Rheinland-Pfälzische Technische Universität
Kaiserslautern-Landau

MASTER THESIS

Model Translation for Security Analysis and Formal Verification of Embedded Systems

Author:
Mayur Dattatray Sawant
(Matriculation Number : [redacted])

Supervisors:
Prof. Dr. Klaus Schneider
M.Sc. Marvin Häuser

A thesis submitted in fulfillment of the requirements
for the degree of Master of Science in Commercial Vehicle Technology

in the

Embedded Systems Group

of the

Department of Computer Science

August 1, 2025

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema „Model Translation for Security Analysis and Formal Verification of Embedded Systems“ selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit — einschließlich Tabellen und Abbildungen —, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 01.08.2025

_____

Mayur Dattatray Sawant

Abstract

Safety-critical embedded systems must satisfy two equally stringent assurances: rigorous formal verification of their reactive behaviour and comprehensive security checking to safeguard confidential and authentic data. The synchronous design flow provided by Averest delivers machine-checked proofs for programs written in the Quartz language. At the same time, the SysML-based TTool contributes model-centric performance analysis, real-time checking, and automated verification of confidentiality and authenticity using ProVerif and UPPAAL. This thesis demonstrates that these complementary strengths can be combined through a semantics-preserving translation from Quartz to SysML/AVATAR.

The work proceeds in three stages. (i) A feasibility study maps Quartz constructs, logical instants, broadcast signals, guarded actions, onto equivalent SysML elements and identifies the few features that require controlled approximation. (ii) A prototype translator ingests a Quartz model already proved correct in Averest and emits an XML file that opens unmodified in TTool. (iii) An evaluation on representative benchmarks confirms that the translated models execute faithfully in TTool, inherit the original safety proofs that extend the assurance envelope beyond what either tool provides in isolation.

The resulting workflow, not only enables engineers to start with deterministic, synchronous specifications and obtain machine-checked safety proofs, but also extends the same design artefact with attacker models and cryptographic verification. This practical application of a unified path to safety and security assurance for reactive embedded systems holds great promise for the future.

Zusammenfassung

Sicherheitskritische Embedded-Systeme müssen zwei ebenso strenge Anforderungen erfüllen: eine rigorose formale Verifikation ihres reaktiven Verhaltens und eine umfassende Sicherheitsprüfung zum Schutz vertraulicher und authentischer Daten. Der synchrone Entwurfsablauf, den Averest bereitstellt, liefert maschinengeprüfte Beweise für in der Sprache Quartz geschriebene Programme. Gleichzeitig bietet das auf SysML basierende TTool modellzentrierte Leistungsanalysen, Echtzeit-Checks und automatisierte Verifikation von Vertraulichkeit und Authentizität mittels ProVerif und UPPAAL. Diese Arbeit zeigt, dass sich diese komplementären Stärken durch eine semantikerhaltende Übersetzung von Quartz nach SysML/AVATAR kombinieren lassen.

Die Arbeit gliedert sich in drei Phasen: (i) Machbarkeitsstudie: Quartz-Konstrukte, logische Instanzen, Broadcast-Signale und bewachte Aktionen werden äquivalent in SysML-Elemente überführt; lediglich wenige Sprachmerkmale erfordern kontrollierte Approximationen. (ii) Prototypischer Übersetzer: Ein Übersetzungswerkzeug liest ein bereits in Averest korrektheitsgeprüftes Quartz-Modell ein und erzeugt eine XML-Datei, die unverändert in TTool geladen werden kann. (iii) Evaluation: An repräsentativen Benchmark-Beispielen wird bestätigt, dass die übersetzten Modelle in TTool fehlerfrei ausgeführt werden und die ursprünglichen Sicherheitsbeweise übernehmen, wodurch der Vertrauensbereich über das hinausgeht, was jedes der beiden Werkzeuge einzeln leisten kann.

Der resultierende Workflow ermöglicht es Ingenieurinnen und Ingenieuren, mit deterministischen, synchronen Spezifikationen zu beginnen und maschinengeprüfte Sicherheitsbeweise zu erhalten. Zudem wird dasselbe Entwurfsartefakt um Angreifermodelle und kryptographische Verifikation erweitert. Diese praktische Anwendung eines einheitlichen Pfads zu Safety- und Security-Assurance für reaktive Embedded-Systeme bietet vielversprechende Perspektiven für die Zukunft.

Contents

List of Figures vi

Listings vii

List of Abbreviations viii

1 Introduction 1
- 1.1 General Problem Setting . . . 2
- 1.2 Thesis Objectives . . . 2
- 1.3 Structure of the Thesis . . . 2

2 Language Specifications 4
- 2.1 Quartz: Syntax and Semantics . . . 4
  - 2.1.1 Core Concepts and Semantic Pitfalls . . . 5
  - 2.1.2 Concrete Syntax . . . 10
  - 2.1.3 Formal (Operational) Semantics . . . 11
  - 2.1.4 Combining Syntax and Semantics . . . 13
  - 2.1.5 Conclusion . . . 14
- 2.2 SysML: Syntax and Semantics . . . 15
  - 2.2.1 Diagram Frame and Notation Conventions . . . 17
  - 2.2.2 Concrete Graphical Elements of a SysML State-Machine Diagram . . . 18
  - 2.2.3 Transition Firing Semantics . . . 18
  - 2.2.4 State-Execution Semantics . . . 20
  - 2.2.5 Sub-Machine Reuse, Parameters, and Block Integration . . . 22
  - 2.2.6 Syntax and Semantics of SysML's Remaining Diagram Families . . . 23
  - 2.2.7 Conclusion . . . 24

3 Software Capabilities 26
- 3.1 Averest . . . 26
  - 3.1.1 Compilation . . . 27
  - 3.1.2 Code Generation . . . 28
  - 3.1.3 Formal Verification . . . 29
  - 3.1.4 Conclusion . . . 30
- 3.2 Ttool . . . 30
  - 3.2.1 Security Engineering with SysML-Sec and AVATAR . . . 31
  - 3.2.2 Model-Centric Environment . . . 32
  - 3.2.3 Design-Space Exploration and Performance Simulation . . . 32
  - 3.2.4 Safety and Real-Time Formal Verification . . . 33

3.2.5 Automatic Code Generation and Virtual Prototyping . . . 36
3.2.6 Conclusion . . . 36

4 Translation Feasibility Analysis **38**
4.1 Model-of-Computation (MoC) . . . 38
4.2 Basic Structural Units . . . 39
4.3 Event and Signal Semantics . . . 40
4.4 Time-related Constructs . . . 41
4.5 Control-Flow Statements and Their State-Machine Counterparts . . . 42
4.6 Data-Type Compatibility . . . 43
4.7 Concurrency, Causality, and Determinism . . . 44
4.8 Formal Aspects . . . 46
4.9 Known Gaps and Conclusion . . . 46

5 Program Architecture and Implementation **48**
5.1 Introduction . . . 48
5.2 Averest NuGet Package and Initial Parsing Phase . . . 48
5.3 EFSM-Level Transformation Utilities . . . 49
5.4 In-Memory SysML Structure Types . . . 50
5.5 Requirements Extraction and Filtering Utilities . . . 51
5.6 Translating the EFSM into a SysML State-Machine . . . 51
5.7 Generating Block-level Attributes and Type Information . . . 53
5.8 Requirement Diagram Synthesis and XML Serialisation . . . 53
5.9 Serialising the Complete Model – “Main XML Generation” . . . 55
5.10 Putting It All Together and Concluding Remarks . . . 58

6 Evaluation **60**
6.1 Methodology . . . 60
6.2 Test Case Selection . . . 61
6.3 Results . . . 61
6.3.1 Quartz Specifications . . . 61
6.3.2 Extended Finite-State Machine . . . 62
6.3.3 XML output . . . 63
6.3.4 Observations . . . 64
6.4 Confirmation . . . 66

7 Conclusion and Future Work **67**

A Code **69**

Bibliography **81**

Note **83**

List of Figures

2.1 Curing Schizophrenia Example R0 [8] 8
2.2 Curing Schizophrenia Example R1 without goto [8] 9
2.3 Curing Schizophrenia Example R2 [8] 9
2.4 Guarded Actions of the Surface [8] 14
2.5 Overview of SysML/UML Interrelationship [9] 16

3.1 Averest Design Flow [4] 28
3.2 Overall Approach [6] 33
3.3 UPPAAL Formal Verification [6] 35

6.1 EFSM: Robot02 [30] 63

Listings

2.1 Simple Causality Problems [8] . . . 6
2.2 Multiply schizophrenic local declaration [8] . . . 7
3.1 Pragma-based Directives . . . 31
5.1 SysML Structure . . . 50
5.2 Requirements Extraction and Filtering Utilities . . . 51
5.3 Requirement Info . . . 51
5.4 Filtering Heuristics . . . 52
5.5 Block-level Attributes . . . 53
5.6 Requirement Diagram . . . 54
5.7 Block Diagram Panel . . . 55
5.8 Block Diagram Component . . . 55
5.9 State Machine Component . . . 56
5.10 State Machine Connector . . . 57
5.11 State Machine Subcomponent . . . 57
5.12 Final Statements . . . 58
6.1 Quartz: Robot02 . . . 62
6.2 Fragment of XML . . . 63
A.1 Complete Code . . . 69

List of Abbreviations

EFSM	Extended Finite-State Machine
SysML	Systems Modeling Language
UML	Unified Modeling Language
AVATAR	Automated Verification of Real Time Software
DIPLODOCUS	Design Space Exloration Based on Formal Description Techniques, Uml and SystemC
MoC	Model of Computation
SOS	Structural Operational Semantics
LTL	Linear Temporal Logic
CTL	Computation Tree Logic
AIF	Averest Intermediate Format
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
SGA	Synchronous Guarded Actions
CGA	Control-flow Guarded Actions
DGA	Data-flow Guarded Actions
BDD	Binary Decision Diagram
GALS	Globally Asynchronous, Locally Synchronous
HW	Hardware
SW	Software
ML	Machine Learning
HDL	Hardware Description Language
TEPE	Temporal Property Expression
GUI	Graphical User Interface
XMI	Extensible Markup Language Metadata Interchange
CPU	Central Processing Unit

1 Introduction

The last two decades have seen an unprecedented diffusion of embedded computing: microcontrollers now regulate the fuel–air ratio of an engine, synchronise renewable energy inverters with the grid, and maintain wearable medical devices within therapeutic limits. Market analysts estimate that global revenue for embedded systems will increase from USD 100 billion in 2023 to more than USD 160 billion by 2030, underscoring the strategic role these platforms play in driving innovation across both consumer and industrial sectors, and as their reach widens, the dependability bar rises correspondingly [1]. A modern product must be safe, never placing users in danger, even in the presence of software faults, and simultaneously secure against remote intrusion, data exfiltration, and intellectual property theft [2].

Guaranteeing both attributes is hard because they stress distinct dimensions of a system's behaviour. Safety is usually formulated as constraints on the ordering and timing of events in a reactive system, i.e., a system that maintains an ongoing dialogue with its environment. Tools grounded in synchronous languages, such as Esterel, Lustre, or Quartz, tame this complexity by assuming a global logical clock: every variable has a value at every tick, and causality becomes decidable. The Quartz language, for instance, can also be translated into guarded actions, symbolically model-checked, and then compiled into synthesiser-ready VHDL, a workflow implemented in the open-source Averest framework. [3] [4]

Security assurance, in contrast, adds an adversarial dimension: designers must reason about what an attacker can observe or generate on communication channels. Model-driven security engineers, therefore, move toward graphical notations that integrate threat models and cryptographic protocol primitives, and that link automatically to tools such as ProVerif and UPPAAL. TTool exemplifies this philosophy. By specialising SysML into the AVATAR and SysML-Sec profiles, it offers design-space exploration, real-time model checking, and push-button proofs of confidentiality and authenticity, all in a single tool. [5] [6]

Practitioners are thus caught between two complementary yet mutually exclusive ecosystems. Choosing Averest yields deterministic semantics and strong safety proofs; on the other hand, TTool brings integrated safety and security analysis, but omits the synchronous foundations and verified code generation favoured by hardware–software co-design groups. Bridging this methodological gap is therefore essential: if a Quartz program already proven safe could be faithfully translated into a SysML/AVATAR model, developers would gain access to TTool's security machinery without forfeiting the earlier safety guarantees. This thesis addresses that challenge by analysing the semantic distance between the two languages and realising a prototype translator.

The remainder of the chapter states the precise problem being tackled, enumerates the objectives pursued, and outlines the thesis's structure.

1.1 General Problem Setting

The challenge can be stated concisely: formally verified safety is available in Averest, while integrated security analysis is available in TTool; however, to the author's knowledge, no off-the-shelf workflow delivers both for the same design artefact.

Specifically, Averest/Quartz provides synchronous semantics, guarded-action compilation, and a BDD-based model checker, and TTool/AVATAR-Sec embeds confidentiality and authenticity pragmas, attack-tree modelling, and automated ProVerif calls

Without a translation bridge, designers must either leave synchronous safety proofs to gain security analysis or refrain from integrated security verification to keep deterministic timing semantics. Bridging the languages is non-trivial: Quartz's logical instants, broadcast signals, and causality constraints have no direct counterpart in SysML, while SysML-Sec's attacker model and security pragmas have no equivalent in Quartz. A workable solution must therefore faithfully map constructs, highlight irreconcilable gaps, and automate the conversion so that proofs on both sides remain traceable.

1.2 Thesis Objectives

The objectives of the thesis are:

- Translation-feasibility study:
  Analyse the syntactic categories and semantic domains of Quartz and SysML/AVATAR-Sec, deriving a formal correspondence where possible and identifying constructs that require approximation or are fundamentally unsupported.
- Prototype translation tool:
  Implement an end-to-end translator that takes a verified Quartz program as input, builds an intermediate model, and outputs an XML-encoded SysML file conforming to the current TTool schema, including security-relevant pragmas.
- Empirical evaluation:
  Apply the translator to a number of Quartz programs considered as benchmarks, import the resulting models into TTool, and measure:
  - functional equivalence (simulation traces),
  - carry-over of safety proofs, and
  - remaining mismatches or unsupported constructs will be catalogued to guide future work

These objectives aim to demonstrate that combining Averest and TTool through automated model translation yields a unified, model-driven flow in which formal verification and security checking coexist for the same embedded-system design.

1.3 Structure of the Thesis

This thesis is organised into seven chapters. Chapter 2 surveys the two modelling formalisms, Quartz and SysML, clarifying their syntax, semantics, and the modelling assumptions that will later govern translation. Chapter 3 reviews the capabilities of the tools built around those languages: Averest

for synchronous compilation and safety verification, and TTool for model-centric safety and security analysis. Chapter 4 opens the original contribution in this thesis by analysing the feasibility of translating Quartz into SysML; it formalises semantic correspondences and identifies the gaps that must be bridged. Chapter 5 describes the architecture and implementation of the resulting translation framework, detailing its algorithms, software structure, and integration with both toolchains. Chapter 6 evaluates the code on representative examples. Chapter 7 concludes the thesis, summarising the main findings and outlining avenues for future enhancement and broader integration.

2 Language Specifications

2.1 Quartz: Syntax and Semantics

Quartz belongs to the family of synchronous programming languages, whose central postulate, perfect synchrony, allows every component of a reactive system to share a single, abstract clock [7]. During one clock tick (often referred to as a reaction), all active statements read their inputs and compute their outputs instantaneously; physical time advances only when the program executes an explicit time-consuming construct, such as `pause` [8]. This deceptively simple concept supports the language's deterministic semantics, the tractability of formal reasoning, and the ease with which Quartz specifications can later be mapped to either hardware or software [8].

A need of Global Logical Clock

Modern embedded systems integrate multicore CPUs, dedicated accelerators and complex I/O, all of which must respond predictably to an ever-changing physical environment. Traditional event-driven languages (e.g. VHDL, SystemC) execute processes whenever some signal changes, but physical time can stall if a burst of simultaneous events continues to trigger zero-delay reactions, leading to so-called zero-time races [8]. Synchronous languages dissolve this hazard by indexing computation steps to the logical clock: regardless of internal data dependencies, every synchronous thread reaches the next `pause` together, guaranteeing progress and eliminating scheduling nondeterminism [8].

From a verification standpoint, this means a designer can model the entire reactive behaviour as a finite-state control automaton acting on (possibly) infinite-precision data, an abstract state machine in Quartz terminology. Such machines are amenable to model checking and theorem proving, activities that are otherwise intractable for heterogeneous event-driven code. [8]

Key Concepts Behind the Synchronous Mind-Set

Quartz's execution model rests on four closely related principles that are introduced early in the specifications [8]:

Perfect synchrony: In Quartz, every statement other than pause is treated as taking zero logical time. During one clock tick (a "macro step"), all active threads run their instantaneous code in lock-step and meet again at the next `pause`. Because nothing inside a tick can interleave, reasoning becomes cycle-accurate. The programmer, therefore, writes as if the whole system reacts atomically at each instant, with real time only advancing when a `pause` is reached.

Determinism by construction: Within a tick, each signal (or variable) is either absent or carries exactly one value. Multiple writes that would assign conflicting values in the same tick are forbidden, and the compiler's causality analysis flags such "write-write" conflicts. This single-assignment discipline ensures that, for any given set of inputs, the reaction function of a Quartz program is unambiguous.

Separation of concerns: Quartz makes a clear syntactic and semantic split between control flow and data flow. The predicates `enter`, `move`, and `term` describe how control moves between locations, forming a finite-state control automaton. The work performed along each control edge is packaged as guarded commands (γ, C), which state "when condition γ holds, perform command C". Keeping these two views distinct simplifies both formal proofs and subsequent code generation to hardware, software or data-flow representations.

Extensible synchrony: To model systems that are only locally synchronous, or that must interact with nondeterministic environments, Quartz extends the strict synchronous kernel with dedicated constructs for asynchronous threads and controlled nondeterminism. These additions allow developers to describe GALS (Globally Asynchronous, Locally Synchronous) architectures or abstract, environment-driven behaviour without compromising the deterministic core semantics outlined above.

Quartz in the Landscape of Concurrency Models

In [8], it is highlighted that only synchronous languages deliver all three of (i) constant-time reactions, (ii) deterministic semantics, and (iii) a straightforward path to both hardware and software implementation. Quartz inherits these strengths while overcoming the practical limitations found in predecessors, such as Esterel (e.g., it supports delayed assignments and explicit choice for nondeterminism) [8].

Implications for Syntax and Semantics

Because time and determinism are involved into the execution model, Quartz's syntax tightly couples every control construct with a clear semantic rule explained in [8], like a `pause` marks the only point where the logical clock advances, thus appearing prominently in the grammar, parallel composition operators (`||` synchronous, `|||` asynchronous) encode whether child threads share the same clock or each runs on its own schedule, and immediate (without `next`) versus delayed (`next`) assignments gives precise control over when data becomes visible without resorting to low-level buffering.

The subsequent sections of this chapter will formalise these constructs, beginning with data types and expressions and culminating in the Structural Operational Semantics that prove Quartz programs causality-safe.

2.1.1 Core Concepts and Semantic Pitfalls

Quartz inherits the perfect-synchrony idea from Esterel, yet it must still address two hazards, causality cycles and schizophrenia, that remain in every synchronous language. We first position Quartz among the mainstream models of computation, then explain how its semantics are built to avoid those pitfalls. [8]

Concurrency Landscape

Event-triggered simulation: Hardware-description languages such as VHDL, Verilog and SystemC wake a process whenever a signal in its sensitivity list changes. The simulator then iterates three zero-time phases—elaboration, update and event detection—at the current timestamp before time is allowed to advance. This yields cycle-accurate, deterministic updates, but it can also stall the clock: an unbounded cascade of "same-time" events (a zero-time race) may keep the kernel inside the current instant forever, so that physical time never progresses. [8]

Data-flow process networks: Kahn process networks and CSP-style variants model computation as actors that consume tokens from unbounded FIFO channels; a firing blocks until the required inputs are present and then produces new tokens. Because each actor implements a continuous, monotone stream function, the whole network remains functionally deterministic while exposing abundant streaming parallelism. Practical deployment, however, is fraught with timing issues: fixing a global latency schedule is difficult, finite-buffer feasibility is generally undecidable, and relaxed run-time schedulers can introduce nondeterminism that the pure theory forbids. [8]

Synchronous reactive (Quartz): Languages in the Esterel/Lustre/Signal family, and Quartz itself, run on a single logical clock. Within one tick, every active component reacts atomically: inputs are read, outputs are produced, and all internal updates occur as if in zero time before the system synchronises for the next tick. This "perfect synchrony" makes behaviour deterministic, timing-exact and highly amenable to formal verification. The compiler must nevertheless consider two hazards: causality cycles in the generated equations and multiple re-entries (the "schizophrenia" problem) where a statement is exited and re-entered within the same tick, creating overlapping incarnations of local state. [8]

Clocked Execution and the Control–Data Split

Inside one logical tick, every active statement executes instantaneously; only `pause` (and its macro variants) consumes a tick. Quartz formalises this with three predicates over each statement S [8]: $\texttt{enter}_{\hbar}(S)$ – control enters S, `move(S)` – control moves inside S and `term(S)` – S terminates in this tick.

These predicates form the nodes of a finite-state control automaton, while every data update is packaged as a guarded command $(\gamma,\ C)$. Labelling the automaton's edges with the enabled $(\gamma,\ C)$ yields an abstract state machine whose transitions are purely Boolean, a crucial property used in the causality checker and model checking. [8]

Semantic Pitfalls

Causality cycles:

A causality problem exists if a statement's instantaneous actions depend on their own (as-yet-unknown) result. Simple illustrative modules P16-P19 (Listing 2.1) show that emitting `o` while guarding on `o` leaves the boolean equation system unsolvable, or yields multiple solutions, unless the compiler enforces constructiveness.

LISTING 2.1: Simple Causality Problems [8]

```
module P16(event &o) {
    if(o)
        if(!o) emit o;
}
module P17(event &o1,&o2) {
    if(o1) {
        emit o2;
        if(!o2) emit o1;
    }
}
module P18(event &o1,&o2) {
    if(o1) {
        emit o2;
    ||
```

```
        if(!o2) emit o1;
    }
}
module P19(event &o1,&o2) {
    if(o1) {
        emit o2;
    ||
        if(o2) emit o1;
    }
}
```

Quartz's compiler therefore performs a ternary ("true/false/⊥") simulation of the guard-equation system. A program is accepted only if every signal ends the tick with a unique value (`true`, `false`, or absent). The formal algorithm, equivalent to the speed-independence test in asynchronous circuits, is defined in [8].

Schizophrenia:

A statement is schizophrenic when it is left and re-entered within the same tick, so multiple incarnations of its local variables coexist. The classic loop in Listing 2.2 emits four mutually contradictory signals (`y000`, `y100`, `y110`, `y111`) because the initial iteration of the innermost loop executes the `switch` before any aborts occur, with all signals `x1`, `x2`, and `x3` present, hence `y111` is emitted; the presence of `x3` triggers a weak abort, restarting the loop and upon re-execution, only `x1` and `x2` are present, leading to emission of `y110`; `x2` triggers a weak abort, restarting again with a new scope where only `x1` is present, resulting in `y100`, and finally, `x1` causes another abort, and the restarted loop runs with none of `x1`, `x2`, or `x3` present, so `y000` is emitted.

