaration, or *NamedActivityDeclaration*, which is a standalone activity declaration - *NamedActivityDeclaration* can be referenced from other parts of the model, while *InlineActivityDeclaration* cannot.

[6] In the case when multiple flows are enabled one is chosen nondeterministically

Definition Definitions define how the activity is described; using activity nodes, or by the Gamma Action Language. *ActionDefinition* contains a single *Block*[7], which is executed as is when the activity is executed[8]. *ActivityDefinitions* contain *ActivityNodes* and *Flows*, and are used to *compose* activities.

Composing Activities

ActionNodes may contain a single *ActivityDeclarationReference* to an *ActivityDeclaration*[9], in which case the execution of said node will also include the execution of the underlying *ActivityDeclaration*. This construction gives us the power to pre-declare, or inline specific activities in the model; and use any definition defined above (Section 3.3.1) for them. Figure 3.8 shows this part of the metamodel.

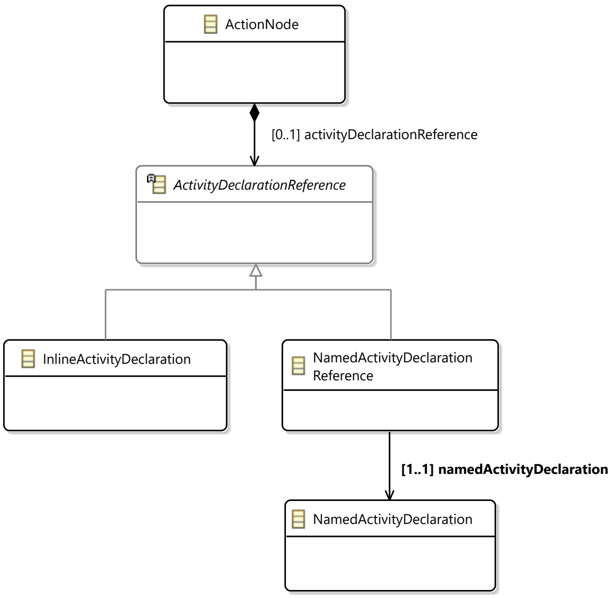

ActionNode
[0..1] activityDeclarationReference
ActivityDeclarationReference
InlineActivityDeclaration
NamedActivityDeclaration Reference
[1..1] namedActivityDeclaration
NamedActivityDeclaration

Figure 3.8: The action node containment hierarchy

Data Source Reference and Data Target Reference

In order to correctly set a data flow's data source and data target, the model has to store where the pins are referenced. A given *InputPin* can be considered a *DataTarget* from outside of the associated *Activity*, however, it is a *DataSource* from inside the *Defintion*. *DataNodes* can be considered both data sources and data targets. See Figure 3.9 for the corresponding metamodel part.

Example of inside-input/outside-output pin Figure 3.10 shows an example for the multidirectional effect of pins. From the perspective of the flow $P_1 \rightarrow P_2$ the pin P_1 is a *DataSource* and the pin P_2 is a *DataTarget*. However, from the perspective of the flow $P_2 \rightarrow P_3$, the pin P_2 is a *DataSource* - because the latter flow is inside the composite activity.

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

[8] This means, that upon execution the given activity is executed atomically; it will not be interlaced with other XSTS transitions. See Section 4.1.1.

[9] If the node does not have any, it is considered a *simple* node, without any implementation

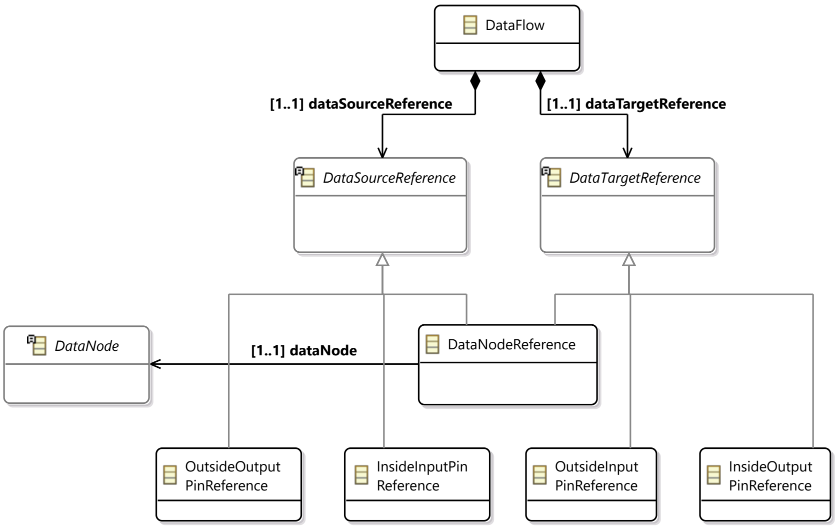

DataFlow
[1..1] dataSourceReference
[1..1] dataTargetReference
DataSourceReference
DataTargetReference
DataNode
[1..1] dataNode
DataNodeReference
OutsideOutput PinReference
InsideInputPin Reference
OutsideInput PinReference
InsideOutput PinReference