LISTING 2.2: Multiply schizophrenic local declaration [8]

```
module Gonthier02(event &y111,&y110,&y101,&y100,
&y011,&y010,&y001,&y000) {
    loop {
        event x1;
        weak abort
            {
            ell1:pause;
            emit x1;
            }
        ||
            loop {
                event x2;
                weak abort
                    {
                    ell2:pause;
                    emit x2;
                    }
                ||
                    loop {
                        event x3;
                        weak abort
                            {
                            ell3:pause;
                            emit x3;
                            }
                        ||
                            loop {
```

```
28                                      switch
29                                          (!x1 & !x2 & !x3) do emit y000;
30                                          (!x1 & !x2 & x3) do emit y001;
31                                          (!x1 & x2 & !x3) do emit y010;
32                                          (!x1 & x2 & x3) do emit y011;
33                                          ( x1 & !x2 & !x3) do emit y100;
34                                          ( x1 & !x2 & x3) do emit y101;
35                                          ( x1 & x2 & !x3) do emit y110;
36                                          ( x1 & x2 & x3) do emit y111;
37                                      else nothing;
38                                      ell0:pause;
39                                  }
40                              when(x3);
41                          }
42                      when(x2);
43                  }
44              when(x1);
45      }
46  }
```

Quartz cures schizophrenia by [8]:

1. Surface/depth splitting: the compiler separates enter actions from steady-state actions, duplicating the "surface" for each possible reincarnation (examples R0–R2 in Figures 2.1- 2.3)
2. Variable renaming: each incarnation gets a unique name; worst-case blow-up is quadratic but rarely met in practice
3. Static checks: the rename map is computed before guarded-action generation, keeping later causality analysis tractable

```
module R0(event &xOn,&xOff) {
  loop {
    event x;
    if(x) emit xOn;
    else emit xOff;
    ℓ:pause;
    emit x;
    if(x) emit xOn;
    else emit xOff;
  }
}
```

```
module ExpandR0(event &xOn,&xOff) {
  event x;
  if(x) emit xOn;
  else emit xOff;
  loop {
    {
    ℓ:pause;
    emit x;
    if(x) emit xOn;
    else emit xOff;
    init {x};
    }
    if(x) emit xOn;
    else emit xOff;
  }
}
```

FIGURE 2.1: Curing Schizophrenia Example R0 [8]

```
module R1(event &xOn,&xOff) {
  loop {
    event x;
    if(x) {
      emit xOn;
      ℓ1:pause;
    } else {
      emit xOff;
      ℓ2:pause;
    }
    emit x;
    if(x) emit xOn;
    else emit xOff;
  }
}
```

```
module ExpandR1(event &xOn,&xOff) {
  event x;
  if(x) emit xOn;
  else emit xOff;
  loop {
    {
    if(x) ℓ1:pause;
    else ℓ2:pause;
    emit x;
    if(x) emit xOn;
    else emit xOff;
    init {x};
    }
    if(x) emit xOn;
    else emit xOff;
  }
}
```

FIGURE 2.2: Curing Schizophrenia Example R1 without goto [8]

```
module R2(event kl,&a,&b) {
  loop {
    weak abort {
      emit a;
      if(!kl) ℓ1:pause;
      emit b;
      ℓ2:pause;
    } when(kl)
  }
}
```

```
module ExpandR2(event kl,&a,&b) {
  emit a;
  if(kl) emit b;
  loop {
    {
    weak abort
      if(!kl) ℓ1:pause;
      if(ℓ1) emit b;
      ℓ2 : pause;
    when(kl);
    init {x};
    }
    emit a;
    if(kl) emit b;
  }
}
```

FIGURE 2.3: Curing Schizophrenia Example R2 [8]

Quartz's Guarantees

Quartz provides guarantees such as determinism: the equation-system semantics plus constructive causality ensure one unique logical outcome per tick, bounded re-entry: surface/depth expansion turns any schizophrenic loop into a finite unfolding; the resulting guarded-action set is still finite-state, and proof-friendliness: because control and data are factored, both causality checking and higher-order logic embeddings can reuse the same predicates ($\texttt{enter}_{\hbar}$, `move`, `term`) as lemmas. [8]

With the conceptual ground cleared, the following section formalises how Quartz looks: its type system, expression language and statement grammar. There, we will see how each syntactic form directly connects to the semantics outlined above.

2.1.2 Concrete Syntax

In Quartz's surface grammar, each construct is mapped one-to-one to the semantics. Because the language is statically typed and clock-driven, its syntax is organised around four themes: data types, expressions, statements (including the module interface), and the formal grammar. The following explanation traces them in that order.

Data Types

The compiler infers a uniquely determined minimal type for every expression at compile time. Definition 2.1 ([8]) enumerates the available atomic types: `bool`, bounded and unbounded bit-vectors `bv[n]` / `bv`, and bounded or unbounded integers `nat<n>`, `int<n>`, `nat`, `int`, followed by the composite constructors `array(`α`,n)` and tuples $\alpha \star \beta$ [8].

This sharp distinction between atomic and composite values is more than cosmetic. Only variables of atomic type can participate in write-conflict checks or causality equations; the compiler considers writes to different fields of a composite variable (for example, separate elements of an `array`) to be independent, which allows safe parallel updates in one clock tick. Bounded numeric types also serve as compile-time evidence that overflow is either impossible or detectable, and they directly inform the ternary causality analysis presented earlier. [8]

Expressions and Operators

Literals encompass Booleans (`true`, `false`), decimal or bit-vector numerals (`1010b`, `7u`, `2AFx`), and tuple/array aggregates. Operator precedence is fixed and provided in Table 2.1 ([8]), ranging from tuple construction at the top down to logical equivalence at the bottom, where each operator is formally assigned its typing rule and result bit-width.

Quartz draws a pragmatic distinction between static expressions, which are fully evaluable during compilation, and dynamic ones that depend on run-time variables. It is explained with the canonical example in [8] that a static term such as `3*7-14` collapses to the constant `7` (type `int<8>`), whereas `x*y-z` remains dynamic even though its bit-width can already be inferred. Static evaluation is performed before type checking so that the constant's minimal type becomes visible to later rules.

Semantically, every well-typed expression τ denotes a value $[\![\tau]\!]\xi$ in the set associated with its type, where ξ is the current assignment of program variables [8].

Statements, Interfaces, and Modules

All executable code resides inside a module declaration that lists its typed ports and local variables. The lexical tokens that begin a module header are (`module`, `implements`, `{`, `}`). [8]

Atomic statements: These contain the instantaneous actions available to a programmer, immediate or delayed assignments (`x =` τ`;`, `next(x) =` τ`;`), Boolean signal emission (`emit x;`), and control-flow markers such as `pause`, `halt`, or the `await` family. Even the delayed form `next(x) =` τ`;` executes in zero logical time; it simply stores τ for assignment at the next tick. These instantaneous actions constitute the "micro-steps" that the operational semantics will bundle into one synchronous reaction. [8]

Compound statements: Structures such as sequential composition, synchronous (`||`) and asynchronous (`|||`) parallelism, conditionals, loops, and pre-emption constructs (`abort`, `suspend`, `loop ... each`) are built from the atomic core. Operator precedence ensures that, for example, `S1;S2 || S3` parses as `S1; {S2 || S3}` without relying on parentheses. The `pause` boundary embedded in many of these constructs is the syntactic cue that time has advanced; while other statement execution is perceived as instantaneous within a tick. [8]

A complete module declaration combines ports, local variables, and a body statement. The implements clause optionally relates a behavioural module to a property-only specification module, laying the foundation for later proof obligations [8].

Syntactic Choices and Their Semantic Consequences

Syntax and semantics in Quartz are tied so closely that each surface decision brings a predictable behaviour [8]:

- Immediate versus delayed assignments select whether data observation precedes or follows control movement in a tick, and therefore which guarded-command set the compiler generates.
- Distinct parallel operators (`||` deterministic, `|||` nondeterministic) make the programmer's intent explicit and dictate how the `enter / move / term` predicates compose.
- Explicit bit-widths let the type checker detect overflow hazards early and feed accurate range information to the causality solver.

In short, the grammar is not just a parsing convenience; it is the first line of defence in preserving Quartz's deterministic, causality-free semantics.

2.1.3 Formal (Operational) Semantics

Quartz's informal intuition, "everything between two `pause` s happens in zero logical time", is made mathematically precise through a Structural Operational Semantics (SOS). [8]

Why an SOS?

Plotkin-style SOS rules let one describe the step-wise effect of every syntactic construct without appealing to a particular implementation strategy. By treating a complete program state as a tuple $<$ `E, ι, S` $>$, environment `E`, incarnation map ι, residual statement `S`, each rule derives the next state plus meta-information such as delayed actions and an instantaneity flag. This micro-step view aligns with the synchronous mindset: several micro steps may occur within one tick, but the SOS labels the last one of a tick as non-instantaneous, thereby specifying a macro step or reaction. [8]

Control–Data Separation

The semantics splits neatly into control and data. For every statement `S`, three predicates are defined recursively [8]: $\texttt{enter}_{\hbar}$ `(S)` tells when control can start executing `S`, `move(S)` captures internal progress, `term (S)` holds when `S` finishes this tick.

Because these predicates are pure Boolean formulas over location variables, they encode a finite-state automaton that ignores data entirely. Data effects are instead recorded as guarded commands `(`γ`, C)`, where guard γ is a Boolean condition and command `C` a data assignment or assertion. Attaching the enabled guarded commands to the edges of the control automaton yields the abstract state machine. [8]

SOS Transition Rules (Micro Steps)

According to the transition rules written in [8], the premise ensures the immediate assignment is consistent with the current environment; the conclusion emits no delayed actions ($\emptyset$) and marks the step as instantaneous (`true`). Similar rules collect delayed assignments, check `assume`/`assert`, and treat `pause` as a non-instantaneous action.

Compound rules assemble these atomics: sequence rules concatenate residual statements, `if/else` chooses the branch whose guard evaluates to `true`, and statements that are parallel must and can be executed are unions of the corresponding sets. Crucially, instantaneity is propagated upward: a sequence is instantaneous only when both sub-statements are instantaneous. [8]

Three-Valued Instantaneity and Causality

During rule evaluation, the semantics operate in a three-valued logic $\bot$ `<` `0` `<` `1` that distinguishes between unknown and definite `false/true`. Three-valued logic is defined $\wedge, \vee, \neg$; the SOS uses these operators to propagate partial knowledge until a fixed point is reached. If, after convergence, some signal is still $\bot$, the program exhibits a causality cycle and is rejected. Because the lattice is finite, fix-point iteration terminates for all well-typed programs. [8]

Symbolic SOS and the `MacroStep` Interpreter

Algorithm MacroStep shows how an interpreter alternates: (i) environment approximation, adding default values for signals left untouched in the current tick, and (ii) parallel execution analysis using the transition rules until a stable environment is found. Only then does the interpreter apply delayed actions to form the state of the next tick. The algorithm's structure also clarifies why delayed assignments never influence the semantics of the current reaction: they are stored, not executed, in the `Dprv` register, waiting for the subsequent invocation. [8]

Computing Control Predicates Symbolically

$\texttt{enter}_{\hbar}$ is written in pure propositional terms (Definition 4.5 in [8]; analogous definitions exist for `move` and `term`. Lemma 4.10 in [8] then proves fundamental invariants such as $\texttt{enter}_{\hbar}$`(S)` → `next(in S)` and `move(S)` → `in(S)`, facts later reused to verify compiler transformations. A start-signal variant (Definition 4.9 in [8]) permits restarting statements in a loop context and is needed for equating the SOS with hardware circuits.

Guarded Actions and Surface/Depth Split

After control flow is symbolic, the next task is to derive guarded actions for code generation. Definition 4.14 ([8]) computes the surface actions, that is, the effects that occur upon entering a statement, while a dual definition collects depth actions executed after the first micro step.

This surface/depth dichotomy is essential for curing the schizophrenia problem identified earlier: duplicating the surface of a loop (and renaming local variables by their incarnation index ι) ensures that each variable appears at most once per reaction without altering the control predicates. [8]

Machine-Checked Semantics

All definitions are written "by primitive recursion over statements", which helped to embed Quartz directly into the HOL theorem prover. The embedding has already delivered proofs of compiler correctness and surface/depth equivalence (see Lemmas 8.4–8.6 in [8]). This mechanisation underlines that the SOS is not merely descriptive; it is executable and formally trustworthy.

2.1.4 Combining Syntax and Semantics

At this point, the language's surface grammar and its structural–operational semantics (SOS) can be read side-by-side: for every Quartz phrase, the grammar prescribes a concrete form, while the SOS assigns that form an exact meaning.

From Types to Values

Choosing a data type at the initial stage of programming is important. Definition 2.1 in [8] lists atomic and composite types and explains that each denotes a set of mathematical values, `bool = { true, false }`, `bv[n] =` $\{0 \ldots 2^n - 1\}$, and so on. The SOS therefore evaluates an expression $[\![\tau]\!]\xi$ only when its free identifiers are mapped to elements of those sets in the current environment ξ. Static expressions, which the grammar restricts to variable-free terms and `sizeOf` operators, collapse to constants at compile time; the constant's minimal type becomes available to every later typing and causality rule. In this way, the syntax-level notion of statically evaluable directly feeds the semantics: once an expression is static, no runtime rule can ever observe an unknown ($\perp$) value for it. [8]

Control Predicates driven by Grammar

Compound statements are grammatically depicted in Table 3.2 ([8]), yet each also carries a semantic contract encoded by the predicates $enter_{\hbar}$, `move`, and `term`. Lemma 4.10 in [8], proved in HOL, relates those predicates by facts such as `move(S) → in(S) ∧ next(in S)`. The syntax, therefore, tells us where to write a semicolon or a parallel bar; the semantics tells us when control may cross it. Recursive definitions for `move` under SOS reaction rules illustrate the point: the rule for a sequential composition explicitly mentions the grammar's "semicolon" alternative and shows that `move(S1; S2)` is possible only when `S1` either moves or terminates and passes control to `S2`. [8]

Guarded Commands set from Surface / Depth

Once control movement is symbolic, data updates are isolated by surface and depth functions. Figure 2.4 constructs the surface action set `ActSurf~(`ϕ`, S)` using nothing more than the syntax tree of `S`; every atomic assignment in the source text becomes a pair `(`γ`, C)` whose guard γ accumulates the syntactic path conditions that reach that assignment. The complementary depth statement removes those start-time actions, preserving only the residual behaviour; Lemma 8.4 in [8] proves that `Depth(S)` shares the same control predicates as `S`, while Lemma 8.5 in [8] shows that `Surface~0(S); Depth(S)` is behaviourally equivalent to `S`. In other words, the grammar determines what text is surface (e.g., the left-hand side of an immediate assignment), and the semantics guarantees that duplicating that text for every loop incarnation cannot change the program's observable behaviour, which is a crucial step in curing schizophrenia. [8]

- $\mathsf{ActSurf}_h\,(\varphi, \texttt{x=}\tau) := \{(\varphi, \hbar(\texttt{x}) = \hbar(\tau))\}$
- $\mathsf{ActSurf}_h\,(\varphi, \texttt{next(x)=}\tau) := \{(\varphi, \texttt{next(x)=}\hbar(\tau))\}$
- $\mathsf{ActSurf}_h\,(\varphi, \texttt{assume}(\sigma)) := \{(\varphi, \texttt{assume}(\hbar(\sigma)))\}$
- $\mathsf{ActSurf}_h\,(\varphi, \texttt{assert}(\sigma)) := \{(\varphi, \texttt{assert}(\hbar(\sigma)))\}$
- $\mathsf{ActSurf}_h\,(\varphi, \texttt{nothing}) := \{\}$
- $\mathsf{ActSurf}_h\,(\varphi, \ell\texttt{:pause}) := \{\}$
- $\mathsf{ActSurf}_h\,(\varphi, \texttt{if}(\sigma)\ S_1\ \texttt{else}\ S_2)$
  $:= \mathsf{ActSurf}_h\,(\varphi \wedge \hbar(\sigma), S_1) \cup \mathsf{ActSurf}_h\,(\varphi \wedge \neg\hbar(\sigma), S_2)$
- $\mathsf{ActSurf}_h\,(\varphi, \{S_1; S_2\}) := \mathsf{ActSurf}_h\,(\varphi, S_1) \cup \mathsf{ActSurf}_h\,(\varphi \wedge \mathsf{inst}_h\,(S_1), S_2)$
- $\mathsf{ActSurf}_h\,(\varphi, \{S_1 \parallel S_2\}) := \mathsf{ActSurf}_h\,(\varphi, S_1) \cup \mathsf{ActSurf}_h\,(\varphi, S_2)$

- $\mathsf{ActSurf}_h\,(\varphi, \texttt{do}\ S\ \texttt{while}(\sigma)) := \mathsf{ActSurf}_h\,(\varphi, S)$
- $\mathsf{ActSurf}_h\,(\varphi, \{\alpha\ \texttt{x};\ S\}) := \mathsf{DisableDelayed}(\{\texttt{x}\}, \mathsf{inst}_h\,(S), \mathsf{ActSurf}_h\,(\varphi, S))$
- $\mathsf{ActSurf}_h\,(\varphi, \texttt{during}\ S_1\ \texttt{do}\ S_2) := \mathsf{ActSurf}_h\,(\varphi, S_1)$
- $\mathsf{ActSurf}_h\,(\varphi, \texttt{[weak] abort}\ S\ \texttt{when}(\sigma)) := \mathsf{ActSurf}_h\,(\varphi, S)$
- $\mathsf{ActSurf}_h\,(\varphi, \texttt{weak immediate abort}\ S\ \texttt{when}(\sigma))$
  $:= \mathsf{DisableDelayed}(\mathsf{LocVar}\,(S), \sigma, \mathsf{ActSurf}_h\,(\varphi, S))$
- $\mathsf{ActSurf}_h\,(\varphi, \texttt{immediate abort}\ S\ \texttt{when}(\sigma)) := \mathsf{ActSurf}_h\,(\varphi \wedge \neg\hbar(\sigma), S)$
- $\mathsf{ActSurf}_h\,(\varphi, \texttt{[weak] suspend}\ S\ \texttt{when}(\sigma)) := \mathsf{ActSurf}_h\,(\varphi, S)$
- $\mathsf{ActSurf}_h\,(\varphi, \texttt{weak immediate suspend}\ S\ \texttt{when}(\sigma)) := \mathsf{ActSurf}_h\,(\varphi, S)$
- $\mathsf{ActSurf}_h\,(\varphi, \texttt{immediate suspend}\ S\ \texttt{when}(\sigma)) := \mathsf{ActSurf}_h\,(\varphi \wedge \neg\hbar(\sigma), S)$

FIGURE 2.4: Guarded Actions of the Surface [8]

Equation Systems and Constructiveness

After surface/depth splitting, both control and data are mere Boolean-valued equations. It has already been explained that each location variable ℓ receives an initial equation and a transition equation of the form `next(`ℓ`) = `$\Omega\ell$; together, these equations demonstrate the determinism of the control flow for every legal Quartz program. The same translation produces equations for data variables, which the compiler feeds into a ternary (`true/false/`$\perp$) solver. A program is constructive precisely when that solver eliminates all $\perp$ values; the syntax therefore restricts which programs are even considered, while the semantics rejects the non-constructive remainder. [8]

Static Well-Formedness Rules that Enforce Semantic Safety

Two families of syntactic checks shield the semantics from undefined cases [8]:

- Type lifting and subtyping: It is allowed, where safe, for an expression's type to be promoted so that the subtraction `4 - 2u` is accepted even though the literals originate as `int<5>` and `nat<3>`. Without those grammar-level concessions, many otherwise sensible programs would be semantically untypable.
- Variable direction and storage: Every declaration must label a variable as `input`, `output`, `inout`, or `local`, and as `event` or `memorized`; the grammar forbids writes to an `input` port or reads from an `output` port, so the SOS never confronts an illegal access pattern.

2.1.5 Conclusion

Quartz's language design shows how a minimal concrete syntax can be merged with a fully mechanised operational semantics so that every program accepted by the parser already satisfies strong

semantic guarantees. The type rules ensure that each variable belongs to a well-defined value set and that static expressions are evaluated to constants at compile time. There are syntactically well-formed statements, the three control predicates $enter_{\hbar}$, `move`, `term` define a finite-state automaton whose edges carry guarded commands; a fix-point iteration over a three-valued logic eliminates all "unknown" values, so an accepted program is automatically deterministic and constructive. Surface/depth splitting and incarnation-indexing finally tame the schizophrenia problem without altering the original code's meaning. [8]

Although this section has focused on the core syntax and semantics, the reference manual also illustrates several extensions that are currently under active development. First, delayed assignments have been generalised to cover quantitative time annotations, enabling static schedulability analyses that go beyond the single-tick abstraction. Second, missing Esterel features, such as micro-step variables and inout ports, are slated for either inclusion or systematic source-to-source transformations, as already outlined for variable reincarnation. Finally, ongoing work on asynchronous and data-flow extensions promises to bridge Quartz's deterministic core with globally asynchronous, locally synchronous (GALS) architectures, while preserving the formal backbone that makes verification practical. [8]

These developments suggest a vibrant hereafter in which the language remains small enough for mechanical reasoning yet broad enough to serve as a single executable specification, from high-level functional models down to synthesised circuits and code.

2.2 SysML: Syntax and Semantics

The Systems Modeling Language (SysML) can be defined as "a general-purpose, graphical modeling language for specifying, analyzing, designing, and verifying complex systems that may include hardware, software, data, procedures, and facilities [9]." In contrast to its parent language, UML, which was created for software-centric object-oriented design, SysML was conceived to support the needs of systems engineering across multiple disciplines [9]. Its diagrams, therefore, span four equally weighted pillars[9]:

- Structure – Block Definition and Internal Block Diagrams describe composition, interfaces, and allocation of functionality
- Behaviour – Activity, Sequence, and Behaviour State-Machine diagrams capture control and data flow at every abstraction level
- Requirements – a first-class Requirements diagram enables traceability from stakeholder statements to design artefacts
- Parametrics – Parametric diagrams embed constraint equations that support performance and trade-study analyses

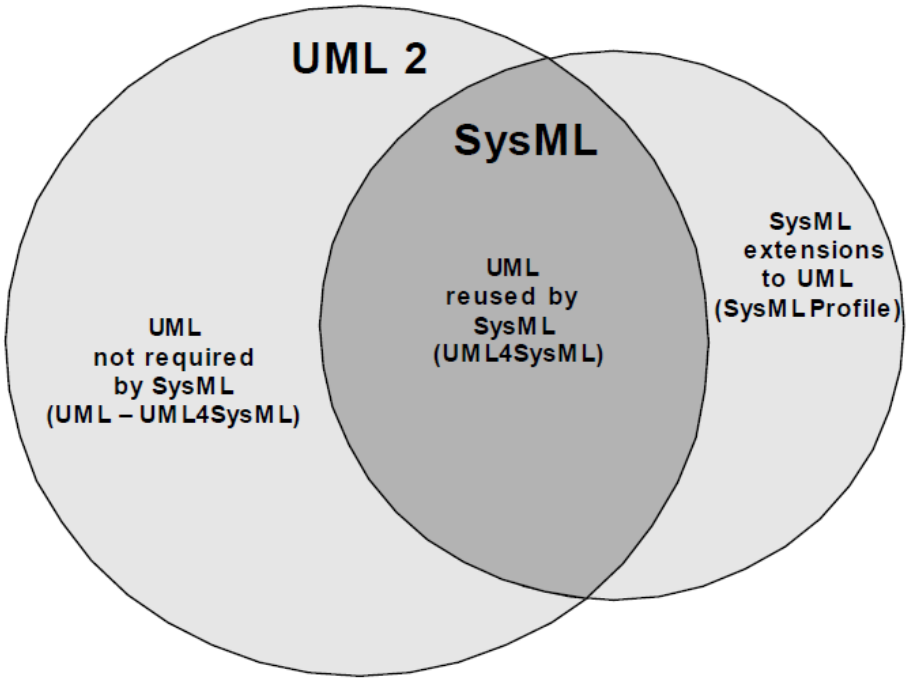

FIGURE 2.5: Overview of SysML/UML Interrelationship [9]

SysML achieves this economy by re-profiling UML rather than inventing a new metamodel from scratch. The language definition begins with a subset called UML4SysML, as shown in Figure 2.5, and then adds or tailors metaclasses as needed by systems engineering practice. For instance, Block, ValueType, and Requirement are added; Class, Component, and other purely software notions are excluded or aliased. There is a list of more than thirty UML metaclasses—among them UseCase, Deployment, and ProtocolStateMachine—that SysML explicitly excludes because their capabilities fall outside the system scope or are duplicative. [9]

The scope of SysML therefore extends from the earliest concept exploration, where Requirements diagrams capture stakeholder needs, through logical and physical architecture (Block and Internal Block Diagrams), down to detailed behaviour specification and performance analysis. Yet the language remains intentionally neutral about implementation: it does not prescribe a development process, a simulation kernel, or a code-generation flow. This neutrality is codified in the conformance clause, which states that "tool vendors may support any subset of diagram kinds, provided they preserve the normative abstract syntax and semantics of the supported elements." [9]

Finally, the specification highlights three guiding design principles [9]:

1. Leverage UML where possible – reuse UML semantics to protect existing modelling investments.
2. Simplify where necessary – remove software-specific or rarely used UML features so that system engineers face a leaner language surface.
3. Extend judiciously – add only those constructs (e.g., requirements, parametrics) that fill clear gaps for multidisciplinary system work.