Figure 3.9: The Data Source-Target reference structure

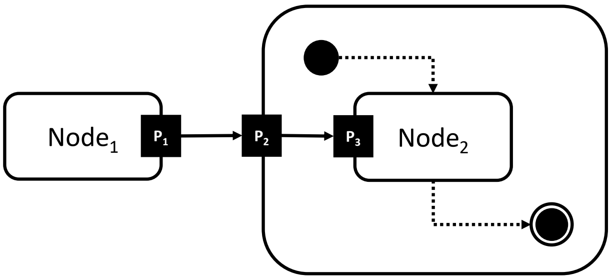


Figure 3.10: The Data node reference structure

Pin Reference

Pin references are used to define a rigid pin-reference structure in the model. *InputPinReferences* have a reference to a specific *InputPin*, *OutputPinReferences* have a reference to a specific *OutputPin*. *InsidePinReferences* and *OutsidePinReferences* are used to define the direction in which the reference sees the given pin; inside references see it from the inside, outside references see it from the outside. The *OutsidePinReference* must also have a reference to the specific *ActionNode* the pin is associated with[10]. See Figure 3.11 for the corresponding metamodel part.

3.3.2 Concrete Syntax

In order to make it easier to test and visualise activities, I defined a grammar for the metamodel in Xtext. The SysMLv2 language served as the main inspiration for the language, however, much of the *syntactic sugars* have been omitted for the sake of simplicity. Listing 3.1 is the activity Figure 2.3 example written in GATL. For the exact grammar definition, please see Section A.2.

[10] Note that the inside pin reference does not have a node reference, because the node must be the activity that contains the pin

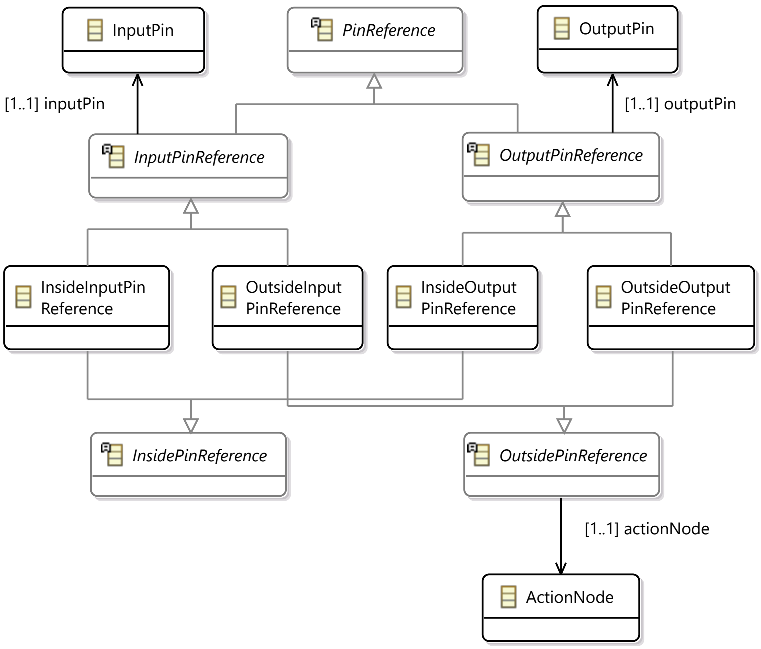


Figure 3.11: The Pin reference structure

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

    initial Initial
    merge Merge
    fork Fork

    action Read1 : activity (out pr1 : integer)
    action Compile1 : activity (in pc1 : integer)
    action Read2 : activity (out pr2 : integer)
    action Compile2 : activity (in pc2 : integer)