These principles explain why, for example, SysML keeps behaviour state machines but omits protocol state machines: the former already express both modal behaviour and interaction arrangements without the extra semantic overhead of the latter. Similar reasoning drives the exclusion of deployment and timing diagrams, which are replaced by allocation tables and parametric constraints that better suit cross-domain analyses. [9]

In summary, SysML positions itself as a concise yet comprehensive dialect of UML, purpose-built to bridge mechanical, electrical, software, and human-centric aspects of a system within a single, tool-processable model. Its carefully curated set of diagrams, supported by a common, UML compatible metamodel, provides the foundation on which specialised chapters (such as the state-machine semantics detailed next) build their precise notations and execution rules. [9]

2.2.1 Diagram Frame and Notation Conventions

Unlike UML, every SysML diagram is legally required to sit inside an explicit rectangular frame. Annexe A in [9] explains that the frame "designates a model element that is the default namespace for the model elements enclosed in the frame"; in the case of a state-machine diagram, that designated element is, of course, a `StateMachine` [9]. The rule carries two important consequences [9]:

- All unqualified names that appear inside the drawing are automatically interpreted in the context of the owning state machine, eliminating ambiguity about where a port, variable, or signal belongs.
- If the figure is reused inside another block, for example, embedded as a sub-machine, the outer block must qualify the reused names only when they would otherwise clash with its own.

The heading of the frame sits in a "name-tag" with the top-left corner clipped. The syntax is prescribed as `stm [block] <BlockName> [<DiagramName>]` for state machines, where `stm` is the official abbreviation for the diagram kind. The bracketed `block` keyword records the metaclass (or an applied stereotype) of the designated element; it appears only if there could be confusion, for instance, when several diagrams share the same block or when a stereotype like «`stateMachine`» has been applied. The optional final field distinguishes multiple diagrams owned by the same element, a convenience when, for example, one wishes to keep nominal and failure modes separate. [9]

A diagram description comment may be anchored to the frame to capture version, completion status, and references. This metadata, though ignored by the abstract syntax, is invaluable for requirements traceability and tool-based quality checks. The frame's border may host modelling elements that are logically "on" the owning state machine but visually distinct from its interior. For state machines, the most common border elements are entry and exit points; other diagram kinds place block ports, activity parameters, or interaction gates in the same privileged position. [9]

Inside the frame, graphical elements must conform to the concrete syntax. Table 13.1 in [9] lists the admissible nodes, simple states, composite states, pseudostates, history markers, termination, and so on. While Tables 13.2 and 13.3 in [9] list the paths (transitions, connection-point references), as well as the auxiliary call-outs. Because these tables are normative, a diagram that mixes notation from another section of the specification (for example, activity edges) would be non-conformant; tools are expected to enforce that restriction. [9]

Finally, SysML introduces «`diagramUsage`» stereotypes to name specialised usages, such as ContextDiagram and ModeChart, which remain syntactically state-machine diagrams. The stereotype

appears immediately above the `stm` keyword in the header; its presence lets a tool apply additional consistency checks without altering the underlying semantics of the state machine. [9]

2.2.2 Concrete Graphical Elements of a SysML State-Machine Diagram

The SysML specification anchors every element that can appear inside a state machine frame in three normative tables, as discussed above. Together they define a closed vocabulary of shapes and labels that a compliant tool must recognise and that a modeller may rely on.

The nodes catalogued in Table 13.1 in [9] encompass the full range of behavioural aspects inherited from UML. At the simplest end lie simple states, rectangles that may carry up to three behaviour compartments, entry, do, and exit, each naming an activity that runs on the corresponding lifecycle hook. A simple state can be promoted to a composite state merely by drawing an internal border; that border denotes at least one region and therefore the possibility of orthogonal (concurrent) sub-configuration. SysML treats composites and their regions as ordinary `UML4SysML::State` and `UML4SysML::Region` instances, so no extra stereotype adornment is required. [9]

More specialised vertices appear as pseudostates. An initial pseudostate (black-filled circle) marks the default entry vertex; choice and junction diamonds resolve alternative flows; shallow (`H`) and deep (`H*`) history icons re-establish a previously active configuration when re-entering a composite. Entry and exit points, drawn either inside a composite or on the diagram frame itself, formalise cross-border transitions. At the same time, a terminate cross (a filled circle with an "X") indicates that the encompassing state machine has finished its work. Every one of these symbols maps to `UML4SysML::PseudoState` in the metamodel, with the `kind` attribute identifying its exact flavour. [9]

The node list concludes with the final state and the sub-machine state, whose compartment references another state-machine block; the latter is crucial for behavioural reuse and is still an instance of `UML4SysML::State`, distinguished only by its embedded machine name. [9]

Paths connect those nodes. The principal path is, of course, a transition: a solid arrow whose label follows the canonical `trigger [ guard ] / effect` syntax. The transition object is `UML4SysML::Transition`, and its firing semantics, trigger event occurs, guard evaluates to true, then run effect, remain precisely those of the UML run-to-completion engine. Two auxiliary paths, both instances of `UML4SysML::ConnectionPointReference`, provide alternative routes into or out of a composite via named entry or exit points; they appear as dashed arrows attached to the relevant border markers and are the preferred way to keep large diagrams legible when cross-hierarchy transitions proliferate. [9]

Call-out symbols can be recorded that allow a state machine to be referenced from other diagram types, such as Block Definition Diagrams, Interactions, Associations, or as the principal of an AdjunctProperty. These references never add new runtime semantics but ensure that every syntactic occurrence of the machine, whether as a block on a BDD or as a stereotype tag in a parametric diagram, resolves to the same `UML4SysML::StateMachine` definition. [9]

2.2.3 Transition Firing Semantics

In SysML state machines, a transition fires exactly once per run-to-completion step and only when three enabling conditions hold simultaneously [9]:

- Source state active – the vertex from which the arrow originates must be part of the current configuration; this can be an ordinary state, a history pseudostate, or an entry/exit point on the frame.
- Trigger satisfied – at least one event in the machine's event pool matches the transition's trigger (or no event is required for a completion transition, i.e., one whose label omits the trigger field).
- Guard true – the Boolean guard expression evaluates to `true` in the current variable environment.

These three predicates constitute the "enabled" relation defined for `UML4SysML::Transition` and make the path eligible for selection during the current micro-step. [9]

Run-to-Completion Cycle

SysML inherits UML's run-to-completion execution model: once an event has been chosen from the pool, the interpreter selects a maximal, non-conflicting set of enabled transitions (one per orthogonal region) and executes them atomically. No new event is examined, and no state entry/do/exit code is interleaved, until the configuration becomes stable again. This guarantees single-threaded determinism at the granularity of one external stimulus, even in the presence of orthogonal regions inside a composite state. [9]

Prioritisation and Conflict Resolution

When more than one transition is enabled from the same active vertex, SysML adopts UML's hierarchical priority rule [9]:

- Inner-most (lowest-level) vertices are examined first.
- If several paths share the same source, completion transitions (those without a trigger) take precedence over event-triggered ones.
- If ambiguity persists, it is resolved by tool-defined tie-breaking, but the specification encourages modellers to avoid such designs.

Execution Sequence of a Fired Transition

Once selected, a transition executes the following deterministic sequence [9]:

- Exit chain: starting with the source vertex, every state up to, but excluding, the least common ancestor with the target is exited. Each exit invokes its `exit` behaviour (if present) in LIFO order.
- Effect behaviour: the `effect` expression attached to the transition label runs. It may raise further events; these are queued for the next run-to-completion step.
- Entry chain: states on the path from the ancestor down to the target are entered; their `entry` activities run FIFO, and if the target is composite, its default initial pseudostate is traversed automatically, possibly cascading through additional entry actions.

The concrete syntax `trigger [ guard ] / effect` shown in [9] corresponds one-to-one with this semantic sequence.

Special Forms of Transition

Special forms are explained in [9] as:

- Internal transitions (labelled on the state icon rather than as an arrow) omit the exit/entry chain entirely; only their `effect` executes.
- Deferred events: a state may list events that should not trigger outgoing transitions while it is active; instead, they remain in the pool until the machine leaves that state.
- Time events and change events are handled like ordinary triggers except that the interpreter adds the corresponding events (at the specified timeout or when the Boolean changes to `true`) to the pool automatically.

Semantics of Choice, Junction and History

Pseudostates are governed by additional firing rules as described in [9]:

- A choice vertex re-evaluates its outgoing guards at firing time, selecting exactly one `true` branch; if more than one evaluates to `true`, priority follows the order defined in the model repository.
- A junction vertex requires all guards along a chain of junctions to be satisfied before a single compound transition is formed.
- Shallow (`H`) and deep (`H*`) history vertices store the last active sub-configuration; when a transition targets a history, it fires immediately into that remembered configuration instead of the composite's default initial pseudostate.

Each of these constructs is still a `Transition` in the metamodel, guaranteeing that the general three-condition enabling rule applies.

Determinism and Analysability

Because only one configuration change can occur per event, and guards are side-effect-free Boolean expressions, SysML state machines are input-deterministic. This deterministic firing semantics is central to the translation strategy developed in this thesis. Every Quartz-guarded action becomes a transition whose guard is the Quartz predicate γ, ensuring that the resulting SysML model preserves the original module's synchronous, deterministic reaction semantics while remaining analyzable with standard UML/SysML verification tools. [9]

2.2.4 State-Execution Semantics

A SysML state machine records an object's locus of control as a configuration of active states; that configuration evolves only at the boundaries of a run-to-completion step, thereby guaranteeing that no state entry, exit or do-activity can overlap in time with the evaluation of guards or the dispatching of events. [9]

Lifecycle of a Simple State

When control first flows into a simple state, the interpreter performs, in deterministic order, the following actions ([9]):

- Entry behaviour: the `entry / ...` activity executes exactly once.
- DoActivity: the `do / ...` behaviour may start and, if it is continuous, it remains eligible for execution between event cycles; a discrete do-activity yields control back to the interpreter as soon as it finishes.

- Internal transitions: triggers listed inside the state icon may fire without exiting the state; each occurrence suspends any currently running do-activity, runs the transition's `effect`, and then potentially resumes the do-activity.
- Exit behaviour: immediately before leaving the state via a fired external transition, the `exit / ...` activity runs once and must complete before the transition's own `effect` (if any) begins.

The specification emphasises that deferred events, identified by the `defer` keyword, remain in the event pool while the state is active; they are not lost but will be reconsidered once the machine transitions to another state that does not defer them. [9]

Composite States, Concurrency and Recursion

A composite state encloses one or more regions. Upon entry, an initial pseudostate in each region determines which sub-state becomes active; if several regions exist, they all start concurrently, establishing an orthogonal configuration. While the composite is active, events are dispatched to every region, and a transition in one region does not affect the others unless it leaves the composite as a whole. Should a transition target the composite itself, the entire active sub-configuration is exited in depth, first order, before the composite's own `entry` behaviour runs again. This rule ensures that hierarchical composition remains semantically transparent: the presence or absence of a hierarchy border does not alter the run-to-completion guarantees. [9]

A sub-machine state behaves like a composite whose single region is filled by the referenced state-machine diagram. Parameter binding and result propagation use ordinary association links to the owning block; thus, the execution semantics are identical to those of an in-line composite, but reuse is achieved without diagram duplication. [9]

History Pseudostates

SysML provides two flavours of history pseudostate for restoring a previously active sub-configuration when re-entering a composite state as described in [9]:

- Shallow history (`H`): remembers only the direct child state active the last time the composite was exited. When a transition targets the shallow history icon, the interpreter re-enters that one child and then follows its default path (usually its own initial pseudostate). Nested composites are not reinstated; they start anew from their initial state.
- Deep history (`H*`): records the entire nested configuration (including all orthogonal regions) below the composite. Re-entry, therefore, reproduces exactly the control situation that existed at the moment of last exit. If no history exists, because the composite has never been left before, the target falls back to the region's initial pseudostate(s).

In both cases, execution of `entry`/`exit` behaviours respects the normal ordering rules: the stored states' `entry` actions are rerun upon restoration, whereas the outer composite's `entry` does not repeat because the composite itself never ceased to be active during a history-targeted transition. [9]

Completion, Final and Terminate Semantics

A completion transition, one whose label omits a trigger, fires automatically after all `entry` activities, immediately scheduled do-activities, and internal transitions have finished, provided its guard holds. When the configuration reaches a final state inside a composite region, the interpreter exits

that region and evaluates any completion or external transitions at the parent level. Reaching a terminal pseudostate (circle-cross) ends the entire state machine: no further events are processed and all do-activities are aborted, signalling that the owning block has finished its behaviour. [9]

Determinism and Tool Implications

Because entry and exit actions must complete before any subsequent transition can fire, and because do-activities suspend during run-to-completion processing, no two state behaviours race for shared resources within a single logical step. This matches the determinism requirement we enforced earlier for Quartz guarded reactions. Equally important, the formal definition of history ensures that restoring a configuration is functionally idempotent; re-entering twice yields the same control state. This invariant simplifies equivalence proofs between the hierarchical SysML model and its flattened, guarded-command representation derived from Quartz. [9]

2.2.5 Sub-Machine Reuse, Parameters, and Block Integration

SysML purposefully links behavioural descriptions to the structural fabric of a model so that a piece of behaviour, for instance, a controller mode chart, can be reused, parameterised, and integrated just like any other block. [9] introduces the mechanism behind this linkage: the sub-machine state. Whereas a composite state encloses its nested vertices in-line, a sub-machine state merely references an external state machine, thereby turning the vertex into a call-out to a separately defined behaviour package. The referenced machine may live in the same owning block, in another block, or a common library, making the construct SysML's analogue of a reusable function or subroutine. [9]

Binding Actual Parameters with `AdjunctProperty`

To connect a caller to those formal parameters, SysML borrows the UML call behaviour action idea but implements it declaratively. The modeller places an `AdjunctProperty` on the calling block or on the sub-machine state itself; its `refinement` or `bindingPath` attribute identifies the structural feature that supplies the actual argument. Semantically, the property establishes a bidirectional link: the state machine can read and, if marked non-read-only, write the bound property while it is active. Multiple adjunct bindings may coexist, supporting the common pattern where one state machine instance controls several physical channels distinguished only by parameter values. [9]

Connectors and Associations

Because a «`stateMachine`» is a block, its ports can participate in connectors on the owning block's Internal-Block Diagram. Entry and exit events emitted by the sub-machine instance thus propagate through ordinary SysML signal ports, giving the integration the same "look and feel" as wiring hardware parts together. Associations defined on a Block Definition Diagram may link the state-machine block to value-type parameters, such as calibration constants, or context objects, like error logs. These associations have no runtime effect beyond what the behaviour itself models, thereby preserving run-to-completion determinism. [9]

Semantic Equivalence to In-Line Composite States

Despite the indirection, the execution semantics of a sub-machine state are identical to those of an in-line composite with a single region containing the referenced vertices. Entry into the state triggers the entry behaviour of the sub-machine's top-level state, history semantics (shallow or deep) apply exactly as defined within that referenced machine, and exit performs the corresponding `exit` actions before control returns to the caller. The only observable difference is the presence of parameter

bindings, whose current values form part of the interpreter's environment when evaluating guards and actions. [9]

2.2.6 Syntax and Semantics of SysML's Remaining Diagram Families

While the preceding sections explored behavioural state machines in depth, a complete understanding of SysML's language core also demands familiarity with its three other diagram pillars, structure, requirements, and parametrics, because they share the same frame conventions yet introduce their own modelling vocabularies and execution meanings. The specification devotes separate clauses to each of these families. However, they all follow the same pattern: a concise, concrete notation table, an explicit mapping to `UML4SysML` metaclasses, and a narrative semantics that defines how instances of those metaclasses behave at run time or analysis time. [9]

Block Definition Diagrams (BDD)

A BDD uses the frame header `bdd [package] ...` or `bdd [block] ...` to declare its default namespace. Inside, the block rectangle is the primary classifier; its compartments list value properties, part properties, flow ports, and operation signatures. Inheritance is drawn with the hollow-triangle generalisation arrow, while composition is expressed by a black-diamond-headed association whose multiplicity text sits at the composite end. Semantically, a block defines a type, not an instance; every part property declared within the owning block represents a role that a runtime instance of the block will occupy. The specification states that "all value properties are instantiated once for every block instance," establishing a one-to-one mapping between structural syntax and object-level state. Aggregation depth is unbounded but must be acyclic, a constraint enforced in the abstract syntax and checked by tools at model-validation time. [9]

Internal Block Diagrams (IBD)

An IBD is framed with the heading `ibd [block] ThisBlock`, indicating that it displays a single instance of `ThisBlock` along with its internal structure. Each part property appears as a nested rectangle with `name: Type [mult]`, optionally expanded to reveal its own ports. Connectors, solid lines, bind compatible ports; arrowheads on flow ports indicate item direction, and binding connectors on value properties (depicted as thick black lines) impose an equality constraint between the joined variables. Execution semantics follow instance semantics: at runtime, each connector represents a communication path that obeys the flow-direction rules declared on the connected ports. While binding connectors propagate value changes instantaneously and bidirectionally, unless constrained by stereotypes such as «`multiRate`». [9]

Parametric Diagrams

Parametric diagrams reuse the IBD frame (`ibd`) but specialise the interior for analytical relationships. The centrepiece is the constraint block, a block stereotyped «`constraint`» whose body contains an equation or inequality, written in the expression compartment. When a constraint block is instantiated in an `ibd`-style diagram, the resulting constraint property is shown as an ellipse annotated «`constraint`». Its binding connectors attach the constraint's parameters (also value properties) to value properties elsewhere in the system hierarchy. Semantically, the entire diagram becomes a set of simultaneous equations that a solver must satisfy; the specification notes that binding connectors are "purely declarative and impose no simulation causality," meaning that any variable may be considered dependent or independent depending on the analysis context. Constraint evaluation is therefore outside run-to-completion execution and instead part of an analysis execution defined by the tool. [9]

Requirement Diagrams

Requirement diagrams adopt the frame header `req [package] ...`. The requirement stereotype extends `Block` with mandatory attributes `id` and `text`, plus optional `risk`, `priority`, and `verifier` [9]. The diagram introduces five standard relationships, each with concrete line notation and formal semantics described in [9]:

- deriveReqt: a derivation dependency asserting that the source requirement was refined into the target.
- satisfy: a trace from a design element (often a block or parametric model) to the requirement(s) it fulfils.
- verify: a dependency linking a test case, analysis, or simulation to the requirement it demonstrates.
- refine: a broader transformation relation, typically mapping behavioural or parametric artefacts back to high-level textual statements.
- trace: a non-semantic link for bookkeeping, required to be acyclic but otherwise unconstrained.

The semantics clause specifies that `satisfy` and `verify` relations must form an acyclic covering of every leaf requirement, a rule that tools enforce via static analysis of the dependency graph. [9]

Allocation Tables and Cross-Diagram Semantics

Although it is not a separate diagram kind, allocation appears as a table or as dependency arrows (`allocate`) overlaying any other diagram. Its abstract syntax extends `Dependency` with `kind = allocate`; the semantic clause mandates that allocated elements belong to distinct concerns, for example, allocating a behaviour to a hardware block. This cross-cutting relationship completes the language's goal of end-to-end traceability. [9]

Through these structured notations, SysML ensures that every artefact —structural, behavioural, analytical, or textual —has a precise graphical syntax, a normative metaclass mapping, and an execution or evaluation meaning that is tool-agnostic yet formally defined. Together with the behavioural state-machine rules discussed earlier, they provide system engineers with a single, coherent language capable of capturing requirements, architecture, behaviour, and quantitative constraints within a rigorously specified model.

2.2.7 Conclusion

SysML positions itself as a lean yet comprehensive dialect of UML, expressly tailored to multidisciplinary systems engineering. Language's remit as specifying, analysing, designing, and verifying systems that span hardware, software, people, and facilities is defined, while pruning UML constructs that serve purely software concerns. Across all diagram types, the specification imposes a uniform framing convention. Every diagram resides inside a headered rectangle whose clipped-corner tag identifies the diagram type, the owning model element, and optionally a user-defined name. This rule makes the frame's element the default namespace for every unqualified symbol on the canvas, thereby eliminating ambiguity and enabling rigorous cross-diagram traceability. [9]

The structural pillar, Block Definition and Internal-Block Diagrams, offers a precise, object-oriented syntax whose semantics is instance-based: each block instance owns exactly the value- and part-properties declared in its classifier, and each connector represents a concrete interaction path governed

by port flow direction or binding equality. The behavioural pillar retains UML's run-to-completion semantics for state machines, ensuring that a transition fires only when its source is active, its trigger occurs, and its guard is true, after which a deterministic exit–effect–entry sequence produces a new stable configuration. Hierarchical composition, orthogonal regions, and shallow/deep history extend this semantics without introducing race conditions. [9]

Complementing behaviour, parametric diagrams transform constraint blocks into first-class model elements: their value-property parameters, connected by binding connectors, form analysable equation systems whose solution space supports trade studies and verification without affecting execution order in behavioural diagrams. At the top level, requirement diagrams formalise stakeholder intent in stereotype-extended blocks and tie that intent to design artefacts through semantically typed relations, such as deriveReqt, satisfy, verify, and refine, whose acyclic covering of leaf requirements is a mandated well-formedness rule. [9]

Taken together, these provisions provide SysML with a single, coherent core of concrete syntax tied directly to precise, tool-checkable semantics. Blocks define what exists, behaviours say how those things evolve in time, parametrics constrain quantitative properties, and requirements ensure that every design decision remains traceable to its original rationale.

3 Software Capabilities

Formal-methods toolchains rarely cover the full spectrum of concerns that arise in the design of modern cyber-physical systems. Averest offers a rigorous path from the synchronous language Quartz to verified VHDL, C and SystemC, but its proof machinery is confined to functional safety and temporal correctness. TTool, on the other hand, prioritises safety and security by enriching SysML with the AVATAR and SysML-Sec profiles. However, it begins with graphical models rather than the textual, clock-synchronous programs favoured by hardware designers. The central objective of this thesis is therefore to bridge the two frameworks: by translating Quartz into an equivalent SysML representation, we can retain Averest's compilation and model-checking strengths while unlocking TTool's push-button security analysis and rapid design-space exploration. [10] [11]

The present chapter prepares for that integration. It first surveys Averest, showing how its compilation pipeline, guarded-action semantics and symbolic verifier together guarantee functional correctness from source text to generated code. It then turns to TTool, detailing how multi-profile SysML modelling, early performance simulation, embedded safety proof and automated security verification converge in a single, model-centric environment. With these capabilities laid out, the remainder of the thesis will demonstrate how a systematic Quartz-to-SysML translation allows designers to carry a verified synchronous model into TTool, apply confidentiality and authenticity checks, and still rely on Averest's synthesis back-end for implementation. In other words, the chapter explains what each tool contributes individually and what new assurances become attainable once the two are connected through a standard intermediate notation.