    join Join
    action Link
    decision Decision
    action Edit
    final Final

    control flow from Initial to Merge
    // ..
    control flow from Fork to Read2
    data flow from Read1.pr1 to Compile1.pc1
    data flow from Read2.pr2 to Compile2.pc2
    // ..
    control flow from Decision to Edit [errors]
    control flow from Edit to Merge
    control flow from Decision to Final [!errors]
}
```

Listing 3.1: Gamma Activity Language representation of the compilation activity.

Chapter 4

Integrating the Activity Language Into Gamma

In the previous chapter, I introduced the Gamma Activity Language - the important parts of its metamodel along with the semantics of the model elements and their textual syntax. However, in order to formally verify activity models, the language has to be integrated into the Gamma transformation pipeline (see Figure 2.7). For this integration, I start by defining *how* the semantics of activities alongside statecharts in Section 4.1, after which I define the integration semantics in Section 4.2. Finally, I overview the implementation details in Section 4.3.

4.1 Activities Alongside Statecharts

In this section, I present the various questions that arose when I integrated GATL into Gamma statecharts, and present my solutions and reasoning for said problems.

4.1.1 Calling Activities

In order to create the connection between statecharts and activities, the user needs an interface to do so. There are many ways to call activities from statecharts:

Transition Actions The first solution that came in mind was calling activities from transition actions. At first glance, this makes the most sense, however, as activities are inherently parallel in nature, the resulting implementation would have to *flatten* the activity model - the action cannot contain loops, and has to be atomic. Because of time constraints, this feature was not chosen.

Do Behaviours In SysML, states may have do behaviours. This means, that the *behaviour* (activity in our case) is *ran* while the state machine is inside the given state, and *halted* when the state is left. However, the semantics are not clear.

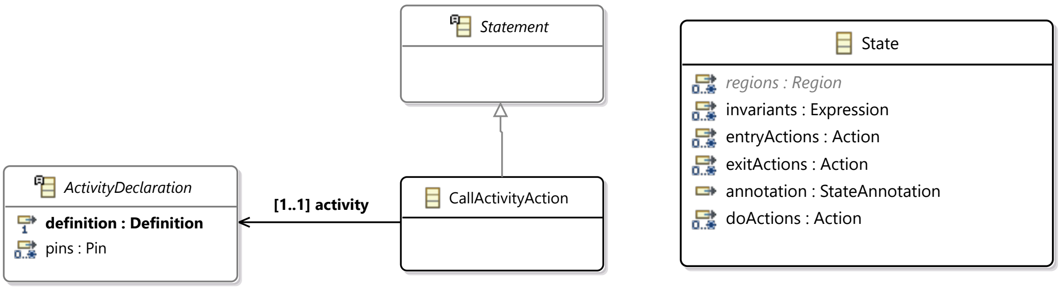
Semantically, the halting of the activity can happen in two ways:

- The state *signals* the activity, that is should end, and waits until it halts - creating a synchronisation step.

- The activity checks if it should run - whether the state machine is in the given state.

Another open question is what happens, when a state has a loop transition into itself? Is the activity stopped, asked to stop, or a whole new activity is created?

For simplicities sake, I chose do behaviours with option two: check each time when the activity would do something, whether it is *enabled* to run, or not. Additionally, when the containing state becomes active, the activity nodes are reset to *Idle* state, and the initial nodes are started. Figure Figure 4.1 shows the added elements to the Gamma metamodel.



(a) Call activity action

(b) Do action on states

Figure 4.1: Extensions to the Gamma metamodel

4.1.2 Activities Defining Components

In SysML, components' behaviour may be defined directly by activities. However in Gamma, all components use reactive semantics, which would have been difficult to implement. Thus, activities may only appear inside statecharts for now.

Although this constrains us to only use statecharts directly, we can still utilise the reactive nature of components, by adding a special activity node, that listens for such events.

4.1.3 Trigger Node

TriggerNodes extend the activity semantics, by adding an additional assumption operation inside the $O_{NodeRun}$ operation function, checking if the given event has been received. The following example shows a trigger node, that will be executed only when it has a token from the initial node, and the *start* event is received from the *Control* port.

```
statechart Statechart [
    port connection : requires Control
] {
    // definitions ...

    activity DoActivity {
        initial Initial
        trigger AcceptEvent when connection.start
        final Final

        control flow from Initial to AcceptEvent
        control flow from AcceptEvent to Final
    }
}
```

4.2 Integration Semantics

As presented in the transformation pipeline (Figure 2.7), Gamma transforms the *statecharts* and *components* into XSTS. In the following, I show how to merge XSTS created from *components* with the called activities.

The unified transformation is done in three steps. First, in Section 4.2.1 we preprocess the composite model, by creating activity *instances* and adding the initialisation action to the calling states.. Next, in Section 4.2.2 we transform the components - using the Gamma transformation pipeline - and then transform the called - now unique - activities. The resulting XSTS models are merged, resulting in the final XSTS. This XSTS model then can be forwarded to the model checkers.

4.2.1 Preprocess Components

As the first step, we find all states that contain a *call activity* do action - denoted as S. For each $s \in S$, we create an instantiation for the called activity: S_{Act}, and add an initialisation action to the state's *entry* actions. This step ensures, that two states calling the same activity won't have conflicting variables in the resulting XSTS, and the called activity is cleaned before it is started. Figure 4.2 shows the extensions added to the Gamma metamodel.

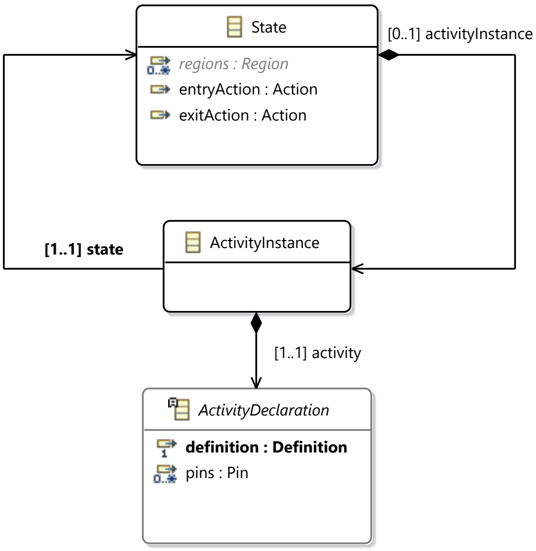


Figure 4.2: Activity instance extension for the Gamma metamodel

4.2.2 Transform Components and Activities

As the next step, we transform the Gamma components, denoted as $XSTS_{Comp}$, and the activity instances, denoted as $XSTS_{S_{Act}}$. In order to prevent the activity from running when the state is not active, an additional *assume* operation is added to the $O_{Node}(n)$ function, checking whether the associated state is active or not. Finally, the resulting XSTS models are merged.

$$XSTS = XSTS_{Comp} \bigcup \left(\bigcup_{s \in S} XSTS_{s_{Act}} \right)$$

4.3 Implementation Remarks

In order to implement the language, I had to immerse myself in multiple technologies. Gamma is implemented as an Eclipse plugin[1], and uses multiple frameworks: Ecore Modeling Foundation[2] to create the metamodels, Xtext[3] to define the language grammars, and Viatra[4] to transform the models.

During the implementation, I added a new *model* project and a new *language* project. I then had to extend the preexisting Gamma Statechart Language, and the transformation pipeline.

The contributions include 12, 383 line additions, 2, 184 line deletions in 38 commits. There is currently an open pull request to the main GitHub repo[5].

These changes implemented are visualised in Figure 4.3: added parts are outlined with green, while edited parts are outlined with orange.

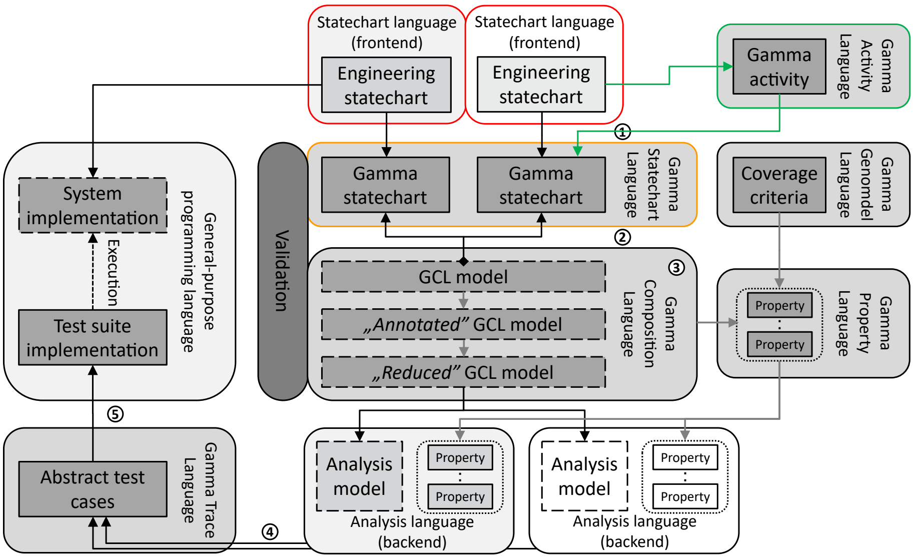


Figure 4.3: Overview of my modification to the Gamma transformation pipeline.

[1] https://www.eclipse.org/downloads/

[2] https://www.eclipse.org/modeling/emf/

[3] https://www.eclipse.org/Xtext/

[4] https://www.eclipse.org/viatra/

[5] https://github.com/ftsrg/gamma

Chapter 5

Evaluation

In this chapter, I present three experiments created to showcase the usability and verifiability of the Gamma Activity Language.

5.1 Compilation

The first experiment involves the running example introduced in Figure 2.3. The goal of this experiment is to evaluate a simple activity's formal verification.

5.1.1 Modeling

The GATL representation of the model has already been presented in Listing 3.1, however this some modifications have to be implemented in order to run the verification.

- Activities have to be wrapped inside statecharts.
- Activities can not be referenced from GPL, only statechart constructs.

For this reason, I wrapped the activity inside a statechart, and added an *activityDone* variable, that turns true when before the activity reaches the final state.