3.1 Averest

Averest is an integrated framework for the model-based design of reactive embedded systems. Centred on the Esterel-inspired synchronous language Quartz, it offers a seamless tool flow in which a designer can write a high-level behavioural specification, compile it into a symbolic transition system, subject that system to machine-checked temporal-logic verification, and finally refine the very exact representation into synthesiser-ready VHDL/Verilog or execution-grade C / SystemC code. The Averest Intermediate Format (AIF), across all stages —compilation, optimisation, verification, and code generation — therefore operates on an identical semantic object, guaranteeing that proved properties remain valid in the executable artefact. Originally developed by the Embedded Systems Group at the RPTU in Kaiserslautern, the toolbox has since been applied to domains ranging from robotics and automotive control to cyber-physical systems research, demonstrating that formal methods can be incorporated into ordinary industrial workflows. [10][3]

3.1.1 Compilation

Compilation is the entry point of the Averest tool flow: it takes a high-level synchronous program written in Quartz and converts it into a symbolic transition system expressed in the Averest Intermediate Format (AIF). Everything that follows, including optimisation, code generation, and model checking, works exclusively on AIF, so understanding what happens in this first phase is crucial for appreciating the rest of the toolbox. [3]

Source-level analysis and elaboration

A Quartz design is structured as a hierarchy of modules. The front-end (namespace Averest.Quartz) parses the concrete syntax into an abstract syntax tree, whose top node is a QModule. Statements such as concurrency (S1 ‖ S2), pre-emption (abort/suspend), and control flow are represented by specialised QStatement subclasses. [12]

After translating the Quartz source into the Averest Intermediate Format (AIF), a single, flat set of synchronous guarded actions, the Averest.Quartz.Compiler expands (inlines) every module call, even those nested inside abort or suspend blocks, so that no hierarchy remains. With this IR in hand, the compiler performs the synchronous–language–specific semantic analyses: type and storage-mode analysis distinguishes between memorised variables (which retain their previous value) and event variables (which reset to a default value when absent). This classification determines each signal's reaction-to-absence semantics, and Constructive causality analysis checks that the instantaneous data-dependency graph in every reaction step is acyclic, ensuring a unique execution order even though all guarded actions are assumed to execute in zero time. Only AIF programs that pass these checks proceed to the subsequent synthesis stages. The result of elaboration is a normalised control/data-flow tree in which all control operators have been expanded into basic guards and assignments, ready for symbolic encoding. [3]

Translation to synchronous guarded actions

The core compilation step is a systematic rewrite of the elaborated tree into a set of synchronous guarded actions (SGAs). An SGA has the form $\gamma \Rightarrow \alpha$ where the Boolean guard γ is evaluated in the current reaction and the action α (an assignment) is executed immediately if the guard is true. Two variants are generated: Control-flow guarded actions (CGA) encode the evolution of the logical program counter (pause labels, abort triggers, etc.), and Data-flow guarded actions (DGA) encode the value updates of variables, distinguishing between immediate and delayed assignments. [13]

Because guards and actions refer only to the current or next logical instant, the collection of SGAs can be interpreted as a symbolic transition relation and lends itself to BDD or SAT encodings. Guarded actions, therefore, provide just enough structure for both software generation and hardware synthesis while abstracting away the rich syntax of the source language. [13] [3]

Averest Intermediate Format (AIF)

The SGAs are serialised into an XML dialect called AIF, implemented by the namespaces Averest.Systems.Aif (AIFSystem, AIFModule, etc.). Each <aif:module> element lists its CGAs and DGAs, its port interface, and an ω-automaton encoding of every temporal-logic assertion embedded in the source. Because AIF is declarative and side-effect–free it supports: Separate compilation: individual modules can be compiled once and later linked symbolically at system-assembly time, Property-preserving transformation: optimisation passes merely rewrite AIF documents; they do not

translate to a different representation, so the link between model and proof is never lost, and Tool interoperability: both the Beryl model checker and the Topaz code generators read the same AIF files, which eliminates front-end duplication. [10] [12]

Conceptually, compilation therefore freezes the high-level model in a canonical form, so that all subsequent activities —verification, synthesis, and simulation — operate on an identical semantic object. The design-flow diagrams in Figure 3.1 for synchronous models make the two-step architecture (Quartz → AIF → targets) explicit and motivate it from an embedded-systems perspective. [14]

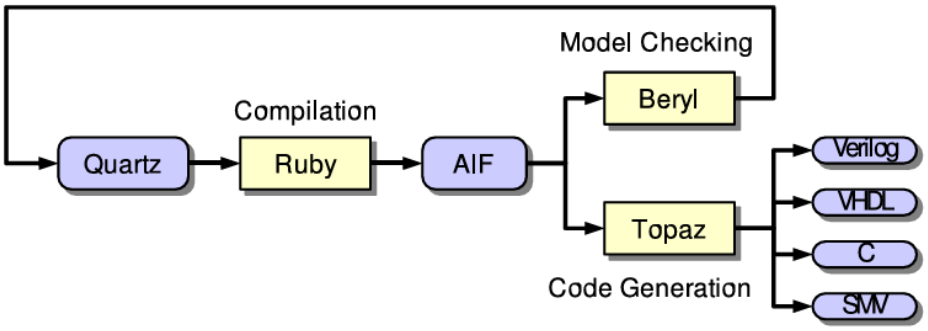

FIGURE 3.1: Averest Design Flow [4]

In summary, the compilation phase transforms a rich synchronous language into a guarded-action transition system expressed in AIF. By performing causality checks, flattening pre-emptive control, and encoding temporal assertions early, the Averest compiler lays a mathematically precise foundation on which both the model checker and the code generators can work without having to understand Quartz syntax. This "single-representation philosophy" is what allows Averest to guarantee that every subsequent optimisation, verification run, or synthesis step is sound regarding the exact behaviour the designer specified in the source.

3.1.2 Code Generation

Once a Quartz design has been compiled into an AIF transition system, Averest's code-generation stage translates that symbolic description into concrete artefacts that can run on a microprocessor or be synthesised onto a chip. Conceptually, this step is performed by the Topaz backend together with a small family of AIF-to-AIF transformation passes that normalise the model before emission. Whereas compilation freezes the semantics, code generation is about refinement: it schedules the synchronous guarded actions in real-time, resolves bit-widths, inserts interface glue, and finally prints a target-language source that is behaviorally equivalent to the original specification. The Averest documentation and teaching materials describe various transformations and code generation that directly leverage the AIF produced by Ruby and feed the rest of the tool flow. [10] [15]

The transformation chain begins with the structural optimisation of the AIF graph. Mutually exclusive guards are connected; combinational loops introduced by earlier clock-refinement steps are broken by introducing temporary registers; and unused signals are pruned so that the eventual HDL netlist or software state machine is free of dead logic. These rewrites are all property-preserving because they operate on the exact declarative representation over which formal verification was carried out.

In effect, every rewrite is a semantics-preserving AIF $\rightarrow$ AIF mapping, so the model checker's proof obligations remain valid after each optimisation pass. [15]

When the graph is in this canonical shape, the Topaz generator emits code. For hardware, it produces synthesiser-ready VHDL or Verilog, in which each guarded action becomes a synchronous process that triggers on the global clock. The control-flow guards compile into a state register and a case statement, while the data-flow guards become conditional signal assignments. For software, it emits an imperative "tick function" in languages such as ISO C, F#, Java or SystemC: the function evaluates every guard, updates memorised variables, and returns when the logical instant is complete. A decade of tool papers list the supported targets as "ISO C, LEGO C, SystemC, Java, Standard ML, Verilog and VHDL", underscoring the back-end's role as a unified HW/SW path. [4] [15]

The back-end, finally, appends a trace-replay harness to each generated file, allowing the counter-examples produced by the Beryl model-checker to be simulated against the concrete code. This post-processing step closes the loop between verification and implementation: designers can, without leaving their IDE, walk a failing temporal-logic trace in the actual C or VHDL code. [4] [15]

In summary, Averest's code-generation stage embodies the tool set's "single-representation philosophy". By starting from the verified AIF, transforming it only through semantics-preserving rewrites, and delegating the final print-out to Topaz, the framework guarantees that every line of C or every synthesised flip-flop implements exactly the behaviour that was proved correct in the preceding verification phase.

3.1.3 Formal Verification

The Averest tool chain treats verification not as a bolt-on analysis but as the semantic centre of gravity around which every other activity revolves. As soon as the compiler has encoded a Quartz program into the guarded-action transition system contained in an AIF document, that same AIF instance is handed to Beryl, Averest's entirely symbolic model checker. Because the transition relation is preserved exactly, the checker works on the very object that will later be optimised and synthesised, eliminating the usual "front-end mismatch" risk that plagues post-hoc verification frameworks. [16]

At the specification level, the designer expresses expected behaviour in standard branching and linear temporal logics, such as CTL, LTL, and even μ-calculus, as ordinary assume/assert clauses woven into the Quartz source. During compilation, these formulas are translated into ω-automata, whose states and transitions are stored alongside the system's guarded actions within AIF. The result is a single, self-contained artefact that describes both the model and the properties to be checked. The verification phase, therefore, requires no additional translation and inherits all static checks (type consistency, causality, clock discipline) already performed by the compiler. [4] [16]

Beryl itself supports several complementary engines. A BDD-based global algorithm explores the reachable state space symbolically and can prove inductive invariants and CTL fix-point properties outright; a SAT/SMT-based bounded engine discharges deep LTL obligations by k-step unrolling and can be configured to switch transparently to k-induction once the completeness threshold has been crossed. For designs whose data path is unbounded (e.g. integer counters), Beryl adds abstract interpretation layers that conservatively fold the infinite lattice back into a finite BDD domain. These engines are not mutually exclusive: the framework allows the user to combine global, local, bounded, and unbounded modes in a single run, so that each property is proved using the cheapest method that suffices. [16]

Whenever a specification fails, Beryl extracts the shortest counter-example trace and writes it back into the AIF file. Because the simulator and waveform viewer understand the same format, the engineer can accurately replay the violating execution cycle, inspect signal histories, and iterate on the design without ever converting between representations. Conversely, once every property has been discharged, the unchanged, already-verified AIF instance flows straight into the Topaz code generator; code generation is therefore refinement rather than recompilation, which means the proofs remain valid by construction. [16]

Averest's verification stage thus realises the classic "correctness-by-construction" ideal: properties are stated in the source language, embedded in the intermediate representation, analysed symbolically in Beryl, and finally preserved as the model is transformed into concrete software or hardware.

3.1.4 Conclusion

Taken together, the three primary stages of the Averest toolbox —compilation, code generation, and formal verification —form a tightly closed semantic loop in which every artefact can be traced back, without loss of information, to the Quartz program that the designer originally wrote. The compiler translates that program into the guarded-action transition system of AIF and, in the same step, embeds the temporal-logic assertions that will later constitute the proof obligations. Verification operates directly on this very AIF instance; it neither requires an auxiliary front-end nor duplicates earlier elaboration work, so the risk of front-end divergence is eliminated by construction. Once the model checker has discharged the properties (or delivered counter-example traces for debugging), the unchanged, already verified AIF becomes the input to the Topaz backends. Code generation is, therefore, properly a refinement rather than a translation. Every optimisation or scheduling decision is expressed as a semantics-preserving transformation of the exact intermediate representation, and the final C or HDL inherits correctness from the proof that was carried out before any implementation details were introduced.

This single-representation philosophy, where verification is conducted on an abstraction that is only loosely related to what finally ships. In Averest, model, proof, optimisation and implementation are all facets of one artefact, continuously maintained from source text to running system. For a formal-methods-centred design flow, especially one aimed at reactive and safety-critical applications, Unity offers the essential guarantee that the executable system enacts exactly the behaviour that was proved.

3.2 Ttool

Where Averest begins with a textual synchronous program and reasons forward to an implementation, TTool starts with a graphical system model and layers increasingly specialised semantics onto a single SysML artefact. Its central idea is that one diagram set, expressed in standard SysML but governed by domain-specific profiles, can serve every engineering concern, from early architecture exploration to cryptographic proof and cycle-accurate prototyping. The tool therefore packages several profiles under one roof: DIPLODOCUS for high-level hardware–software partitioning, AVATAR for detailed real-time behaviour, SysML-Sec for security, and legacy TURTLE for backwards compatibility. Each profile exposes the analysis engines most appropriate to its viewpoint: SystemC simulation for performance, UPPAAL or CADP for safety, and ProVerif for security. However, all operate on the same model, so that results remain traceable across profile boundaries. [17]

3.2.1 Security Engineering with SysML-Sec and AVATAR

The core AVATAR profile was designed for real-time functional design; it provided signals, state-machine behaviour, and timing annotations, but lacked an explicit security layer. A closer analysis revealed several structural gaps: (i) there was no construct for pre-shared keys, (ii) cryptographic algorithms had to be re-modelled from scratch in every design, (iii) the point-to-point channel semantics prevented any realistic eaves-dropping scenario, (iv) an attacker had to be hand-crafted for each project, and (v) neither requirement diagrams nor the temporal-property language (TEPE) provided a place to declare confidentiality or authenticity goals. These limitations, documented in the early AVATAR security studies, made rigorous secrecy or authenticity proofs practically impossible. [17]

The SysML-Sec extension layer closes each of those gaps while retaining the familiar SysML notation. First, it introduces pragma-based directives that are written in block-diagram notes:

LISTING 3.1: Pragma-based Directives

```
1 # InitialCommonKnowledge  BlockA.key  BlockB.key
2 # Confidentiality         BlockX.secret
3 # Authenticity            Sender.s1.m  Receiver.s2.m
```

The `InitialCommonKnowledge` pragma records data that exists in several blocks before the first clock tick, a prerequisite for modelling pre-installed keys. Confidentiality and Authenticity pragmas attach the two most common security objectives directly to attributes and message exchanges. Second, a built-in cryptographic library block exposes canonical operations (sencrypt, sdecrypt, MAC, verifyMAC, ...), so designers write encryption and MAC calls as ordinary state-machine actions instead of re-inventing protocol logic. Third, every block automatically acquires two broadcast signals, `chout` and `chin`, bound to a public asynchronous bus; messages sent over that bus are, by default, visible to the environment, thereby modelling the full Dolev–Yao attacker assumed by formal crypto analysis. [17] [18]

Security work now follows an iterative model-driven flow. At the requirements stage, the engineer stereotypes SysML requirements with a security domain (confidentiality, authenticity, privacy, ...) and links them to attack-tree nodes captured in parametric diagrams; these links drive traceability throughout the process [19] [20]. During design, the confidentiality and authenticity pragmas, together with the cryptographic calls, annotate the same block diagram. A single menu command triggers an automatic transformation of the AVATAR model into π-calculus plus ProVerif queries; TTool supplies the standard Dolev–Yao intruder and launches ProVerif. Proof results—success, failure, or an explicit attack trace—are re-imported and visualised on the original state machines, so the modeller can navigate directly to the point where a secret leaks or a forged message is accepted. [17]

If a proof fails, TTool offers a counter-measure wizard that suggests or inserts encryption, MAC verification, or the designation of a private bus segment. The updated model can be re-checked immediately, and its performance impact can be quantified via the usual SystemC simulation backend. Because the entire cycle—requirements capture, threat analysis, formal proof, counter-measure insertion, performance assessment—operates on the same SysML artefact, security properties remain traceable and valid to the generated C code or HDL that TTool eventually produces. In this way, SysML-Sec elevates AVATAR from a purely functional modelling language to a unified environment for co-engineering safety, real-time behaviour and formal cryptographic security. [19]

3.2.2 Model-Centric Environment

TTool's most distinctive architectural choice is to treat the model, rather than any particular formalism or implementation technology, as the primary artefact. A single graphical repository, encoded in XMI but manipulated through the familiar SysML/UML notation, can be instantiated under several specialised "profiles", each tailored to a different phase of an embedded-system workflow. At the partitioning stage, the DIPLODOCUS profile offers coarse, non-deterministic operators and Y-Chart views, enabling engineers to explore alternative hardware/software splits and evaluate power or latency long before RTL is available. When attention shifts to real-time software design, the model is re-opened under the AVATAR profile, which adds state-machine semantics, timing constructs and TEPE property diagrams while retaining all previously captured requirements. For projects that demand a dedicated security layer, the same diagrams are enriched by the SysML-Sec extensions—confidentiality pragmas, attack graphs, ProVerif integration—without ever forking the repository. Legacy designs created with the TURTLE profile are also loadable. However, TURTLE is now deprecated; its analysis and deployment views can be translated forward to AVATAR, illustrating the toolkit's commitment to upward compatibility. [21] [22]

Because all profiles are hosted in a single executable and GUI, switching viewpoints is a menu action, not a file conversion. The underlying transformation engine maintains stable diagram identifiers, ensuring that requirements traceability, simulation traces, and verification results are preserved intact from one profile to the next. In practical terms, this means that a function first sketched as a high-level DIPLODOCUS task can later acquire timed transitions in AVATAR, be annotated with security pragmas in SysML-Sec, and finally generate C, SystemC or VHDL, all without recreating diagrams or losing proof obligations. The tool's homepage lists this multi-profile capability alongside safety, security and performance verification as the core rationale for "a free and open-source environment for modelling embedded systems". [11] [22]

The consequence for engineering practice is a truly model-centric development cycle: profile changes expose new analysis services (design-space exploration, symbolic model checking, cryptographic proof, code synthesis) while preserving a single source of truth. TTool thus demonstrates that rich, domain-specific semantics can be layered onto standard SysML/UML without fragmenting the design artefacts—a prerequisite for rigorous, end-to-end assurance in complex cyber-physical systems. [21]

3.2.3 Design-Space Exploration and Performance Simulation

In the early phases of an embedded-system project, the central question is which combination of processors, memories, and buses can meet the latency, power, and cost targets of a given workload? The DIPLODOCUS profile, integrated into TTool, was created precisely to answer that question, while the design remains moldable. Its name, DesIgn sPace expLOration based on formal DescrIption teChniques, UML and SystemC, already hints at the method: a UML/SysML model is first captured at a very high level of abstraction, automatically translated into SystemC, and then executed to obtain quantitative metrics that guide the architect towards an optimal hardware/software partition. [23] [11]

The approach adopts the classical Y-chart design heuristic. One branch of the "Y" is the application model: functional tasks are described as non-deterministic data-flow blocks linked by abstract channels whose only semantics is token count. The opposite branch is an architectural template that lists CPUs, DSPs, hardware accelerators and shared buses but leaves their exact parameters open. [23]

The DIPLODOCUS workflow generalises the Y-chart idea into an iterative, multi-level process: application, architecture and mapping views form the partitioning level; results feed forward to software

design (AVATAR) and, in parallel, to a hardware model. At each stage, simulation and formal verification can trigger a reconsideration of earlier partitioning choices, producing a feedback loop that converges on an implementation meeting performance, safety and other constraints (see Figure 3.2).

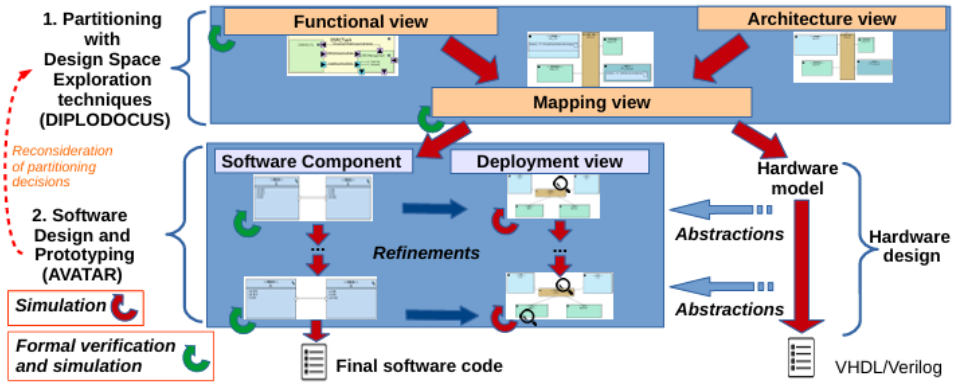

FIGURE 3.2: Overall Approach [6]

During simulation, the automatically generated code traces every task arrival, start, completion, FIFO read/write, cache miss or bus arbitration event. TTool converts the raw trace into both numerical reports—end-to-end latency distributions, mean power consumption, buffer-depth statistics—and graphical artefacts such as UML sequence diagrams that visualise the temporal interplay of tasks on their mapped processors. Results are kept within the same project, allowing the modeller to juxtapose, for example, the latency histogram of a two-core ARM design with the energy bar chart of a four-core heterogeneous one and decide whether the extra headroom justifies the increased silicon area. The original conference paper introducing the environment highlights this integration of automatic SystemC generation, metric extraction and visualisation as the key enabler for early, quantitative trade-off studies. [24] [23]

What distinguishes DIPLODOCUS from many other system-level simulators is that its abstraction deliberately hides algorithmic detail. A task consumes “n” abstract computation units; a channel transmits “k” data tokens; a processor executes “m” units per cycle at “p” milliwatts per unit. Because those parameters are symbolic, the same functional model can be reused when silicon measurements tighten the estimates, yet the structure of the model (tasks and channels) remains stable. This means that the effort invested in design-space exploration is not thrown away but carried forward into the AVATAR and SysML-Sec profiles used later for real-time scheduling and security proof. [24]

Hence, through DIPLODOCUS TTool offers an industrial-strength, model-centric path from early architecture exploration to detailed performance validation, all without leaving the SysML canvas and without rewriting a single line of functional code. By enabling the fast evaluation of architectural alternatives long before RTL or firmware exists, it substantially reduces the risk of discovering, too late, that a chosen platform cannot meet its real-time or energy budget. [24]

3.2.4 Safety and Real-Time Formal Verification

TTool does not delegate assurance to an external, optional phase; the editor itself is a front-end to several model-checking engines that can be invoked at any stage of the design. As soon as a SysML design passes a syntactic check, the user can open the Verification window and choose between two

complementary back-ends. The first is the native AVATAR model-checker, an explicit-state engine implemented within TTool. It constructs a reachability graph on demand and can prove or refute deadlock freedom, reinitializability, state reachability, liveness, and general CTL safety properties. Properties are written either as CTL formulae in a dedicated pink «Safety property» note or, more casually, by right-clicking on any state and marking it for reachability/liveness checking. A single button launches the verification; if a property fails, the tool colours the offending state red and attaches a counter-example trace that the user can replay step-by-step inside the original state-machine diagram. The user manual documents the CTL syntax, the optimisation flags that tame state-explosion, and the automatic generation of AUT/Dot graphs for external visualisation. [25]

For properties that hinge on quantitative time, the designer switches to the UPPAAL back-end. TTool translates the AVATAR or DIPLODOCUS model into a network of timed automata, preserving guard conditions, clocks and synchronisation semantics. In DIPLODOCUS, this export can be executed pre-mapping so that schedulability and task liveness are proved even before the architecture is fixed; in AVATAR, it verifies the timed behaviour of the concrete software design. Reachability ("Is state S ever possible?") and universal liveness ("Is state S eventually inevitable?") are specified by a simple right-click on the operator concerned. At the same time, deadlock-freedom is a single tick box in the verification dialogue. TTool then calls the UPPAAL verifier, captures the outcome, and paints each marked state green or red in the original diagram, thereby avoiding the cognitive switch to an external tool language. [24] [21]

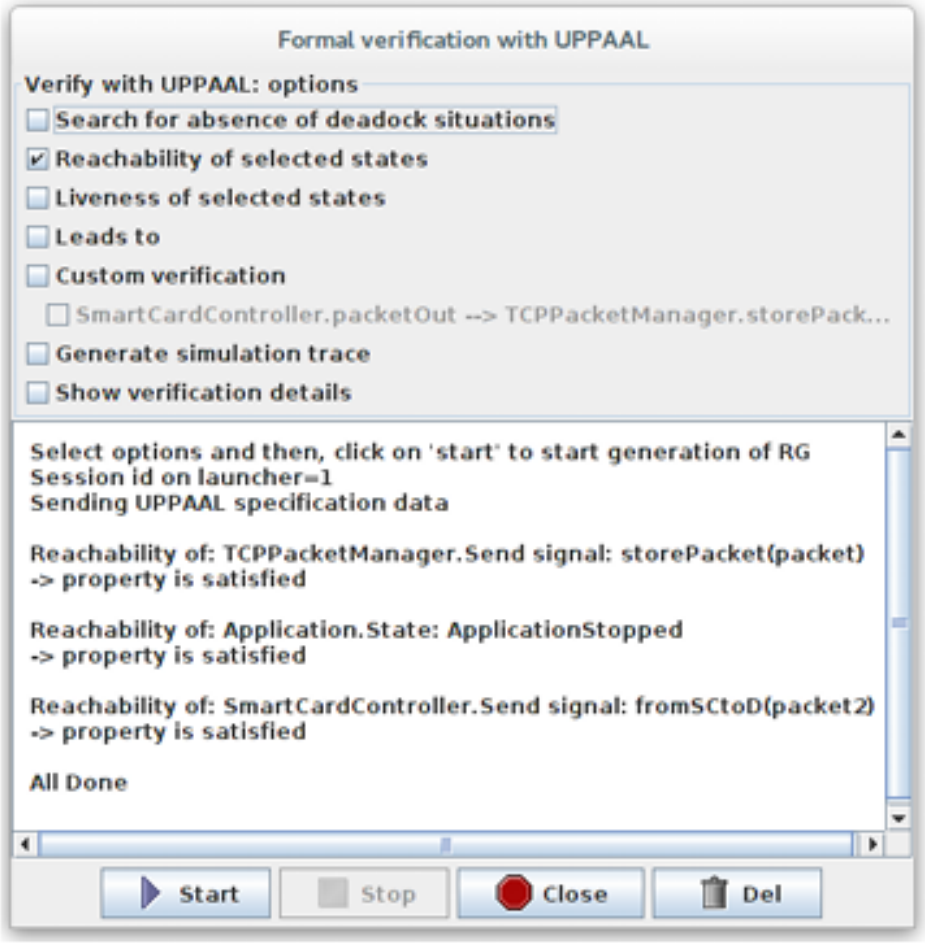

FIGURE 3.3: UPPAAL Formal Verification [6]

Figure 3.3 captures the TTool dialogue that drives UPPAAL verification. In the upper part of the window, goals can be selected, such as absence of deadlock, reachability or liveness; pressing Start launches the checker. The lower console pane then streams the results; each query is echoed, along with UPPAAL's verdict ("property is satisfied" or an error trace if it fails), allowing the user to confirm proofs or inspect counter-examples without leaving the modelling environment.