```
statechart CompilationWrapper {
    var activityDone : boolean := false
    region Main {
        initial MainEntry
        state Wrapper {
            do / call CompilationProcess;
        }
    }
    transition from MainEntry to Wrapper
    activity CompilationProcess {
        // ...
        action Done : activity [language=action] {
            activityDone := true;
        }
        final Final
        // ...
        control flow from Decision to Done [!errors]
        control flow from Done to Final
    }
}
```

I then added the statechart to *cascade* component, and defined the property I wished to check for.

```
cascade Compilation {
    component CompilationWrapper : CompilationWrapper
}
```

```
component Compilation

// Cam the variable activityDone ever be true?
E F [ { variable CompilationWrapper.activityDone } ]
```

Finally, I ran the verification using the *Theta* model checker. The result shows, that the variable *activityDone* can indeed become true.

5.1.2 Results and Conclusion

The following trace shows the steps the model checker took, in order to reach the desired state. The first step shows the initial values of the system, and the following steps show how the values changed after the component was scheduled. As you can see, the second step is repeated 17 times before the variable became true - and these steps were the same. As there are no XSTS transition from the statechart, the only transitions that can be fired come from the activity. The token in the activity goes through 8 nodes, 8 flows to reach node *Done*, after which one additional step is needed to execute its action: $8+8+1=17$.

```
step {
    act {
        reset
    }
    assert {
        CompilationWrapper.Wrapper
        CompilationWrapper.activityDone = false
    }
}
step {
    act {
        schedule component
    }
    assert {
        CompilationWrapper.Wrapper
        CompilationWrapper.activityDone = false
    }
}
// 16 more
step {
    act {
        schedule component
    }
    assert {
        CompilationWrapper.Wrapper
        CompilationWrapper.activityDone = true
    }
}
```

The reason for the repetition, is the lack of trace information provided by the activity; the XSTS transitions are executed as expected, however, the trace mechanism of Gamma can not display these changes.

This case study proved, that the underlying XSTS formalism does indeed work as expected, however, there is still more work to be done, to let the user choose activity nodes as reachability properties, and to show activity variables in the resulting trace (see Chapter 6).

5.2 Complicated System

5.3 Space-mission

This section introduces an example model from the aerospace domain which we use as a case study to demonstrate the applicability of Gamma to describe SysML behavioral models. The example model was proposed by NASA in the context of the OpenMBEE[1] framework. The goal of OpenMBEE is to create a common model repository to facilitate tool integration, so the ability to handle models in the scope of this project can raise the relevance of any model analysis tool.

The Gamma models used in this section are a modification of a previous case study [8], in which the activities were mapped to statechart formalism. In this case study, I modified the models to use the GATL to represent the activities.

5.3.1 System Modeling

The SysML model describes how a satellite communicates with a ground station as well as the state of the battery of the satellite. The state-based behavior of the system can be seen in Figures 5.1 and 5.2, while the more complex activities are depicted in Figures 5.3 and 5.4

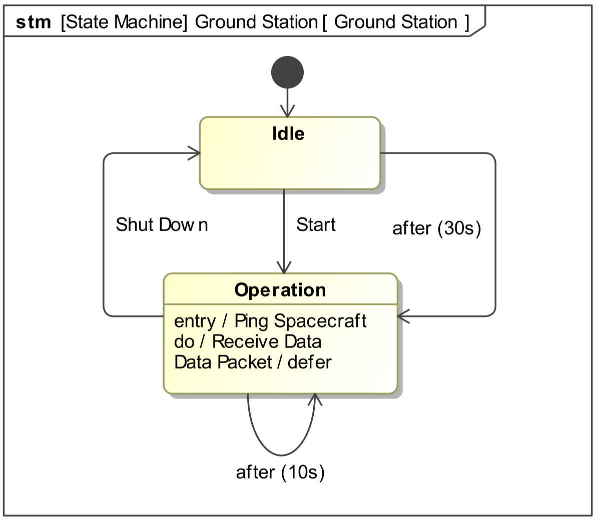


Figure 5.1: The state machine describing the behaviour of the ground station component

The challenge of the mapping came from some of the not supported modeling elements used in the model. The SysML activities use *duration constraints* to model how much time a given action takes to execute. The GATL language does not support such constraints, however, we can use *timeouts* and *trigger nodes* to insert waiting in before the given action can be executed. The other not supported, but used activity formalisms are interrupting

[1] https://www.openmbee.org/

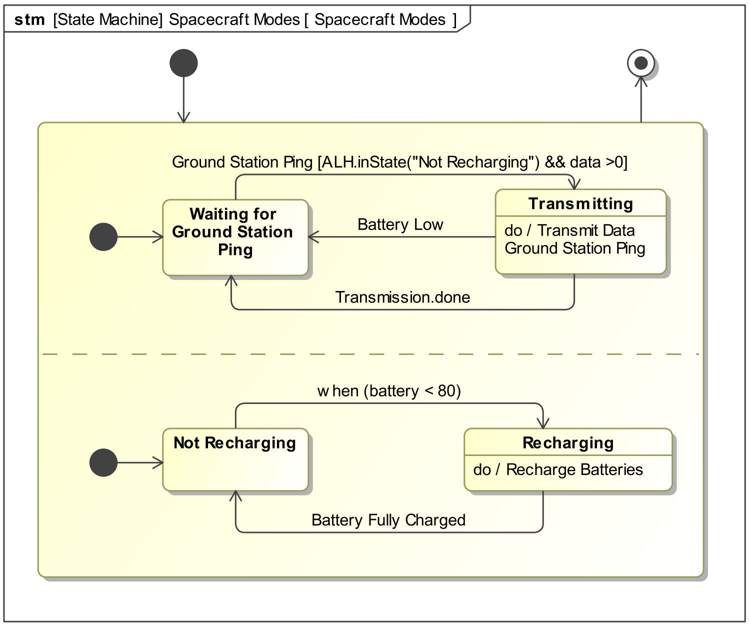


Figure 5.2: The state machine describing the behaviour of the spacecraft component

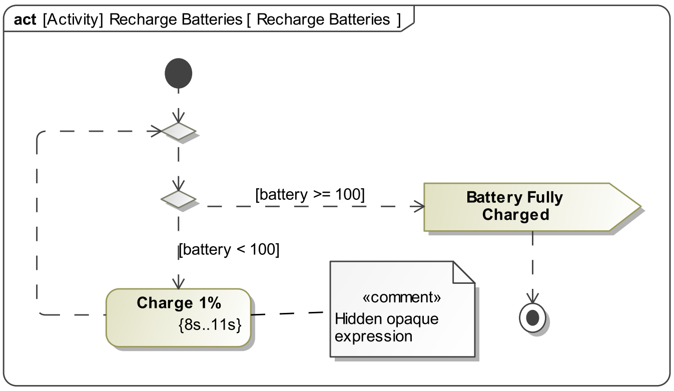


Figure 5.3: The activity diagram describing the battery recharge process of the spacecraft component

edges, and internal signals. I used additional boolean variables to model the behaviour of these modeling formalisms. You can find the created Gamma models in Section A.3

5.3.2 Results and Conclusion

To check the conformance of the Gamma model with the original SysML model, I ran a set of verifications on the resulting model, and all traces were consistent with the behaviour described by the SysML.

This case study showed, that real world SysML models can be mapped to the Gamma Activity Language, although, certain semantics (e.g., *interrupting edge*) have to be mapped to different formalisms. Nevertheless, this case study provides proof of the usability of the language in industrial contexts.

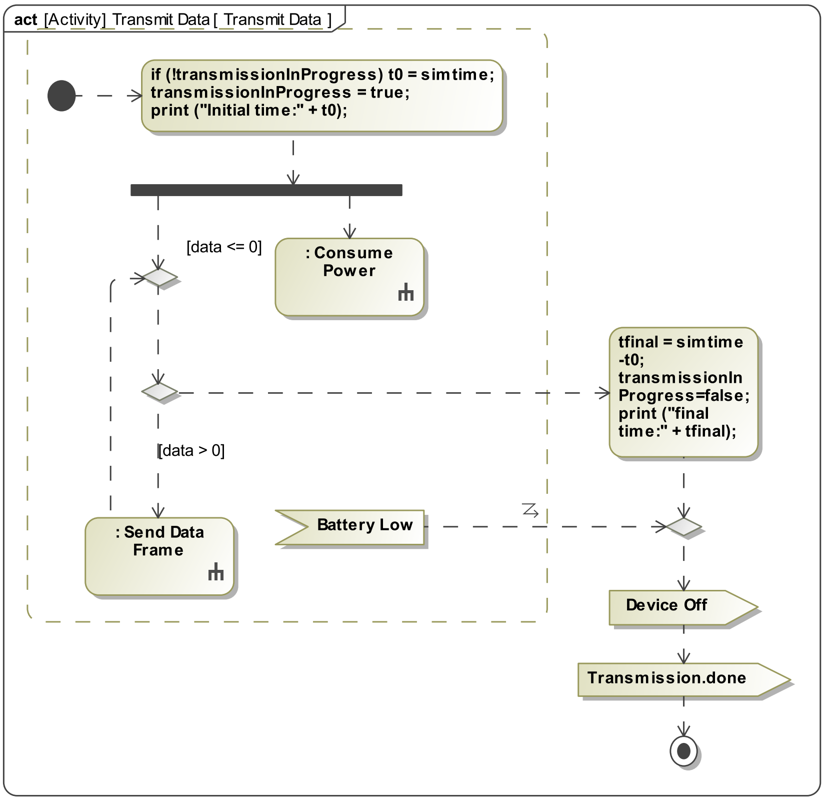

act [Activity] Transmit Data [Transmit Data]
if (!transmissionInProgress) t0 = simtime;
transmissionInProgress = true;
print ("Initial time:" + t0);
[data <= 0]
: Consume Power
tfinal = simtime -t0;
transmissionInProgress=false;
print ("final time:" + tfinal);
[data > 0]
: Send Data Frame
Battery Low
Device Off
Transmission.done

Figure 5.4: The activity diagram describing the data transmission process of the spacecraft component

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

Listings

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] Wolfgang Reisig. Petri nets and algebraic specifications. *Theoretical Computer Science*, 80(1):1–34, 1991. ISSN 0304-3975. DOI: `https://doi.org/10.1016/0304-3975(91)90203-E`. URL `https://www.sciencedirect.com/science/article/pii/030439759190203E`.

[26] 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 XSTS Language

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

Types

XSTS contains two default variable types, logical variables (*boolean*) and mathematical integers (*integer*). 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> }

type Color : { RED, GREEN, BLUE }
```