Although less often highlighted, TTool can also export to CADP for bisimulation and branching-time analysis; the mechanism is analogous, involving automatic translation, remote invocation, and the graphical return of counter-examples, and shows the same commitment to keeping the verification loop within the modelling canvas. Across all engines, the guiding principle is that the SysML model is the sole source of truth: verification never edits an auxiliary copy, and every proof result is stored back into the same XMI file. Consequently, safety obligations established in the early architecture model persist into the detailed, timed, and even security-annotated versions of the design, providing the engineer with continuous and continuously checkable evidence that the system remains free of deadlocks, meets its deadline contracts, and satisfies any CTL property stated in the requirements. [21]

3.2.5 Automatic Code Generation and Virtual Prototyping

The last step in the TTool flow converts a verified SysML model into an executable prototype, eliminating the need for the designer to write a single line of implementation code manually. When the Generate command is issued, the back-end traverses each AVATAR (or DIPLODOCUS) block and emits a pair of translation units, <block>.c and <block>.h, that encode the block's state machine as a POSIX thread; the scheduler for those threads, the time manager and every flavour of synchronous or asynchronous channel are provided by a lightweight "AVATAR runtime" library of roughly two thousand lines of C. A main.c file is generated alongside and contains a declarative deployment table, one line per thread, specifying its target core, priority, and the intended scheduling policy. The designer may, at modelling time, attach fragments of hand-crafted C to particular actions; those fragments are spliced verbatim into the generated code, so algorithmic detail can be introduced precisely where simulation and model checking are no longer required. [21]

Compilation takes two routes. On a desktop workstation, the generated sources are linked with pthread and executed immediately, providing an early functional prototype that preserves the temporal granularity of the AVATAR clocks. For cycle-accurate studies, the same sources are cross-compiled together with the MutekH real-time kernel and loaded into the SoCLib virtual platform. SoCLib is a SystemC library that offers instruction-set simulators for ARM, MIPS, PowerPC, and a catalogue of on-chip buses and peripherals. It can run at transaction-level for speed or at bit-precise accuracy when timing fidelity is paramount. The processor, cache and interconnect models embed hardware counters and spy modules, so cache-miss statistics, bus latencies or lock-contention times appear in TTool as additional sequence-diagram annotations that overlay the functional trace; in effect, the model is "re-animated" with cycle-level numbers while retaining the clarity of SysML semantics. [6] [11] [21]

Hardware-oriented projects go one step further. If a block has been flagged as an HW-accelerator in the deployment diagram, its behavioural description is translated into SystemC processes that comply with the SoCLib component interface; the surrounding platform therefore sees a homogeneous array of CPUs and hardware IPs driven by the same AVATAR runtime. Recent extensions even mix SystemC-AMS leaf components with digital SoCLib modules, allowing analogue peripherals (such as sensors and RF front-ends) to co-simulate with the software stack in a single time-synchronised kernel. The outcome is a virtual prototype that combines binary-accurate software with transaction- or cycle-accurate hardware, ready to answer architectural questions that lie beyond the reach of high-level design-space exploration. [24]

Crucially, the traceability chain remains unbroken. Because the generator works directly from the model that has already passed safety, timing, and, if the SysML-Sec profile was active, security proofs, the produced prototype inherits those guarantees by construction. Execution traces captured on SoCLib are streamed back into TTool and rendered as UML sequence diagrams, so any misbehaviour manifests itself as a deviation from the very diagram that was earlier verified. Thus, automatic code generation and virtual prototyping are not mere convenience features: they are how TTool validates that the high-level assurances established during modelling survive contact with an implementation that is close enough to silicon to expose real-world timing, power and platform effects. [26] [11] [21]

3.2.6 Conclusion

TTool positions itself as a single, model-centred workbench in which one SysML artefact flows through every stage of an embedded-system project. Under the DIPLODOCUS profile, that artefact feeds rapid, quantitative architecture exploration; under AVATAR, it acquires timed behaviour and is

subjected to push-button safety verification; with the SysML-Sec layer, it gains first-class concepts for threats, cryptography and formal security proof. Because the same repository then drives automatic C/SystemC/HDL generation and cycle-accurate virtual prototyping, the functional, real-time and security guarantees established in the diagrams are preserved to an executable prototype. In this sense, TTool realises the long-promised ideal of seamless, traceable co-engineering of performance, safety and security within a free, open-source environment. [22] [11]

4 Translation Feasibility Analysis

Quartz and SysML represent two distinct paradigms in system modelling and specification. Quartz is primarily a synchronous language, explicitly tailored towards embedded systems, ensuring determinism and predictable concurrency. It emphasises precise control of timing and state transitions through a synchronous programming model, where all concurrent activities progress simultaneously in discrete steps. Quartz also provides formal semantics for verification and code synthesis, allowing rigorous validation of system behaviours before implementation. [8]

In contrast, SysML (Systems Modelling Language) is a general-purpose modelling language derived from the Unified Modelling Language (UML), explicitly extended and customised to cater to systems engineering requirements. SysML integrates various diagrammatic representations, encompassing structural, behavioural, and parametric modelling to support a broad spectrum of system complexities. It allows modelling of diverse system domains, including software, hardware, and hybrid systems, through diagrams such as state machines, activity diagrams, and block definitions. [9]

While Quartz is wholly rooted in formal and synchronous paradigms, SysML provides a flexible framework ideal for systems engineering, capable of capturing structural, functional, and behavioural aspects simultaneously. This foundational difference affects the feasibility of translation between the two languages.

4.1 Model-of-Computation (MoC)

At the core of any translation effort lies an alignment or at least a workable reconciliation of the two languages' execution models. Quartz programs support is written for a perfect-synchrony MoC: every thread reacts in lock-step to a global, logical clock. All reads, writes, and signal emissions that occur between two successive pause statements are deemed instantaneous; only the pause itself consumes a logical tick, thereby limiting macro-steps of computation and providing a precise barrier across which delayed assignments (next(x) = τ) become visible. [8]

SysML, by contrast, inherits from UML the run-to-completion MoC for behaviour state-machines. An event placed on the queue triggers a cycle in which the active configuration of states executes all enabled internal transitions, entry/exit actions and do-activities until it reaches a quiescent configuration; only then is the next event dequeued. The cycle is atomic with respect to other events but not with respect to time. The specification leaves physical duration implicit and allows platforms to interleave artefacts such as time-outs or external interrupts. [9]

From a translation viewpoint, these two models are not opposed, but they do differ in three critical dimensions, which will be explained further:

- Tick granularity versus event granularity
- Determinism guarantees

- Concurrency representation

In practice, this alignment proves feasible but exposes two residual mismatches. First, SysML tools rarely offer a notion of logical time, so visual simulations show all macro-steps collapsing onto a single timeline; analysts must read guards ([clk↑]) to spot tick boundaries [9]. Second, specific Quartz constructs, notably the abort and suspend lineages that rely on pre-emption during a tick, require micro-state expansion in SysML to break the run-to-completion atomicity and inject the requisite control flow. These expansions increase diagram size but preserve trace semantics.

Overall, the mapping of perfect synchrony onto run-to-completion is sound provided that each Quartz macro-step is packaged as a single, indivisible event cycle in SysML. This strategy retains determinism, honours delayed assignments, and keeps concurrency synchronous, thereby laying a solid MoC foundation for the subsequent syntactic and semantic translations that follow in the chapter.

4.2 Basic Structural Units

Quartz subsumes interface, local scope and thread within the single syntactic notion of a module [8]. In contrast, SysML separates the description of data from the description of behaviour by pairing a «`block`» with one or more distinct behaviour diagrams [9]. Translation, therefore, is less a matter of inventing new abstractions than of repackaging the constituents of a Quartz module so that they fit the dichotomy of block-as-structure and state-machine-as-behaviour set.

A Quartz header lists event variables and data variables together, indicating direction by a leading '?' for inputs and '!' for outputs [8]. In the target model, this mixed list divides naturally into two categories. Every variable that conveys a persistent value across logical ticks becomes a block property; its directional semantics "provided" or "required" is carried by the isFlow and isConjugated tags of a flowProperty, or, where tool support is limited to ports, by adding a stereotyped proxyPort connected externally through a connector [8]. Event variables, by contrast, possess no duration in Quartz: they are momentary Booleans that revert to false at the next tick [8]. A faithful representation in SysML is to retain a Boolean property for data flow analysis, but pair it with a Signal definition so that other blocks can trigger the state machine without repeatedly polling the Boolean. The separation preserves information important for schedulability analysis, signals travel along connectors, and Booleans remain local, introducing no semantic distortion to the synchronous design.

Inside the module body, Quartz allows nested declarations whose lifetime is delimited by braces [8]. SysML offers no lexical scope on properties, so the translator must either hoist such variables to the block level or simulate their lifetime by resetting them on entry and exit of the state that represents the lexical block. The first solution simplifies the abstract syntax tree and is adequate when the liveness of the variable beyond its textual region is irrelevant to formal verification. The second solution preserves encapsulation at the cost of additional transitions; it becomes necessary only when the absence of the variable outside its scope conveys meaning, for instance, when an out-of-scope reference would have been illegal in Quartz.

Parallelism in Quartz is expressed compositionally by the `||` operator, which spawns synchronous threads [8]. SysML addresses this need through orthogonal regions in a composite state; however, Quartz threads may themselves contain nested concurrency, resulting in an arbitrary tree structure. An inductive mapping copies each parallel level into a composite state with as many regions as child threads, reproducing the synchrony contract by entering all regions simultaneously and by placing a join pseudo-state before every pause. Because SysML ensures that transitions between separate areas

are executed within the same run-to-completion step, the observable outcome aligns with Quartz's lock-step semantics without requiring tool extensions [9].

Assertions at module level (assert A G ϕ, or inline assert(ϕ)) must surface as verifiable artefacts in the SysML model. The normative mechanism is a «`requirement`» element placed in a dedicated requirement diagram and connected to the implementing block via a «`satisfy`» relationship. This manoeuvre respects the SysML philosophy that requirements are not part of execution semantics, yet remain prioritised, traceable down to every element that claims to fulfil them [9]. The original predicate is preserved directly in the text tag of the requirement, allowing a model-checker integrated with TTool to translate it back into temporal logic if desired.

4.3 Event and Signal Semantics

A second axis along which translation feasibility must be examined concerns the notion of an event, that is, something that happens instantaneously and can be sensed by several concurrent actors in the same logical instant [8]. Quartz and SysML both possess first-class mechanisms for modelling such phenomena, yet their contractual details differ just enough to make a basic one-to-one mapping unstable.

Quartz treats an event variable as a Boolean whose truth value is scoped to a single macro-step. Suppose a thread executes `emit(x)`, the variable `x` becomes present for the remainder of that tick and automatically reverts to absent at the following tick boundary. Notably, presence has level semantics rather than count semantics: multiple concurrent emissions of the same event do not accumulate, and a guard such as `if x then ...` is satisfied as soon as at least one emission occurred in the current macro-step. These semantics embody the design philosophy of synchronous languages, such as Esterel and Quartz, where communication is broadcast and instantaneous. [8]

SysML, by contrast, offers two distinct artefacts that together cover this spectrum. A signal is an element of the metamodel that can be raised on a transition and consumed by a trigger elsewhere. A Boolean property held by a block represents a durable state. The UML-inherited run-to-completion rule ensures that if a signal is put on the event pool during a cycle, any transition triggered by that signal within the same cycle will observe it. Once the cycle completes, however, the signal is considered consumed and will not fire again unless it is re-raised. In this respect, signals already match the "single-tick lifetime" of Quartz emissions. What SysML lacks is the convenience of accessing the signal's value as an ordinary Boolean expression in arbitrary guards and actions; guards can only test the occurrence of a signal, not its presence as a variable. [9]

A semantically faithful translation can therefore proceed in two complementary layers. The first layer introduces a signal definition for every Quartz event variable and schedules an outgoing signal emission at each point where Quartz writes emit(e). The second layer adds a Boolean flow property of the same name to the owning block and uses assignment actions to set the flag to true when the signal is emitted and to reset it to false in the entry actions of every state reached after a tick boundary. Because entry actions execute as part of the run-to-completion micro-step that follows the signal emission, the flag is guaranteed to be cleared before any subsequent synchronous reactions, replicating the automatic reset performed by Quartz. Guards inside the state machine may now refer to the Boolean property `[x]` exactly as they did in the source program. However, the outside world interacts solely through explicit signal connectors, preserving SysML's separation between data flow and control flow.

Two secondary issues call for brief comment. First, Quartz allows multiple concurrent emit(e) statements. In the proposed mapping, each statement emits the same SysML signal during the same run-to-completion step. UML semantics dictate that identical signals posted simultaneously combine into a single pool entry, thereby preserving the "idempotence" of emission. Second, Quartz offers valued events (for example, event int31 nickel_amount). SysML signals can carry attributes, so the numeric payload migrates naturally into a signal attribute of type Integer, while the companion Boolean property remains a simple flag indicating presence [9]. The translator must, however, inject an assignment that copies the payload into a dedicated block attribute if the value is still needed after the tick in which it was emitted.

4.4 Time-related Constructs

In Quartz, the notion of time is not anchored to physical duration but to the succession of logical ticks created by the pause statement. Inside a tick, every assignment, signal emission, and guard evaluation is considered instantaneous; only the boundary marked by pause advances the global clock and makes assignments written with the `next( )` operator observable in the subsequent tick. SysML, on the other hand, treats time as something that can be modelled either implicitly, through the sequencing of run-to-completion cycles, or explicitly by using time events or duration guards. [8] [9]

The point of such a mapping is that a Quartz tick corresponds bijectively to a single run-to-completion step of a SysML state machine. Hence, every pause appearing in the source program must be materialised as a transition that leads either to a distinct successor state or back to the same state, thereby forcing the SysML engine to complete the current step and begin a new one. By treating pause as a semantic delimiter rather than a delay instruction with physical duration, one re-creates in SysML the idea of discrete, global time that underpins Quartz's causality analysis [8].

The second temporal idiom in Quartz is the delayed assignment expressed by `next(x) = τ` [8]. Semantically, the right-hand side is evaluated in the current tick, but the new value of x becomes visible only in the next tick [8]. SysML does not have a native construct for one-tick latency. Nonetheless, the effect can be reproduced with an auxiliary slot convention: one introduces a hidden attribute `x_next`, writes the expression into that slot during the current run-to-completion cycle, and copies `x_next` back into `x` at the beginning of the following cycle, typically using an entry action executed on every state reached after a pause. This two-phase commit ensures that data-flow graphs derived from the translated model respect the causality relations mandated by the Quartz semantics.

Quartz also offers quantitative waiting constructs, most prominently the `await`, whose guard is evaluated repeatedly until a condition on either variables or time is satisfied [8]. An await(ϕ) means "stay in a busy-wait loop across ticks until ϕ holds" [8]. In SysML, the nearest counterpart is a self-transition guarded by the negation of ϕ, combined with an external transition guarded by ϕ, both sharing the same triggering event introduced earlier for tick demarcation. Should the designer wish to attach real-time meaning, for example, that a condition must hold within 50 ms, the SysML constraint blocks can be invoked to relate the abstract tick to wall-clock duration. Still, such an interpretation is an optional refinement rather than a prerequisite for behavioural equivalence.

A further asymmetry concerns pre-emption inside a tick, realised in Quartz by statements such as `abort S when(`ψ`)` [8]. Pre-emption is effective at the logical level: if ψ becomes true, the body S is terminated immediately, and control continues in the same tick [8]. SysML state-machines, restrained by run-to-completion semantics, cannot abort a transition mid-flight [9]. Feasible translation, therefore, expands the single Quartz tick into a micro-protocol of two or more SysML states: the first

collects enabling signals, the second executes S under the assumption that ψ is still false, and an alternate branch bypasses S altogether when ψ is true.

Finally, time progress itself is implicit in Quartz: if every thread eventually executes a pause, ticks are guaranteed to accumulate. SysML offers the same guarantee only if no transition guard can stay true indefinitely; otherwise, the model may livelock inside a single run-to-completion cycle. When translating, it is therefore prudent to annotate the produced state machine with an assumption that entry actions terminate quickly, or to include a watchdog constraint block that forbids unbounded execution in a single tick. Such annotations do not change behaviour but document, within the SysML model, the very assumption of finite intra-tick computation on which synchronous theory relies.

4.5 Control-Flow Statements and Their State-Machine Counterparts

The pragmatic value of any translation hinges on the faithfulness with which it conveys control structure, the rules that govern when a segment of behaviour begins, when it ends and under what circumstances it may be bypassed or repeated. Quartz specifies these rules by a compact collection of surface forms, conditional choice, sequential composition, synchronous parallelism, iteration and pre-emption, layered atop the synchronous-reactive MoC outlined earlier. SysML, in turn, equips its behaviour diagrams with pseudo-states, composite states and orthogonal regions expressly to encode branching, looping and concurrency. Although the names differ, a careful reading of their operational definitions shows a near-isomorphism that permits mechanical translation once binding conventions are adopted. [8] [9]

Consider first the conditional. The Quartz phrase `if` ϕ `then S1 else S2` executes either `S1` or `S2` in the same logical tick, depending on whether the Boolean guard ϕ holds at the instant of evaluation [8]. In SysML, the canonical equivalent is a choice pseudo-state whose outgoing transitions bear mutually exclusive guards ϕ and $\neg\phi$. Because run-to-completion semantics ensure that only one guard can fire in a given cycle, the exclusive-execution guarantee of Quartz is preserved [9]. Any assignments nested inside `S1` or `S2` appear as actions on the corresponding outgoing transition; the next-to-auxiliary-slot handles immediate visibility requirements discussed earlier, so no ordering anomalies arise.

Plain sequential composition (S1; S2) entails no semantic complication: the exporter emits an unconditional transition from the state encoding S1 to that encoding S2, so all micro-steps of S1 settle before S2 begins, exactly as Quartz requires [8].

Quartz's synchronous parallel operator (S1 ‖ S2) demands closer scrutiny. The language requires that, within a tick, both threads advance in lockstep, exchanging signals instantaneously and detecting the absence or presence within the same macrostep [8]. The most literal SysML analogue is an orthogonal composite state containing two regions. Each region hosts the translation of its respective sub-statement, and both regions are entered simultaneously whenever the composite state is entered. Because the standard stipulates that all enabled transitions in orthogonal regions fire within one run-to-completion step, the simultaneity contract of Quartz is satisfied without auxiliary synchronisation code [9].

Iteration in Quartz appears in two guises: the unbounded `loop {S}` and the bounded `do {S} while (`ϕ`)`. Both rely on the implicit re-entry of control at the tick boundary [8]. In SysML, these patterns naturally map onto a self-transition when the body consists of a single state, or onto a composite state that ends in a junction pseudo-state, whose outgoing branch either returns to the body

(if ϕ holds) or exits (otherwise) [9]. Because guards are evaluated at the instant the junction is reached after all actions in the body have completed, the tight feedback loop intrinsic to Quartz semantics is preserved.

Finally, Quartz's statement immediate always emit e, which fires an event in the micro-step that precedes the next pause, corresponds to a SysML internal transition or an entry action, depending on whether the surrounding state already owns other entry behaviour. Either placement guarantees that the signal is raised without waiting for an external trigger, mirroring the "immediacy" contract embedded in the keyword.

4.6 Data-Type Compatibility

A translation that preserves behaviour must, at the very least, maintain the range and interpretation of every data manipulated by the source program. Consequently, assessing feasibility at the type level means asking whether each value domain offered by Quartz can be represented, either natively or through a disciplined encoding, in the value space that SysML (and its underlying UML metamodel) makes available.

Quartz exposes a concise yet expressive catalogue of atomic domains. At one end of the spectrum lie logical signals (bool, declared with or without the event modifier) and reals. On the other hand, the language provides parametric integers intk and natk whose bit-width k is part of the static type and therefore of the program's formal contract. A companion construct, the bit-vector (btvk), combines a fixed width with unsigned arithmetic and bitwise operators. Composite structures are built through tuples and arrays, but these are syntactic sugar: a tuple behaves like an anonymous record, an array like a mapping from bounded indices to homogeneous elements; neither contributes novel semantic subtleties beyond those of its constituents. The reference manual explicitly states that arithmetic on bounded integers is modular: overflow wraps rather than raising an exception. [8]

SysML, by contrast, inherits from UML a modest suite of primitive types, Boolean, Integer, Real, String and extends it with ValueType as a user-defined specialisation. A ValueType may own constraints written in parametric diagrams and, under Annexe C in [9], can refine its semantics through OCL or informal text. What SysML lacks natively is the idea that an integer's width belongs to the type system rather than to a run-time representation. Overflow semantics are left to the code generator or to a constraint block supplied by the modeller, not to the core specification itself. [9]

From that comparison, four translation cases emerge.