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>
```

The variable can only take a value of the specified 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).

Examples:

```
var a : integer
ctrl var b : boolean = false
var c : Color = RED
```

Operations

The behaviour of XSTS can be described using basic and composite operations. Basic operations include assignments, assumptions and havocs. In the following, you can see an example for each, where `<expr>` is an expression returning a value, and `<varname>` is the name of a variable.

```
assume <expr>

<varname> := <expr>

havoc <varname>
```

Composite operations are either non-deterministic choices, sequences or parallels. Nondeterministic choices have the following syntax, where `<operation>` are arbitrary basic or composite operations:

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

Sequences have the following syntax:

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

And parallels have the following syntax:

```
parallel {
    <operation>
    <operation>
}
```

Transitions

Each transition is a single operation (basic or composite). We distinguish between three sets of transitions, *Tran*, *Init* and *Env*. 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.2 Gamma Activity Language

Language elements

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

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.

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 `guard`s 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.2.

```
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 composite activity can be easily created using inline activity definition:

```
activity Example {
    initial Init1
    final Final1

    action InlineActivityExample : activity {
        initial Init2
        final Final2

        control flow from Init2 to Final2
    }

    control flow from Init1 to InlineActivityExample
    control flow from InlineActivityExample to Final1
}
```

```
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
}
```

Listing A.2.1: Basic adder example.

Example

A simple example activity can be seen in Listing A.2.1. This example shows two read actions, that return a random value, which are added together and logged to the console. The addition is defined using a *NamedActivitDeclaraionReference*.

```
package mission

import "Interface/Interfaces"
import "Groundstation/GroundStation"
import "Spacecraft/Spacecraft"

sync Mission [
port _control : requires StationControl
] {
    component station : GroundStation
    component satellite : Spacecraft
    bind _control -> station._control
    channel [ satellite.connection ] -o)- [ station.connection ]
}
```

```
import "Mission.gcd"
component Mission

E F [ { variable satellite.batteryVariable = 14 } ]
```