- Unbounded primitives: Quartz bool maps directly to SysML Boolean, and real to Real; no semantic gap is present because both languages treat the domains as mathematical continua and leave precision to platform choice.
- Bounded and modular integers: A fixed-width int7 (range `-64 ... +63`) admits no direct host in SysML, yet the language's constraint mechanism enables an exact surrogate. The target model introduces a ValueType Int7 with a single attribute value: Integer and a parametric constraint `-64 <= value < 64` [9]. If modular behaviour is essential, for instance, for bit-accurate hardware synthesis, one adds context Int7::value post: result = (self.value + arg) mod 128 to every arithmetic operation. Though verbose, the pattern is systematic and tool-neutral; it pushes width information into analysable constraint blocks without altering surrounding behaviour diagrams.

- Bit-vectors: A bit-vector of width k can likewise be recast as a ValueType Bv_k whose value attribute is an Integer constrained to `0 <= value <` 2^k, accompanied, if necessary, by explicit behavioural libraries that define and, or, xor, and shift in terms of the underlying integer. Alternatively, some SysML toolchains furnish a built-in Unsigned type with user-editable width; where available, the mapping simplifies to assigning that type and recording the width in a tagged value [9].
- Composite aggregates: Because tuples in Quartz are anonymous products and arrays are extensions of ordinary variables by an index set, both structures map cleanly to either nested value types or to blocks with parts in SysML [8]. The choice hinges on intended usage. If the aggregate is intended for arithmetic or bitwise manipulation, a ValueType is more suitable, maintaining the structure atomic in the sense of the metamodel. If, however, individual components participate in connectors, representing the tuple as a block with ports renders these interaction points explicit and verifiable.

Two semantic mismatches nonetheless deserve attention. Firstly, Quartz programs rely on the compiler to generate modular arithmetic, whereas UML/SysML leave overflow behaviour unspecified. A translator that aspires to soundness must therefore develop the modular constraints; otherwise, a SysML execution engine might signal an arithmetic exception where Quartz would wrap around. Secondly, the event qualifier in Quartz imbues a Boolean with a tick-scoped lifetime, a distinction that neither the primitive Boolean nor the Signal in SysML records. Preserving that aspect requires the twin-representation pattern (a signal for inter-block synchronisation, combined with a Boolean property and automatic reset for intra-tick guards).

Despite those caveats, no Quartz type is inexpressible in the vocabulary endorsed by the SysML standard. Where the mapping is not bijective, disciplined use of ValueType and constraint blocks suffices to re-establish the exact range and algebraic properties assumed by Quartz semantics. Hence, at the level of value domains, the feasibility of translation is upheld, with clearly delineated obligations on the translator to inject auxiliary constraints wherever the target language would otherwise leave behaviour underspecified.

4.7 Concurrency, Causality, and Determinism

The final axis on which translation feasibility must be weighed concerns the rules that govern how simultaneously active pieces of behaviour interact, whether their combined effect is deterministic, and under what circumstances a specification is statically rejected as causally inconsistent. Quartz and SysML each provide mechanisms for parallel composition. Still, they inhabit markedly different conceptual worlds: Quartz is rooted in the algebra of synchronous languages, whose first principle is the existence of a single, global logical instant, whereas SysML inherits from UML an architecture centred on interleaving and an explicit event queue [8] [9]. Reconciling the two, therefore, calls for a careful review of both the guarantees Quartz offers and the freedoms SysML leaves open.

Quartz enforces determinism through three pillars. First, the language interprets the operator `||` as perfect synchrony. Every constituent thread observes the exact logical tick boundaries, and every signal is broadcast instantaneously to all threads in that tick. Second, Quartz subjects each module to a causality analysis: if two threads could, in the same tick, write to the same variable or simultaneously emit and test the same signal cyclically, the compiler reports an error. Third, the semantics specifies

that, given an input history, a program yields a unique trace of variable valuations and signal presences; non-determinism is disallowed by construction. These properties make reasoning tractable, for they reduce verification to the study of a single, total state transition relation. [8]

SysML, although capable of encoding synchronous behaviour, does not impose it by default. Parallelism appears in two impressions. Within a single state machine, concurrency is expressed through orthogonal regions; during a run-to-completion step, every area is scheduled conceptually simultaneously. However, the standard permits the execution engine to reorder independent actions, as long as the externally visible trace remains observationally equivalent. Across multiple state machines, either separate blocks or separate behaviours inside the same block, scheduling relies on an event pool that serves one event at a time. The standard explicitly allows more than one transition to be enabled by a given event; if they are not ordered by priority, the resulting choice is nondeterministic. Consequently, determinism in SysML is possible but nowhere guaranteed; it is an emergent attribute of the diagram authored by the modeller. [9]

For translation, the question becomes whether Quartz's determinism survives mapping into a language that admits non-determinism. The solution is affirmative under a disciplined transformation policy. Because Quartz's causality check ensures that no pair of threads races on the same variable, each transition set generated for the corresponding orthogonal regions will be conflict-free: guards derived from distinct source threads are mutually exclusive, or they write disjoint attributes [8]. Suppose a rare case arises where two guards test the same condition but do not use the same variable. In that case, the translator can insert an ordered pair of transitions labelled by the same guard but assigned strict numeric priorities, thereby emulating the stable arbitration order inherent in the synchronous model.

Causality hazards of the instantaneous feedback type, where a signal is both emitted and queried within the same tick, need particular scrutiny. Quartz compiles such patterns successfully only if the resulting Boolean equation has a unique least fixed-point; otherwise, the program is rejected [8]. SysML offers neither language-level fixed-point semantics nor a causality checker [9]. The translation, therefore, implicitly bans such feedback, as any Quartz program that passes compilation is already guaranteed to yield an acyclic, well-founded dependency graph. The guards and actions exported into the target model inherit that acyclicity, so the absence of a causality checker in SysML does not risk stability.

A subtler divergence concerns broadcast semantics. In Quartz, an `emit (e)` performed by any thread renders `e` present for all threads during the remainder of the tick; presence is a level property, not a counting one [8]. In SysML, broadcasting is realised through the posting of a signal instance into the event pool [9]. Multiple emissions of the same signal within one run-to-completion cycle naturally connect because UML stipulates that only one pool entry exists per signal when identical attribute values are supplied [9]. Nevertheless, another block could consume the signal and re-emit it within the same macro-step, violating the single-producer-single-consumer illusion that is inherent in synchronous semantics [8]. Preservation of broadcast semantics, therefore, requires that emissions be confined to the state-machine generated from the same Quartz module, or that external senders be constrained via «constraint» blocks or modelling guidelines, to emit only between macro-steps. This prerequisite does not hinder feasibility; it merely identifies an interface contract between the translated component and its SysML environment.

4.8 Formal Aspects

Whereas the previous sections have concentrated on the operational alignment of Quartz and SysML, practical translation also depends on how the two languages capture non-functional intent and formal correctness claims. In safety-critical development, such metadata can be as consequential as behaviour itself; losing it in translation would weaken traceability. Although Quartz and SysML approach the problem from different angles, they provide complementary mechanisms that can be related in a semantics-preserving way.

Quartz distinguishes two broad classes of logical annotations. Inline `assert(`ϕ`)` statements stipulate invariants that must hold at the instant of execution. At the same time, the satisfies block at module level lists temporal properties, typically interpreted under an LTL or CTL semantics, ranging over the infinite tick sequence. Both kinds are executable in the sense that the run-time monitor produced by the Averest tool chain will trap if a violation occurs. The presence of an assume construct further allows the modeller to declare environment constraints that need not be enforced by the program but must be relied upon during verification. [8]

SysML, in contrast, externalises correctness claims into a requirements model. The standard prescribes a «requirement» stereotype with attributes id, text, and risk, and provides three relationships to the design model: «deriveReqt», «satisfy», and «verify». Behavioural constraints can also be expressed as constraint blocks, which embed equations or Boolean predicates and can be tied to value flows in parametric diagrams. [9]

A semantics-oriented mapping, therefore, associates:

- every assertion in Quartz with a low-level «requirement» element whose text field contains the original Boolean expression, and whose «satisfy> connector targets the state machine or, when the assertion refers to data only, the owning block;
- every assumption with a constraint block stereotyped «assumption», connected via «refine» to the context that is expected to honour the condition;
- every temporal property with a higher-level «requirement» plus a «verify» relationship pointing to a test case artefact. That test case carries the temporal formula, either in raw syntax or translated into TTool's pattern language, so that automated prove-or-falsify tasks can be scheduled [9].

In this scheme, the SysML model gains complete traceability from requirement to design, while the Quartz origin of each property remains visible.

4.9 Known Gaps and Conclusion

The comparison carried out in the foregoing sections shows that almost every construct of Quartz can be functionalised within the expressive envelope of SysML. However, perfect translatability is not a binary attribute; instead, it shades gradually from "straightforward" through "possible with disciplined patterns" to "requires future language or tool support." This closing section enumerates the residual mismatches that currently occupy the latter two zones and, in doing so, frames an agenda for continued research.

The first and most conspicuous gap lies in the treatment of bounded integers. Quartz types, such as int7 or btv16, take priority in the static semantics: they support wrap-around arithmetic and participate

in compile-time causality checks that guarantee overflow cannot propagate inconsistently. SysML, by contrast, views an Integer as an abstraction over the width provided by the execution platform and leaves overflow behaviour to code generators or constraint annotations. While the value-type pattern discussed earlier faithfully constrains the range of a variable, it can only approximate the algebra of modular arithmetic by embedding the requisite mod-k equations in OCL or parametric constraints. Because few industrial tools evaluate such constraints symbolically, the verification power available in Quartz is challenging to replicate downstream. A principled solution would extend the UML metamodel with a fixed-width integer specialisation whose semantics are formally aligned with synchronous languages. [8] [9]

A second, subtler issue concerns the ordering of multiple next updates that share a logical tick. Quartz guarantees that the evaluation of right-hand expressions is instantaneous and that the commit to base variables co-occurs at the start of the next tick [8]. SysML offers no primitive to explicitly state that two assignments executed in one run-to-completion step must commit in lockstep at the beginning of the following step [9]. The auxiliary-slot technique recovers tick-delayed visibility, but the relative ordering among several such commits remains implicit. In practice, the ambiguity rarely manifests, yet a faithful semantics would either serialise all delayed updates into one composite action or insert micro-states whose sole responsibility is to perform the commits in a deterministic order. Formalising a tick-atomic stereotype for transition actions represents another avenue for future standardisation.

A third limitation pertains to broadcast semantics across block boundaries. Within an individual state machine, the mapping of Quartz events to SysML signals, along with Boolean flags, maintains the present/absent invariant. However, once distinct blocks begin to exchange those signals, nothing in the core specification prevents one block from re-emitting a signal inside the same macro-step in which it was consumed, thereby violating synchronous single-producer assumptions. Constraint blocks can forbid such patterns, but their enforcement is tool-dependent [9]. A language-level notion of tick-local broadcast, analogous to the run-to-completion pool yet scoped globally, would eliminate the reliance on user discipline.

Finally, there is a methodological question about verification carry-over. Quartz boasts a mature tool chain for model checking temporal properties formulated in the satisfies clause [8]. SysML, conversely, relies on external profiles (PSM, MARTE) or vendor-specific integrations to conduct temporal verification [9]. Translating a property into a «requirement» or «test case» element preserves traceability but says nothing about the soundness of re-proving the property after architectural elaboration. Establishing a proof-transformation theorem stating that a property proved on the Quartz model remains valid on the translated SysML model under well-defined refinement conditions is an open research problem.

5 Program Architecture and Implementation

5.1 Introduction

Model-driven design flows for embedded systems routinely combine a diversity of specification languages and analysis tools. In such settings, the Quartz language offers a concise, synchronous notation for reactive behaviour [8]. In contrast, SysML, particularly its AVATAR profile, as supported by TTool, serves as a primary language for simulation, model checking, and code generation [17]. Bridging these two worlds requires more than syntactic conversion: it demands that the behavioural semantics and any formal properties stated in Quartz survive intact in the diagrammatic representation understood by TTool.

The translation program written for this thesis fulfils that role. Written entirely in F#, it parses a Quartz source file, asks the Averest back-end to compile the program into an Extended Finite-State Machine (EFSM), performs a series of semantic refinements, and finally serialises the result as a single, self-contained XML document that TTool can import. Because the exporter also lifts each Quartz `assert` clause into a dedicated requirement element and automatically wires a «`satisfy`» connector from the requirement to the resulting state-machine, the traceability chain remains explicit from source code to SysML artefact.

5.2 Averest NuGet Package and Initial Parsing Phase

At the first section of the translator lies the Averest NuGet package, a mature collection of libraries that implement the complete Quartz front-end: lexical analysis, parsing, static type checking, and the generation of a clock-synchronous EFSM [12]. Leveraging this package offers the advantage of obtaining a SysML state machine equivalent model, which can be easily translated into SysML state machines.

The translator begins with a single line of configuration:

```
let quartzFile = @"...Robot02.qrz"
```

Internally, the path is passed to

```
let parsedModule = Compiler.ParseQuartzModuleFromFile quartzFile
```

which yields a `ModuleDef` comprising two parts: a sequence of declarations (`declStmt.decL`), and the main body (`declStmt.stmt`). Both are F# data structures produced by Averest, already decorated with type information, lexical positions, and elaborated names. These features will later be used to generate accurate SysML attributes and requirement identifiers.

Immediately after parsing, the translator invokes

```
let efsm = Quartz.ComputeStateTrans true mainStmt
```

where the boolean flag requests control-flow mode. In that mode, the Averest compiler inserts explicit "pause boundaries" into the EFSM. Every synchronous step delimited by the `pause` keyword in Quartz becomes a pair of non-instantaneous nodes. This explicit alternation will later enable a faithful representation of temporal semantics in the SysML state machine.

Finally, a call to `Quartz.AddSinkState` appends a terminal node that collects all transitions flagged as immediate; this node guarantees that the EFSM is structurally complete (every execution path ends somewhere) and makes it straightforward to represent the termination of synchronous instant behaviour in SysML.

At this point, the input file has been transformed into a fully elaborated semantic model, an EFSM enriched with type and location data, ready for the refinement and export stages described in the remainder of the chapter.

5.3 EFSM-Level Transformation Utilities

Once the raw EFSM has been obtained from Averest, the translator applies a pair of semantic refinements whose purpose is to make the machine operationally compatible with the execution model expected by AVATAR. The core difficulty stems from Quartz's `next()` operator: within a single logical tick, a process may assign the future value of a variable while simultaneously reading its present value. Averest records such statements as `AssignNxt` actions, but SysML offers no native notion of deferred assignment [12].

The translator, therefore, adopts a two-phase update protocol implemented by the mutually independent functions `transformEFSMWithCommit` and `commitAssignments`.

Conceptual idea is that instead of trying to preserve `AssignNxt` directly, the compiler first rewrites every deferred update `AssignNxt(LhsVar(x), ..., e)` into an immediate assignment on a shadow variable `x_next`. In a second sweep, it inserts, on every EFSM node, an unconditional "commit" action that copies `x_next` back to `x`, thus emulating the synchronous register update that occurs between ticks. Because the commit action is unconditional and appended to the surface-action set of each state, it is executed in the same atomic step as any other transition leaving the state.

Technical details.

- `makeNextQName` queries Averest's global name table, allocates a fresh symbol called `<original>_next`, and returns it as a `BasicName`.
- `transformEFSMWithCommit` traverses the set `surfActs` attached to every EFSM node, pattern-matches on `AssignNxt`, and converts it into `AssignNow` using the newly minted shadow variable.
- `commitAssignments` collects all assignments whose left-hand side ends in "`_next`", constructs the corresponding base variable via string replacement, and emits unconditional `AssignNow` actions of the form `x := x_next`. These unconditional commits are merged into the existing `surfActs` set via a simple set union, preserving determinism.

One minor optimisation is performed while traversing the EFSM: assertions are recognised by their `AssignNow` signature or explicit `Assert` tag and are left untouched, because they will later be reused to build requirements in the SysML model. An internal-naming predicate (`isInternal`) prevents temporary variables such as "`__ell...`" from entering into the final diagram.

The outcome of this two-pass transformation is a structurally unchanged EFSM, whose updates are purely immediate, greatly simplifying the subsequent mapping to SysML states and connectors.

5.4 In-Memory SysML Structure Types

The exporter does not stream XML directly from the EFSM. Instead, it constructs a language-neutral intermediate representation consisting of four modest record types:

LISTING 5.1: SysML Structure

```
type SysMLState      = { Name: string; EntryActions: string list }
type SysMLTransition = { From: string; To: string; Guard: string; Action: string
    }
type SysMLBlock      = { Name: string; States: SysMLState list;
                         Transitions: SysMLTransition list }
type SysMLConnection = { FromBlock: string; ToBlock: string; Signal: string }
```

Design rationale.

- Entity coherence: By grouping all information related to a given state or transition into one immutable record, the code avoids the scattered information that would arise if fields were emitted in sections.
- Separation of concerns: The generator that consumes the EFSM to build these records remains oblivious to XML syntax; conversely, the serializer that emits XML need not understand EFSM minutiae.
- Extensibility: Additional fields (e.g., colour, stereotype, timing annotation) can be appended later without disturbing existing code.

Field choice.

- `EntryActions` is stored as a list even though the current translator never emits entry actions. The redundancy is intentional: it anticipates future optimisations where constant initialisations can be hoisted from self-loops into state entry code.
- `Guard` and `Action` are plain strings. Retaining the exact textual form produced by Averest guarantees traceability.
- `SysMLConnection` is defined for completeness, even though the present prototype generates a single block. Its presence signals the intent to handle multi-block architectures in later work.

Together, these lightweight data structures form the pivot between Quartz semantics and SysML syntax. They enable the exporter to assemble a coherent, type-checked model in memory, complete with states, transitions, variables, and requirements, before a single byte of XML is written, thereby facilitating testing and future feature growth.

5.5 Requirements Extraction and Filtering Utilities

A faithful translation must preserve the formal properties of the original. Hence, the current prototype focuses on `assert` statements embedded within the operational code, as they are immediately visible in the EFSM generated by Averest.

In SysML / AVATAR, requirements are entirely separate from the state machine that realises them. By lifting Quartz assertions into the requirement domain, we achieve two goals simultaneously:

1. Traceability: Each correctness condition is represented by its own `COMPONENT type="5200"` node, so that subsequent verification tools, or a human reviewer, can locate the specification without scanning behavioural code.
2. Separation of concerns: Removing assertions from the state machine keeps the latter focused on what the system does, while the requirement diagram records why the behaviour is correct.

The utility module comprises two functions:

LISTING 5.2: Requirements Extraction and Filtering Utilities

```
extractRequirementsFromEFSM : NodeOfEFSM array -> RequirementInfo list
removeAssertActionsFromEFSM  : NodeOfEFSM array -> NodeOfEFSM array
```

`extractRequirementsFromEFSM` walks through every `(guard, action)` pair in the `surfActs` set of each EFSM node. Whenever the pattern matcher encounters `Assert(qname, expr)`, it packages the textual predicate into a `RequirementInfo` record whose fields mirror TTool's metamodel [27]:

LISTING 5.3: Requirement Info

```
{ Id          = Guid.NewGuid().SubString(0,8)
  Description = sprintf "%s : assert %s" qname expr
  Kind        = "Functional"
  Criticality = "Low"
  ReqType     = "SafetyRequirement"
      }
```

A fresh eight-character identifier ensures that requirement IDs are unique. The kind and criticality are left at conservative defaults. Still, their explicit presence in the record makes them easy to refine if a future version of the tool reads structured annotations from the Quartz source.

`removeAssertActionsFromEFSM` performs the dual task of filtering: it returns a deep copy of the EFSM in which every `Assert` action, or bookkeeping assignment generated from an assertion (`__rteAtLine...`), has been deleted. The behaviour diagram will therefore remain uncluttered, while the requirement node, produced in a separate `<Modeling>` section, retains the predicate for formal analysis.

5.6 Translating the EFSM into a SysML State-Machine

With requirements safely gathered, the filtered EFSM is ready to be mapped onto an AVATAR state machine. This transformation is encapsulated in the single function `EFSMtoSysML: NodeOfEFSM array -> SysMLBlock`, which converts the graph-structured EFSM into the flat, record-based data model.

Mapping Strategy

- Nodes → States: Each strongly-connected component index (`node.sccIndex`) becomes a state name `State_<n>`. No attempt is made to recover the original Quartz line numbers or identifiers; an SCC-based naming scheme guarantees uniqueness and preserves reachability relations.
- Surface actions → Self-loops: Guard/action pairs that do not trigger a change of SCC are rendered as self-transitions. This choice avoids an explosion of tiny micro-states and keeps the diagram close to the textual EFSM.
- Control transitions: The set `node.rsdStmts` identifies edges that leave the current SCC; each `(condition, target)` pair is emitted as a connector whose source is `State_<src>` and whose destination is `State_<target>`.
- Entry actions: The current compiler version does not attach code to the entry block of a state. Nevertheless, the `SysMLState` record reserves an `EntryActions` list to accommodate future optimisations (e.g., constant initialisations hoisted out of self-loops).

Filtering Heuristics

Two predicates keep the resulting diagram readable:

LISTING 5.4: Filtering Heuristics

```
1 isInternal(name)        // filters __ell, __tmp variables
2 isAssertAssignment(a)   // filters assignments generated from asserts
```

Both are applied before emitting a transition, ensuring that internal signals never appear in the public SysML facade.

Guard and Action Serialisation

Rather than inventing a custom pretty-printer, the translator relies on `Expr.ToString()` and `Action.ToString()` provided by Averest [12]. Although the output is occasionally verbose, preserving exact formatter output has two advantages: Perfect traceability: A developer can copy a guard from the XML and find it unmodified in the EFSM, thereby avoiding mismatches. The risk of mis-parsing or re-interpreting complex expressions (especially those involving bounded integers) is eliminated.

Visual Layout Heuristic

For human legibility, a grid layout is applied, states are placed four per row at 300 px horizontal and 220 px vertical spacing. The coordinates are computed once and stored in a `Map<string, int*int>` so that the connector generator can anchor arrowheads automatically. When the state machine grows beyond one screen, TTool's built-in "auto-layout" feature can be invoked without conflicting with the translator's positions.

The resulting `SysMLBlock` instance, together with the list of requirements produced earlier, now contains all structural information required for final serialisation: block attributes, states, transitions, guards, actions, and satisfy links.

5.7 Generating Block-level Attributes and Type Information

Before a single state can be generated, the exporter must create a block diagram that declares every variable visible at the top level in the Quartz module. AVATAR encodes such data inside the `<extraparam>` tag of a component of type 5000, one line per attribute [27]. The helper routine `generateAttributesXml` therefore acts as a miniature type-translation layer between Averest and TTool.

The routine starts by iterating over the list of (`qname, Decl`) pairs produced by the parser. For each declaration, it queries `Decl.qtype`, passes the result to `typeCodeOfQType`, and receives back the numeric code that AVATAR expects. A bounded integer such as `int{2}` or `nat{10}` is widened to code 8, which represents an int value in TTool [28]. Booleans become code 4 and exotic constructs that have no AVATAR counterpart fall back to 0, the schema's catch-all value [28]. Because Quartz rarely assigns explicit initial values, the companion function `defaultValueForQType` supplies a neutral literal, `0` for integers and naturals, `false` for Booleans, `0.0` for reals, so that the resulting XML passes TTool's static checker without further annotation.

Each attribute is then assembled as a one-line string,

LISTING 5.5: Block-level Attributes

```
<Attribute access="0" var="0"
           id="credit"
           value="0"
           type="8"
           typeOther=""/>
```

and injected verbatim into the block's `<extraparam>` section. The fixed field `access= "0"` grants public visibility; `var= "0"` states that the element is a plain attribute rather than a signal or port; and `typeOther` is left blank because it becomes relevant only when `type= "0"`.