A.3 Spacecraft Model

```
import "Interface/Interfaces.gcd"
import "Mission.gcd"

analysis {
    component : Mission
    language : Theta
    constraint : {
        minimum-orchestrating-period : SCHEDULE_CONSTRAINT ms
        maximum-orchestrating-period : SCHEDULE_CONSTRAINT ms
    }
    optimize : false
    property-file : "Mission.gpd"
}

verification {
    language : Theta
    file : "Mission.xsts"
    property-file : "Mission.gpd"
    optimize : false
}
```

```
package interfaces

interface DataSource {
    out event _data
    in event ping
}

interface StationControl {
    out event start
    out event shutdown
}

const SCHEDULE_CONSTRAINT : integer := 1501
```

```
activity ReceiveData {
    initial Initial

    merge Merge

    trigger DataPacket when connection._data

    action ProcessData

    control flow from Initial to Merge
    control flow from Merge to DataPacket
    control flow from DataPacket to ProcessData
    control flow from ProcessData to Merge
}
```

```
activity RechargeBatteries {
    initial Initial

    merge Merge

    decision Decision

    action SetWait : activity [language=action] {
        set rechargeTimeout := 10s;
    }

    trigger Wait when timeout rechargeTimeout

    action Charge : activity [language=action] {
        batteryVariable := batteryVariable + 1;
    }

    action Full : activity [language=action] {
        batteryFullyCharged := true;
    }

    final Final

    control flow from Initial to Merge
    control flow from Merge to Decision
    control flow from Decision to SetWait [batteryVariable < 100]
    control flow from SetWait to Wait
    control flow from Wait to Charge
    control flow from Charge to Merge
    control flow from Decision to Full [batteryVariable >= 100]
    control flow from Full to Final
}
```

```
activity TransmitData {
    initial Initial

    fork Fork

    merge TransmitMerge
    decision TransmitDecision
    action SetTransmitWait : activity [language=action] {
        set transmitTimeout := 1s;
    }
    trigger TransmitWait when timeout transmitTimeout
    action TransmitData : activity [language=action] {
        _data := _data - 1024;
        raise connection._data;
    }

    merge ConsumeMerge
    action SetConsumeWait : activity [language=action] {
        set consumeTimeout := 1s;
    }
    trigger ConsumeWait when timeout consumeTimeout
    action ConsumeEnergy : activity [language=action] {
        batteryVariable := batteryVariable - 1;
    }
    decision ConsumeDecision

    merge Merge

    action SetDone : activity [language=action] {
        transmitionDone := true;
    }

    final Final

    control flow from Initial to Fork
    control flow from Fork to TransmitMerge
    control flow from TransmitMerge to TransmitDecision
    control flow from TransmitDecision to SetTransmitWait [_data > 0]
    control flow from SetTransmitWait to TransmitWait
    control flow from TransmitWait to TransmitData
    control flow from TransmitData to TransmitMerge
    control flow from TransmitDecision to Merge [_data <= 0]

    control flow from Fork to ConsumeMerge
    control flow from ConsumeMerge to SetConsumeWait
    control flow from SetConsumeWait to ConsumeWait
    control flow from ConsumeWait to ConsumeEnergy
    control flow from ConsumeEnergy to ConsumeDecision
    control flow from ConsumeDecision to ConsumeMerge [batteryVariable >= 40]
    control flow from ConsumeDecision to Merge [batteryVariable < 40]

    control flow from Merge to SetDone
    control flow from SetDone to Final
}
```

```
package spacecraft

import "Interface/Interfaces.gcd"

@RegionSchedule = bottom-up
statechart Spacecraft [
    port connection : provides DataSource
] {
    var batteryVariable : integer := 100
    var _data : integer := 100

    var batteryRecharging : boolean := false
    var batteryFullyCharged : boolean := false
    var transmitionDone : boolean := false

    timeout rechargeTimeout
    timeout consumeTimeout
    timeout transmitTimeout

    region Communication {
        initial CommunicationEntry
        state WaitingPing
        state Transmitting {
            do / call TransmitData;
        }
    }
    region Battery {
        initial BatteryEntry
        state NotRecharging
        state Recharging {
            do / call RechargeBatteries;
        }
    }

    transition from CommunicationEntry to WaitingPing
    transition from WaitingPing to Transmitting when connection.ping [batteryRecharging and _data > 0]
    transition from Transmitting to WaitingPing when timeout transmitTimeout [transmitionDone]
        / transmitionDone := false;

    transition from BatteryEntry to NotRecharging
    transition from NotRecharging to Recharging when timeout consumeTimeout [batteryVariable < 80]
        / batteryRecharging := true;
    transition from Recharging to NotRecharging when timeout rechargeTimeout [batteryFullyCharged]
        / batteryFullyCharged := false; batteryRecharging := false;

    activity TransmitData {
        // ...
    }

    activity RechargeBatteries {
        // ...
    }

}
```

```
package groundstation

import "Interface/Interfaces.gcd"

statechart GroundStation [
    port connection : requires DataSource
    port _control : requires StationControl
] {
    timeout pingTimeout
    timeout autoStart

    region Main {
        initial Entry
        state Idle {
            entry / set autoStart := 30s;
        }
        state Operation {
            do / call ReceiveData;
            entry / raise connection.ping; set pingTimeout := 10s;
        }
    }

    transition from Entry to Idle
    transition from Idle to Operation when _control.start
    transition from Idle to Operation when timeout autoStart
    transition from Operation to Operation when timeout pingTimeout
    transition from Operation to Idle when _control.shutdown

    activity ReceiveData {
        // ...
    }
}
```