5.8 Requirement Diagram Synthesis and XML Serialisation

Immediately after gathering assertions from the EFSM, the translator invokes the written function `generateRequirementsXml`, whose task is to write an entire `<Modeling>` section dedicated to AVATAR's requirement diagram. The routine unfolds a rigid, three-element structure for each property: an element reference, a requirement node, and a satisfy connector that binds the two.

The element reference is emitted first as a `COMPONENT type= "5207"` [29]. Its sole purpose is to serve as an anchor; the inner `infoparam` names the behavioural artefact, here always StateMachine, so that, when the diagram is opened in TTool, the connection to the state machine panel is formed. Coordinates in `<cdparam>` are chosen on a simple horizontal grid (150 px increments) so that reference boxes line up neatly above their requirements, but any later drag-and-drop in the GUI preserves the semantic link because connectors target components by their integer `id`, not by geometry.

The requirement node itself follows immediately as `COMPONENT type= "5200"` [29]. Inside its `<extraparam>` block, the translator writes the predicate in full Averest syntax to the `textline` field. The `reqType` is fixed to SafetyRequirement and tinted with TTool's orange shade `color= "-1773070"` [28]. A fresh eight-character hash is copied into both the id attribute and the component's display name, ensuring uniqueness across the diagram. Flags `satisfied= "true"` and

`verified= "true"` are set optimistically. A later formal-analysis pass could toggle them to reflect the actual proof status without affecting the behavioural model.

Finally, a connector of type 5208 materialises the satisfy relation [29]. The `infoparam` field contains the string «`satisfy`», which TTool's renderer interprets as a stereotype and adorns with the correct iconography. The two-way points `<P1>` and `<P2>` reference the numerical IDs of the polyline's endpoints, allowing the editor to redraw the polyline correctly even after the user rearranges the layout. A `cdparam` entry at the geometric midpoint offers TTool an initial bend position; the attribute `<AutomaticDrawing data= "true"/>` instructs the tool to recalculate the shape automatically when either endpoint moves.

Listing 5.6 shows, with all geometry and sizing information intact, the trio of elements generated for the assertion stop → 2 <= exactDist <= 4 in the Robot-02 case study:

LISTING 5.6: Requirement Diagram

```
<COMPONENT type="5207" id="0" uid="97914769-   " index="0">
  <cdparam x="150" y="150"/>
  <sizeparam width="150" height="30"
             minWidth="10" minHeight="30"
             maxWidth="2000" maxHeight="2000"
             minDesiredWidth="107" minDesiredHeight="0"/>
  <infoparam name="AvatarElementReference" value="StateMachine"/>
</COMPONENT>

<!-- Requirement node -->
<COMPONENT type="5200" id="1" uid="159e9944-   " index="1">
  <cdparam x="150" y="220"/>
  <sizeparam width="180" height="70"    />
  <infoparam name="Requirement" value="Req_exactDist_0d424bf3"/>
  <extraparam>
     <textline data="__asrt000 : assert (stop->exactDist<=4&2<=exactDist)"/>
     <kind data="Functional"/>
     <criticality data="Low"/>
     <reqType data="SafetyRequirement" color="-1773070"/>
     <id data="0d424bf3"/>
     <satisfied data="true"/>
     <verified data="true"/>
  </extraparam>
</COMPONENT>

<!--    satisfy    connector -->
<CONNECTOR type="5208" id="1000" uid="a952f1f3-   " index="1000">
  <cdparam x="170" y="200"/>
  <infoparam name="connector" value="&amp;lt;&amp;lt;satisfy&amp;gt;&amp;gt;"/>
  <P1 x="150" y="180" id="0"/>
  <P2 x="170" y="240" id="1"/>
  <AutomaticDrawing data="true"/>
</CONNECTOR>
```

Every numerical attribute shown—`id`, `index`, `x`, `y`, `width`, `height` is mandatory for TTool's parser. In contrast, textual tags (`infoparam`, `textline`, `kind`) carry the semantic payload. By emitting them in the precise order expected by the DTD, the exporter guarantees that the resulting file is not merely well-formed XML but also schema-compliant, loading in TTool without warnings.

5.9 Serialising the Complete Model – "Main XML Generation"

The final stage in the pipeline takes the in-memory `SysMLBlock` structure, already populated with states, transitions, and the list of extracted requirements, and converts it into a single, self-contained XML document. This file must be simultaneously acceptable to TTool, visually intelligible when rendered, and traceable back to every semantic item present in the EFSM. Achieving those three goals requires a disciplined ordering of elements, careful reuse of identifiers, and a handful of layout heuristics that, while purely cosmetic, greatly ease human inspection.

Cohesion between block diagram and state-machine

AVATAR separates data context from behaviour by placing variable declarations, inputs and outputs in a Block Diagram Panel and behavioural logic in a State-Machine Panel. The exporter, therefore, begins the "Design" section by emitting a `<AVATARBlockDiagramPanel>` that contains exactly one component of type 5000. The `infoparam` of that component is deliberately set to the string "StateMachine", which is the same lexical token used as the panel name of the state-machine emitted later in the file. TTool uses this identity to cross-reference diagrams internally; the connection of `AvatarElementReference` in the requirement view causes the editor to zoom straight to the correct block diagram, which in turn contains a hyperlink to the state machine. By choosing a single, stable name at the root of the hierarchy, the translator guarantees that the three views, requirements, data, and behaviour remain coherently linked.

LISTING 5.7: Block Diagram Panel

```
sb.AppendLine("  <Modeling type=\"AVATAR Design\" nameTab=\"QuartzSysML\" tabs=\"
    Block Diagram\">") |> ignore
sb.AppendLine("    <AVATARBlockDiagramPanel name=\"BlockDiagram\" minX=\"10\"
    maxX=\"2500\" minY=\"10\" maxY=\"1500\" zoom=\"1.0\">") |> ignore
sb.AppendLine("       <MainCode value=\"\"/>") |> ignore
sb.AppendLine("       <Optimized value=\"true\"/>") |> ignore
sb.AppendLine("       <considerTimingOperators value=\"true\"/>") |> ignore
sb.AppendLine("       <Validated value=\"StateMachine;\"/>") |> ignore
sb.AppendLine("       <Ignored value=\"\"/>") |> ignore
```

Block Diagram Component and its extended parameters

Inside the component tagged `type="5000"`, the exporter writes a `<sizeparam>` that mirrors TTool's default canvas settings (461 × 358 px). The bounding box is not strictly required for parsing, yet having it present means the diagram opens centred and at a readable zoom. The `<extraparam>` section holds two fixed meta-entries, `<blockType>` and `<CryptoBlock>`, followed by the complete list of `<Attribute>` lines generated earlier. Because every attribute carries the numeric type code and a concrete initial literal, the block diagram is immediately executable in TTool.

LISTING 5.8: Block Diagram Component

```
sb.AppendLine($"      <COMPONENT type=\"5000\" id=\"0\" index=\"0\" uid=\"{
    blockUid}\">") |> ignore
sb.AppendLine("        <cdparam x=\"445\" y=\"19\"/>") |> ignore
sb.AppendLine("        <sizeparam width=\"461\" height=\"358\" minWidth=\"5\"
    minHeight=\"2\" maxWidth=\"2000\" maxHeight=\"2000\" minDesiredWidth=\"0\"
    minDesiredHeight=\"0\"/>") |> ignore
sb.AppendLine("        <hidden value=\"false\"/>") |> ignore
sb.AppendLine("        <cdrectangleparam minX=\"10\" maxX=\"1400\" minY=\"10\"
    maxY=\"900\"/>") |> ignore
```

```
sb.AppendLine($"       <infoparam name=\"block\" value=\"{escapeXml blockName}\"/>") |> ignore
sb.AppendLine("        <new d=\"false\"/>") |> ignore
sb.AppendLine("        <extraparam>") |> ignore
sb.AppendLine("          <blockType data=\"block\" color=\"-4072719\"/>") |> ignore
sb.AppendLine("          <CryptoBlock value=\"false\"/>") |> ignore
sb.AppendLine(generateAttributesXml allVars) |> ignore
sb.AppendLine("        </extraparam>") |> ignore
sb.AppendLine("      </COMPONENT>") |> ignore
sb.AppendLine("    </AVATARBlockDiagramPanel>") |> ignore
```

Emitting state components

Each EFSM node is mapped to a `COMPONENT type= "5106"`. The exporter lays those components on a rectangular grid, four columns wide, with a 300 px column pitch and a 220 px row pitch. While the grid does not replicate the control-flow topology, it prevents states from overlapping and keeps connectors short enough to maintain a legible visual structure. In the long run, a layout algorithm informed by TTool's force-directed engine would be preferable; however, the grid offers an acceptable trade-off for a prototype.

Each state component receives a fresh GUID in its `uid` field, satisfying TTool's requirement that every shape can be addressed unambiguously when diagrams are merged. No entry actions are currently written, but the `<entry>` tag is left available so that future optimisations can be made.

LISTING 5.9: State Machine Component

```
sb.AppendLine($"      <COMPONENT type=\"5106\" id=\"{compId}\" index=\"{compId}\" uid=\"{uid}\">") |> ignore
sb.AppendLine($"        <cdparam x=\"{x}\" y=\"{y}\"/>") |> ignore
sb.AppendLine($"        <sizeparam width=\"250\" height=\"200\" minWidth=\"5\" minHeight=\"2\" maxWidth=\"2000\" maxHeight=\"2000\" minDesiredWidth=\"0\" minDesiredHeight=\"0\"/>") |> ignore
sb.AppendLine($"        <hidden value=\"false\"/>") |> ignore
sb.AppendLine($"        <cdrectangleparam minX=\"10\" maxX=\"2500\" minY=\"10\" maxY=\"1500\"/>") |> ignore
sb.AppendLine($"        <infoparam name=\"state\" value=\"{escapeXml state.Name}\"/>") |> ignore
if not (List.isEmpty state.EntryActions) then
    let entryStr = String.concat "; " (state.EntryActions |> List.map escapeXml)
    sb.AppendLine($"        <entry>{entryStr}</entry>") |> ignore
sb.AppendLine("      </COMPONENT>") |> ignore
```

Geometric logic for connectors

A connector (`type= "5102"`) in AVATAR is essentially a polyline whose two ends must touch any of the twelve numbered connection points pre-computed for every state [29]. Selecting which points to use can influence the number of bends TTool inserts. The exporter, therefore, consults the relative position of source and destination states. If the target lies to the right, the arrow leaves the right-hand midpoint of the source (`x+width`, `y+height/2`) and enters the left-hand midpoint of the target; if the transition is vertical, the top or bottom edge is used likewise. In the special case of a self-loop, the arrow is drawn from mid-right, downwards, and back up, reserving space for a label.

The `<cdparam>` stored inside the connector is purposely set to the geometric midpoint of the segment; this is not strictly necessary for import, but it provides TTool with an initial bend location that

often results in a cleaner route. The attribute `<AutomaticDrawing data= "true"/>` signals to the editor that it may rearrange the polyline automatically when the user drags either state.

Because TTool identifies connection ends by the numeric `id` of the component rather than by its `uid`, the exporter increments a global `connId` counter by ten after every connector. The large stride is deliberate: it leaves space for TTool's run-time indexing, preventing accidental collisions when the file is opened, saved, and reopened. This idea does not visually work yet, but it is not a difficult task once Ttool's behaviour regarding the connectors is understood.

LISTING 5.10: State Machine Connector

```
sb.AppendLine($"      <CONNECTOR type=\"5102\" id=\"{connId}\" index=\"{connId}\"
     uid=\"{uid}\">") |> ignore
sb.AppendLine($"        <cdparam x=\"{midX}\" y=\"{midY}\"/>") |> ignore
sb.AppendLine($"        <sizeparam width=\"0\" height=\"0\" minWidth=\"0\"
    minHeight=\"0\" maxWidth=\"2000\" maxHeight=\"2000\" minDesiredWidth=\"0\"
    minDesiredHeight=\"0\"/>") |> ignore
sb.AppendLine($"        <infoparam name=\"connector\" value=\"null\" from=\"{
    escapeXml trans.From}\" to=\"{escapeXml trans.To}\"/>") |> ignore
sb.AppendLine($"        <TGConnectingPoint num=\"0\" id=\"{connId + 1}\"/>") |>
    ignore
sb.AppendLine($"        <P1 x=\"{p1x}\" y=\"{p1y}\" id=\"{connId + 2}\"/>") |>
    ignore
sb.AppendLine($"        <P2 x=\"{p2x}\" y=\"{p2y}\" id=\"{connId + 3}\"/>") |>
    ignore
sb.AppendLine($"        <AutomaticDrawing data=\"true\"/>") |> ignore
sb.AppendLine($"        <new d=\"false\"/>") |> ignore
sb.AppendLine("      </CONNECTOR>") |> ignore
```

Sub-components as transition carriers

Immediately after each connector, the exporter emits a `SUBCOMPONENT type= "-1"` whose `father id` field points back to the connector's `id` [29]. This nesting tells TTool that the sub-component visually sits on the polyline and carries the semantic payload of the transition. Four `<TGConnectingPoint>` tags are supplied exactly as required by the schema, numbered consecutively after the sub-component's own `id`.

The `<extraparam>` block inside the sub-component contains one line per attribute that the AVATAR metamodel can attach to a transition. Only three lines matter to the translator:

- `<guard...>` receives the raw text of the EFSM guard, wrapped in square brackets to conform to AVATAR syntax.
- `<actions...>` is inserted only when the EFSM transition carries a non-empty update. Omitting the tag when no update exists keeps the XML minimal while remaining valid.
- `<probability...>` is left empty, anticipating future work on probabilistic EFSMs.

Timing-related tags (`afterMin`, `afterMax`, `extraDelay1/2`) are written but left blank; their presence documents the possibility of extension according to the program.

LISTING 5.11: State Machine Subcomponent

```
sb.AppendLine($"      <SUBCOMPONENT type=\"-1\" id=\"{subcompId}\" index=\"{
    connId}\" uid=\"{subUid}\">") |> ignore
sb.AppendLine($"        <father id=\"{connId}\" num=\"0\"/>") |> ignore
...
```

```
4 sb.AppendLine($"        <extraparam>") |> ignore
5  sb.AppendLine($"          <guard value=\"[{escapeXml guardPart}]\" enabled=\"
       true\"/>") |> ignore
6  ...
7  if actionPart <> "" then
8     sb.AppendLine($"          <actions value=\"{escapeXml actionPart}\" enabled
          =\"true\"/>") |> ignore
9 sb.AppendLine($"        </extraparam>") |> ignore
```

Father–child linkage and index coherency

Every state component, connector, and sub-component carries both an `id` and an `index`. Although TTool treats the two numbers synonymously, using both fields allows a quick visual check. `id` is always equal to `index` for components. Still, sub-components share the connector's `index` to verify that the hierarchy is well-formed. During serialisation, the exporter maintains three independent counters (`compId`, `connId`, `subcompId`), ensuring a monotonic increase and thereby compliance with the internal lookup tables of the editor.

Summary of the generation phase

By the time `generateSysMLXml` returns its string, the document contains in strict top–down order: the requirement diagram, the block diagram with attributes, and the fully wired state-machine. Every coordinate and every identifier already satisfies TTool's static checks; loading the file, therefore, produces a syntactically neat project. The modest grid layout is sufficient for diagrams with several dozen states, where aesthetics matter. TTool's auto-layout can refine the visuals without breaking the semantic bindings, as arrows are anchored by component `id` rather than by absolute pixel position. In this way, the exporter produces an artefact.

5.10 Putting It All Together and Concluding Remarks

The final block of the program, beginning with a call to `applyTransformationWithCommit` and ending with a confirmation printed to the console, serves as the binding unit that integrates into one cohesive compilation pipeline. Conceptually, this driver script does little more than thread data through a series of pure functions. However, its sequencing is crucial because each function consumes an artefact whose structure has been meticulously prepared by the preceding step.

The pipeline unfolds in four brisk statements:

LISTING 5.12: Final Statements

```
1 let transformedEFSM = applyTransformationWithCommit completeEFSM
2 let requirements    = extractRequirementsFromEFSM transformedEFSM
3 let efsmNoAssert    = removeAssertActionsFromEFSM transformedEFSM
4 let sysMLModel      = EFSMtoSysML efsmNoAssert
```

First, the EFSM is normalised so that every update is immediate and every `next()` assignment is mirrored by a commit. Second, the same EFSM is mined for assertions, producing a list of `RequirementInfo` records while still in-memory. Third, those assertions are stripped from the behavioural graph, ensuring that the future state machine will not display them as executable code. Finally, the curated EFSM is mapped onto the neutral `SysMLBlock` data structure introduced earlier. Only after the semantic content has been divided cleanly between behaviour and requirement does the exporter call `let sysmlXml = generateSysMLXml sysMLModel requirements`

to produce the textual representation shown in earlier listings. The system is invoked by the written `System.IO.File.WriteAllText` method, which writes the XML to disk, where it can be directly dragged into TTool's workspace.

Closing the Semantic Loop

The translator now completes a closed semantic loop:

1. Quartz grammar → parsed AST (Averest)
2. AST → EFSM (Averest code generator)
3. EFSM → normalised EFSM (two-phase commit)
4. Normalised EFSM → behaviour+requirements (logical split)
5. Behaviour + requirements → SysML/AVATAR XML (serializer)
6. XML → TTool project (end-user artefact)

Each arrow represents either a pure function or a side-effect-free traversal, making the entire compilation chain amenable to future extension and modification. All intermediate artefacts are expressible in F# record syntax and therefore serialisable for regression testing. More importantly, every syntactic construct that the user writes in Quartz variables, threads, `pause` statements, `next()` updates, guards, actions, and `asserts` re-emerges in the SysML document in recognisable form, either as a block attribute, a state, a transition, or a requirement node.

Outlook

The current prototype is already functionally complete for the portion of Quartz used in the evaluation later: it handles parallel loops, synchronous pauses, and inline assertions. The architecture described in this chapter, built around a sequence of small, testable transformations, leaves ample room for growth. Adding support for probabilistic guards, timing constraints, or hierarchical blocks would require only local adjustments, typically an extra field in one record type and a few lines in the serialiser, without risking the correctness guarantees established up to this point.

Thus, Chapter 5 has documented not only what the translator does but also how and why each design decision was taken. The next chapter will demonstrate empirically that the architecture outlined here succeeds in practice, producing SysML artefacts that are both syntactically valid and semantically faithful across a collection of Quartz programs.

6 Evaluation

The purpose of this chapter is to evaluate the correctness and effectiveness of the developed translation tool, which converts models specified in the Quartz language into their corresponding Extended Finite State Machine (EFSM) representations and ultimately into SysML XML files. Also, it aims to provide tangible evidence that the resulting XML files are structurally conformant with the import requirements of TTool and are, at least in principle, ready for downstream activities such as verification or code generation.

This evaluation aims to ensure that the translation process preserves the intended behaviour, structure, and semantics of the original Quartz models at each stage of the transformation pipeline. The testing further seeks to demonstrate that the tool can handle a range of representative examples, including models that incorporate various features such as concurrency, event handling, control flow constructs, and assertions.

Ultimately, the goal is to establish confidence that the toolchain provides a sound and practical solution for bridging the gap between Quartz specifications and SysML-based modelling and verification environments. At the same time, any residual deficiencies, be they semantic, syntactic, or purely cosmetic, will be documented, thereby outlining a clear agenda for future improvement.

6.1 Methodology

The evaluation proceeds in three carefully delineated stages, each corresponding to one step of the translation pipeline. Stage 1 compiles the Quartz source with the Averest NuGet, yielding an EFSM whose nodes represent micro-states of the synchronous execution model and whose edges carry guards and actions exactly as generated by the compiler. Textual statements of the EFSM are retained so that they can later be aligned with the SysML representation.

In Stage 2, the F# tool converts the EFSM into a SysML XML document. The exporter generates two separate views: an AVATAR Requirement Diagram that re-expresses any Quartz `assert` statements, and an AVATAR Design that comprises both a block diagram used to declare variables and their initial values, and a state machine whose structure mirrors that of the EFSM. Every XML element is written by the tool itself; no manual editing is allowed, ensuring that the assessment reflects the tool's native capabilities.

Stage 3 is a cross-representation audit. For the principal test case, the audit is exhaustive: each variable in Quartz must appear as an attribute in the block diagram; each EFSM node must correspond to a state component; and every EFSM transition must be matched by a connector/sub-component pair whose `<guard>` and `<actions>` tags are textual replicas of the compiler output. Temporal semantics are checked by verifying that the `pause` instruction in Quartz becomes the alternation of states in both the EFSM and the SysML machine. Where feasible, the XML is imported into TTool to confirm that the file is syntactically accepted and that the diagram renders without fatal errors. For

the secondary programs, the same checks are applied selectively, and their associated transitions are traced end-to-end, providing confidence that the findings generalise beyond the primary example.

Throughout the process, discrepancies are logged in a structured format and, where required, traced back to a specific stage of the pipeline. This disciplined, artefact-centric methodology not only demonstrates the present correctness of the tool but also establishes a reproducible framework for future regression tests as the translator and the surrounding toolchain evolve.

6.2 Test Case Selection

The test cases used for evaluating the translation tool were carefully selected from the Quartz example database. The primary motivation behind this selection strategy is to facilitate a focused and manageable testing environment during the initial development phase of the translation program. Specifically, the examples were chosen to avoid those models that involve hierarchical module structures, in favour of self-contained, single-program models. This choice allows for a more straightforward mapping between Quartz and SysML, making it easier to analyse and compare the semantics of both languages in a controlled setting.

Moreover, priority was given to examples that incorporate fundamental programming constructs such as loops, conditional statements, and parallelism. These features are essential in assessing the translation tool's ability to accurately represent concurrency, event synchronisation, and data manipulation in the target SysML format. By focusing on models that showcase parallel threads, synchronisation through events, as well as conditional branching and iterative control flow, the evaluation can directly address the most critical challenges associated with behavioural preservation during translation.

Each model has been chosen not only for its relevance in demonstrating essential control and data flow patterns, but also for its clarity and simplicity, which are vital for effective manual comparison and semantic analysis. This approach ensures that the translation results can be easily interpreted and validated, and also provides valuable insight into the fundamental compatibility of Quartz and SysML state machine semantics.

By grounding the test suite in well-understood examples from the Quartz repository and emphasising core language features, the evaluation can systematically explore the strengths and limitations of the translation process, while setting a clear foundation for future work involving more advanced or hierarchical models.

6.3 Results

The Robot-02 example, from the Quartz database [10], is initially used to evaluate the translator. Although modest in size, the program encompasses a remarkable diversity of language features, including three parallel threads, guarded updates, a non-trivial sensor-error model, immediate event emission, a temporal property expressed as an assertion, and a liveness requirement. Its richness ensures that every stage of the toolchain —Averest compilation, EFSM extraction, requirement extraction, block synthesis, and state-machine generation —is exercised in a realistic setting.

6.3.1 Quartz Specifications

Listing 6.1 reproduces the exact source that feeds the pipeline. A hidden variable, `exactDist`, records the actual distance between the robot and an obstacle. The first thread increments that distance

whenever the push command is received; it also checks, by assertion, that stop is never raised unless `exactDist` lies between 2 and 4. The second thread models a sensor that reports `sensedDist` with a bounded error of ±1, while the third thread constitutes the control logic, which emits stop as soon as the sensed distance is believed to be safe.

LISTING 6.1: Quartz: Robot02

```
module Robot02(event ?push, int{2} ?err, event stop) {
    int{3} sensedDist; // distance determined by vehicle's sensor

    // note that exactDist is made local and therefore hidden to the vehicle
    {int{3} exactDist;
        // this thread maintains the exact distance
        loop {
            if(push & !stop)
                next(exactDist) = exactDist + 1;
            assert(stop -> 2<=exactDist & exactDist<=4);
            w1: pause;
        }
     || // this thread models the sensor which determines sensedDist with error
        loop {
            case
                (err>0) do sensedDist = exactDist + 1;
                (err<0) do sensedDist = exactDist - 1;
            default
                sensedDist = exactDist;
            w2: pause;
        }
    } // end of scope of exactDist
     || // This thread models the vehicle's program whose aim is to emit stop
        // when it is sure that 2<=exactDist<=4 holds only based on sensedDist.
        // The program below is not correct (see driver t1)
        loop {
            if(sensedDist>=3)
                immediate always{ emit(stop); }
            w3: pause;
        }
}
satisfies {
    goal1: assert A F stop;
}
```

6.3.2 Extended Finite-State Machine

Figure 6.1 shows the EFSM obtained from Averest [12]. Three non-instantaneous nodes are sufficient to encode the combined behaviour of the three threads, while a fourth, instantaneous node represents inactivity after the stop has been emitted. Within state 0, the compiler has flattened all guarded assignments of the w1 and w2 threads; the self-loop labelled `<!(3<=sensedDist) ==> ...>` stems directly from the guard in the w3 thread. Assertions from Listing 6.1 are preserved directly, for example `__rteAtLine17002 : assert exactDist-1<3-3<=exactDist+1`, which checks the sensor error bounds.

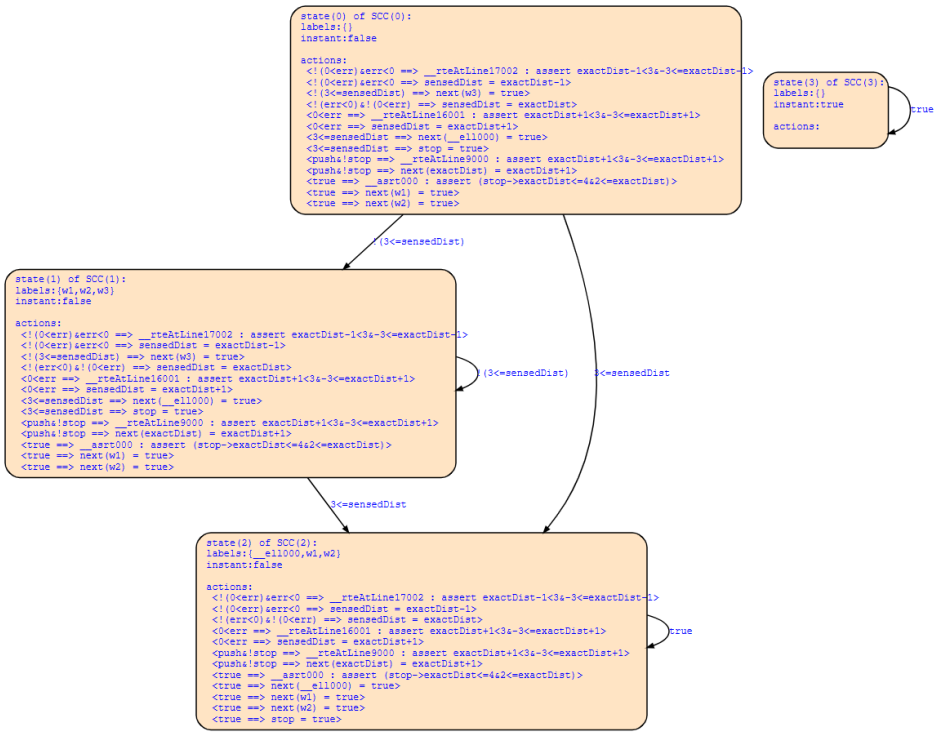

FIGURE 6.1: EFSM: Robot02 [30]

6.3.3 XML output

The XML produced by the translator now contains both a requirement-level view and a behavioural design.

Requirement diagram: The first `<Modeling>` section introduces a requirement node whose textline element re-states the safety claim "`stop → 2 <= exactDist <= 4`" in formal syntax. A «satisfy» connector attaches it to an AvatarElementReference that designates the state-machine, thereby making the proof obligation explicit at the model level.

Block diagram: The block declares three attributes, `push`, `err`, and `stop`, whose names and initial values coincide with those in the Quartz header.

State-machine: Each EFSM node is rendered as a `type="5106"` component, while every edge becomes a connector/sub-component pair. Listing 6.2 shows one such pair, corresponding to the negated guard `!(3 <= sensedDist)` and the assignment for label `w3`.

LISTING 6.2: Fragment of XML

```
<COMPONENT type="5200" id="1" ...>
  <cdparam x="150" y="220"/>
  <infoparam name="Requirement" value="Req___asrt000_0d424bf3"/>
  <extraparam>
     <textline   data="__asrt000 : assert (stop->exactDist<=4&2<=exactDist)"/>
     <reqType    data="SafetyRequirement"/>
     <criticality data="Low"/>
     <satisfied  data="true"/>
  </extraparam>
```

```
</COMPONENT>

<CONNECTOR type="5208" id="1000" ...>
  <infoparam name="connector" value="&lt;&lt;satisfy&gt;&gt;"/>
  <P1 x="150" y="180" id="0"/>
  <P2 x="170" y="240" id="1"/>
</CONNECTOR>

<COMPONENT type="5000" id="0" ...>
  <infoparam name="block" value="StateMachine"/>
  <extraparam>
     <Attribute id="push"  type="4" value="false"/>
     <Attribute id="err"   type="8" value="0"/>
     <Attribute id="stop"  type="4" value="false"/>
  </extraparam>
</COMPONENT>

<COMPONENT type="5106" id="0" ...>
  <cdparam x="100" y="100"/>
  <infoparam name="state" value="State_0"/>
</COMPONENT>

<CONNECTOR type="5102" id="60" ...>
  <infoparam name="connector" value="null" from="State_0" to="State_0"/>
  <P1 x="350" y="200" id="62"/>
  <P2 x="410" y="260" id="63"/>
</CONNECTOR>

<SUBCOMPONENT type="-1" id="10060" ...>
  <father id="60" num="0"/>
  <extraparam>
     <guard   value="[!(3<=sensedDist)]" enabled="true"/>
     <actions value="w3_next = true"     enabled="true"/>
  </extraparam>
</SUBCOMPONENT>
```

Because the translator is currently unable to guarantee a graphical anchor by state identifier, connectors are referenced by numeric connection points. When the file is opened in TTool, several arrows require manual realignment. The semantic payload, the guard, and the action remain intact nonetheless.

6.3.4 Observations

A stepwise check of the artefacts generated for the Robot-02 example confirms that the translator carries forward nearly every semantic detail of the original Quartz program.

Preservation of data context: All externally visible variables of the Quartz header `push`, `err`, and `stop` re-appear in the block description (Listing 6.2). Quartz distinguishes `int{2}` from an unbounded integer, whereas AVATAR offers only a generic signed type (`type= "8"`).

Guards and actions: Every conditional appearing in the EFSM text, whether an inequality (`err<0`), a conjunction (push & !stop), or a negation (`!(3<=sensedDist)`), is reproduced identically in the `<guard>` field of the corresponding SysML transition. Likewise, each state update becomes a one-line `<actions>` entry. The fragment shown in Listing 6.2 illustrates a typical pattern: the guard

[!(3<=sensedDist)] and the assignment w3_next = true match the EFSM that originates in the w3 thread.

Temporal semantics of pause: The synchronous-reactive clock of Quartz manifests itself in the EFSM as an alternation between "pre-pause" and "post-pause" states, and the translator keeps that alternation intact in the SysML machine. Instantaneous EFSM nodes, used to model the immediate always emission of stop, are mapped to SysML states whose instant= "true" attribute forces zero-time execution.

Requirement extraction and traceability: The compiler generates five distinct safety assertions, one per syntactic occurrence of assert, and the exporter converts each into its own SafetyRequirement node (Listing 6.2). Every node is linked to the behavioural model by a «satisfy» connector, so the obligation is propagated automatically to subsequent analysis stages. Although the requirement text remains in formal notation, its direct lineage from the Quartz source is explicit.

Remaining limitations

These examples have revealed four further limitations that, while not blocking, will require dedicated work before the exporter can claim full language coverage and industrial-grade readability.

Bounded-integer fidelity: Quartz allows the programmer to declare fine-grained bit-widths such as int2 or nat10. The current translator widens every such type to the generic signed code 8 mandated by AVATAR. Functionally, this causes no misbehaviour; the generated state machine never produces values outside the original range, but it still represents an information loss. Bit-width constraints often provide immediate insight into resource usage and can drive downstream optimisations (for instance, the choice of a micro-controller register class). A principled solution would either extend AVATAR's metamodel with a "sub-range" attribute or encode the bound as an explicit invariant in the requirement diagram and link it back to each arithmetic update. Both directions are feasible, yet neither is implemented in the present prototype.

Natural-language presentation of requirements: The exporter faithfully copies each Quartz assert predicate into the <textline> field of a SafetyRequirement. While this guarantees semantic precision, it leaves the requirement in formal Averest syntax replete with logical connectives, infix inequalities, and, at times, auxiliary variables such as __rteAtLine.... Typical SysML practice, however, expects requirements to be phrased in controlled natural language for validation. Bridging that gap calls for an additional layer: either a pattern library that paraphrases common relational forms. Until such a component is integrated, the requirements panel will remain formally correct yet less approachable.

Ordering inside the States: The first is the execution ordering of multiple next updates. Inside a single EFSM state, the compiler may emit several updates to disjoint variables next(exactDist), next(w1), next(w2), and so on. It is not yet clear if AVATAR executes all actions attached to one transition atomically or offers any semantics for ordering across distinct transitions enabled by the same guard. As a result, the precise chronology of updates within a logical tick is implicit rather than explicit in the XML. While no counter-example has been observed in practice, a future version of the exporter should either serialise the updates into one composite action or introduce intermediate micro-states to make the order unambiguous.

Graphical anchoring of connectors: In the current implementation, transition arrows are attached to states via numerical connection points (P1, P2) rather than by state identifiers. Ttool imports the file without semantic error, but its layout engine misplaces the arrows, giving the appearance of

disconnected edges. The defect is purely cosmetic; simulation still adheres to the encoded guards and actions, yet it hampers manual inspection.

Despite these caveats, the Robot-02 example discussed, as well as other examples, demonstrate that the translator preserves guarded behaviour, synchronous timing, variable initialisation, local-state semantics, and user-defined safety properties. The resulting XML loads in TTool; the requirement diagram is automatically linked to the behaviour, and all computational steps visible in the EFSM can be located directly in the SysML state machine. The evidence, therefore, supports the claim that the prototype provides a reliable and semantically faithful bridge from Quartz specifications to SysML/AVATAR models, a prerequisite for any subsequent model-driven analysis or code-generation workflow.

6.4 Confirmation

In addition to the in-depth Robot-02 investigation, the translator was exercised on six further Quartz examples from [10]: VendingMachine, EuclidMod, EuclidSub, EulerNumber, Robot01, and SquareRoot. Each model was subjected to the identical three-stage pipeline described in Section 6.3: (i) compilation with Averest to obtain an EFSM, (ii) automated export to SysML/AVATAR XML, and (iii) cross-representation auditing of variables, states, transitions, and safety assertions.

Although the individual programs differ widely, the outcome was uniform. In every case, all Quartz variables reappeared as AVATAR block attributes with the correct basic type; every EFSM node was matched by a `type= "5106"` state component; and each guarded assignment was transferred directly into the `<guard>` and `<actions>` tags of a SysML transition. When assertions were present, the exporter generated a `SafetyRequirement` node whose `textline` reproduced the original predicate and whose «`satisfy`» link correctly targeted the enclosing state-machine.

Minor shortcomings paralleled those already noted for Robot-02. Diagrams required manual realignment of connectors because arrows were bound to numerical points rather than to explicit state identifiers, and the chronological ordering of multiple `next` updates inside a single logical tick remains implicit in the XML. Critically, however, none of the supplementary examples revealed a deviation in functional semantics: every event, guard, action, and assertion visible in the EFSM could be traced one-to-one into the SysML artefact.

The consistency of these results across seven different programs provides strong evidence that the prototype translator is not merely correct for a single showcase example but operates robustly over a representative section of the Quartz language. [4]

7 Conclusion and Future Work

This thesis aimed to address a pragmatic gap that emerged in the model-based development of reactive and embedded systems. Designers who adopt Averest and its synchronous language Quartz obtain a mathematically rigorous platform for functional verification [4]. Designers who rely on TTool and its SysML/AVATAR profile gain integrated facilities for architectural exploration, schedulability analysis, and, crucially, security verification through the ProVerif back-end [17]. Until now, models from these frameworks could be manually rewritten from one notation to another, but no automated, semantics-preserving route existed. The core objective of the thesis was therefore twofold. First, to determine, independently of any implementation, whether a faithful translation from Quartz to SysML is possible in principle. Second, to demonstrate that feasibility by building and evaluating a prototype translator that emits TTool-compatible XML.

This thesis encompasses a range of information, beginning with an understanding of the syntax and semantics of Quartz and SysML, which serve as the reference against which the correctness of any transformation must be judged. Then the capabilities of the two tool suites are surveyed, revealing an apparent complementarity: Averest excels at synchronous functional proof, while TTool contributes architectural, performance and security viewpoints.

Further, the conceptual kernel of the thesis is delivered: the translation feasibility analysis. It showed that perfect synchrony in Quartz can be embedded in UML's run-to-completion semantics by mapping each logical tick to a synthetic event cycle; that control-flow constructs, data types, concurrency and formal assertions each appreciate a sound counterpart in SysML; and that the remaining mismatches (fixed-width integers, broadcast semantics across blocks, explicit tick ordering) are tractable through disciplined modelling patterns or modest tooling extensions.

Guided by that analysis, a prototype translator is designed and implemented in F#. The program parses a Quartz module, constructs the extended finite-state machine produced by the Averest compiler, and serialises the result into a SysML XML file accepted by TTool. The generator handles delayed assignments, synchronous parallelism, explicit requirements and basic data attributes; it injects auxiliary slots (x_next) to preserve tick-delayed visibility and synthesises a requirements diagram so that Quartz assertions appear as «`satisfy`» relationships in AVATAR.

Then the translator is evaluated on a representative set of Quartz programs, including vending machine controllers, Euclidean algorithms, and robotic sensing loops. For each example, the generated XML could be imported into TTool. Functional traces matched those produced by Averest; requirements were present and traceable. The study also identified four shortcomings: the absence of native bounded integers, implicit ordering of multiple `next` commits, cosmetic misplacement of connectors, and the fact that exported requirements remain in formal rather than natural language. These deficiencies do not undermine semantic soundness, yet they mark the frontier between a research prototype and a production-quality bridge.

The work confirms that an unbridgeable semantic divide does not separate synchronous and SysML models. With careful mapping of ticks to run-to-completion cycles, of events to signal-plus-Boolean conventions, and of temporal assertions to requirement elements, a designer can migrate executable behaviour, test benches and even proof obligations from Averest to TTool without rewriting or manual synchronisation. Conversely, architectures elaborated in SysML, such as partitioning and deployment, as well as cryptographic protocols, can be analysed for confidentiality and authenticity without forfeiting the high-confidence functional kernel originally verified in Quartz. The prototype translator, with further modifications, therefore, establishes a round-trip path that allows each tool to contribute its strengths to a single development artefact.

Future Work

Several avenues require continuation:

1. Native support for fixed-width and modular arithmetic:
   Integrating a ValueType sub-profile for bounded integers into AVATAR, together with code-generation templates that enforce wrap-around operations, would remove the need for external OCL constraints and make the translation more transparent to analysts unfamiliar with synchronous theory.

2. Broadcast-discipline contracts:
   Extending AVATAR with a "single-producer-per-tick" constraint block, auto-generated by the translator, would guard against accidental re-emission of signals by other blocks and thus preserve Quartz's broadcast semantics even in large, multi-block models.

3. Reverse translation and co-evolution:
   While the thesis concentrated on the forward direction, industrial practice benefits from both frameworks. Developing a SysML-to-Quartz back-map, at least for the synchronous subset, would enable iterative refinement where security insights lead to functional changes and vice versa.

4. Proof-transformation theorems:
   A formal study relating proofs carried out by the Averest model-checker to those conducted by TTool's ProVerif gateway could supply mathematical assurance that security and functional properties survive architectural elaboration.

5. Usability and visual fidelity:
   Fine-tuning connector anchoring, layout heuristics, and naming conventions would transform the prototype into a tool that practitioners can adopt without requiring manual post-editing of diagrams.

Bridging specification languages is often portrayed as a peripheral engineering task. However, it exercises the very core of semantics: only when two notations can exchange models loss-lessly do we discover how well their conceptual foundations align. This thesis has demonstrated that the synchronous discipline of Quartz and the event-driven richness of SysML are sufficiently coherent to allow for such a bridge, and that doing so multiplies the analytical power available to designers. By coupling Averest's exhaustive functional verification with TTool's architectural and security toolchain, the work provides a step toward an integrated model-driven methodology.

A Code

LISTING A.1: Complete Code

```
// ================== Quartz to TTool SysML XML Translator ==================
// This program parses a Quartz (Averest) program, generates its EFSM,
// and exports everything as a TTool-compatible XML file.
// ==========================================================================


open Averest.Quartz
open Averest.Quartz.Statements
open Averest.TeachingTools
open Averest.TeachingTools.Quartz
open Averest.Core.Expressions
open Averest.Core.Names
open Averest.Core.Actions
open Averest.Core.Types
open Averest.Core
open Averest.Core.Declarations
open System.Text
open System.Security



// -------------------------- File and Parsing ------------------------------

// Path to the Quartz file (change as needed)
let quartzFile = @"E:\Master's Study Material\Thesis\Quartz Programs\Robot02.qrz"

// Parse the Quartz module from the file
let parsedModule = Compiler.ParseQuartzModuleFromFile quartzFile
let mainStmt = parsedModule.declStmt.stmt

// Compile the module into an Extended Finite State Machine (EFSM)
let withControlFlow = true
let efsm = Quartz.ComputeStateTrans withControlFlow mainStmt

// Add a sink (terminal) state to the EFSM
let completeEFSM = Quartz.AddSinkState efsm



// ---------------------- EFSM Transformation Utilities ----------------------

// Helper: produce the "x_next" version of a variable (for next() semantics)
let makeNextQName (qname: QName) =
    match qname with
    | BasicName idx ->
        let originalName = Names.index2NameTable[idx]
```

```
47          let nextName = originalName + "_next"
48          let nextIdx = Names.InsertName nextName
49          BasicName nextIdx
50      | _ -> failwith "Unsupported QName format for 'next' transformation"
51
52  // Helper: wrap a QName and type as a left-hand-side variable
53  let makeLhsVar (qname: QName) (qtype: QType) : LhsExpr =
54      LhsVar(qname, qtype)
55
56  // Transform AssignNxt to AssignNow with *_next variables, excluding Asserts
57  let transformEFSMWithCommit (efsm: NodeOfEFSM array) =
58      efsm |> Array.map (fun node ->
59          let transformedActs =
60              node.surfActs
61              |> Set.map (fun (guard, action) ->
62                  match action with
63                  // Convert next-step assignments to _next variables
64                  | AssignNxt(LhsVar(qname, qtype), _, expr) ->
65                      let nextName = makeNextQName qname
66                      let newLhs = makeLhsVar nextName qtype
67                      (guard, AssignNow(newLhs, qtype, expr))
68                      // Leave other actions (including Assert) untouched
69                  | _ -> (guard, action)
70              )
71          { node with surfActs = transformedActs }
72      )
73
74
75  // Add commit steps (copy *_next vars back to their base vars)
76  // This enables two-phase state variable updates (Quartz model semantics)
77  let commitAssignments (efsm: NodeOfEFSM array) =
78      // Find all assignments to *_next vars in the EFSM
79      let assignments =
80          efsm
81          |> Array.collect (fun node ->
82              node.surfActs
83              |> Set.toArray
84              |> Array.choose (fun (_, action) ->
85                  match action with
86                  // For each assignment to a _next variable, generate a "commit"
                        action
87                  | AssignNow(LhsVar(qname, qtype), _, _) when
88                        qname.ToString().EndsWith("_next") ->
89                      let originalNameStr = qname.ToString().Replace("_next", "")
90                      let originalQName =
91                          if Names.name2IndexTable.ContainsKey originalNameStr then
92                              BasicName(Names.IndexOfName originalNameStr)
93                          else
94                              // Insert the base name if it doesn't exist
95                              let idx = Names.InsertName originalNameStr
96                              BasicName idx
97                      let lhs = makeLhsVar originalQName qtype
98                      let rhs = MkVarExpr qname qtype
99                      Some(BoolConst true, AssignNow(lhs, qtype, rhs))
100                 | _ -> None
101             )
102         )
103         |> Set.ofArray
```

```
104
105     // Add all commit actions to each EFSM node
106     efsm |> Array.map (fun node ->
107         let newSurfActs = Set.union node.surfActs assignments
108         { node with surfActs = newSurfActs }
109     )
110
111 // Apply the above transformations in sequence
112 let applyTransformationWithCommit (efsm: NodeOfEFSM array) =
113     efsm
114     |> transformEFSMWithCommit
115     |> commitAssignments
116
117
118
119 // ---------------------- SysML Structure Types ------------------------------
120
121 // Represents a single state in the SysML state machine
122 type SysMLState = {
123     Name: string
124     EntryActions: string list
125 }
126
127 // Represents a single transition (edge) between SysML states
128 type SysMLTransition = {
129     From: string
130     To: string
131     Guard: string
132     Action: string
133 }
134
135 // Represents a block (the overall SysML state machine)
136 type SysMLBlock = {
137     Name: string
138     States: SysMLState list
139     Transitions: SysMLTransition list
140 }
141
142 // Represents a connection between SysML blocks (not used in this script)
143 type SysMLConnection = {
144     FromBlock: string
145     ToBlock: string
146     Signal: string
147 }
148
149
150
151 // -------------- Requirements Extraction and Filtering Utilities --------------
152
153 // Info required to fill a TTool SysML Requirement block
154 type RequirementInfo = {
155     Id: string
156     Description: string
157     Kind: string
158     Criticality: string
159     ReqType: string
160     Satisfied: bool
161     Verified: bool
```

```
    ElementName: string
}

// Extract all Assert actions in the EFSM as requirements
let extractRequirementsFromEFSM (efsm: NodeOfEFSM array) : RequirementInfo list =
    efsm
    |> Array.collect (fun node ->
        node.surfActs
        |> Set.toArray
        |> Array.choose (fun (guard, action) ->
            match action with
            | Assert(qname, expr) ->
                let assertionStr = sprintf "%s : assert %s" (qname.ToString()) (
                    expr.ToString())
                Some {
                    Id = System.Guid.NewGuid().ToString().Substring(0, 8)
                    Description = assertionStr
                    Kind = "Functional"
                    Criticality = "Low"
                    ReqType = "SafetyRequirement"
                    Satisfied = true
                    Verified = true
                    ElementName = qname.ToString()
                }
            | _ -> None
        )
    )
    |> Array.toList


// Remove all Assert actions from EFSM (so they are not shown as state machine
    actions)
let removeAssertActionsFromEFSM (efsm: NodeOfEFSM array) =
    efsm
    |> Array.map (fun node ->
        { node with
            surfActs =
                node.surfActs
                |> Set.filter (fun (_, action) ->
                    match action with
                    | Assert(_, _) -> false
                    | _ -> true
                )
        }
    )



// ---------------------- EFSM to SysML State Machine ------------------------

// Convert the EFSM to a SysMLBlock for code generation
let EFSMtoSysML (efsm: NodeOfEFSM array) : SysMLBlock =
    // Helper: skip internal state and temp variables
    let isInternal (name: string) =
        name.StartsWith("__ell") || (name.StartsWith("__") && name.EndsWith("
            _next"))

```

```
216     // Helper: check if an action is an assertion or a runtime assertion
            assignment
217     let isAssertAssignment action =
218         match action with
219         | AssignNow(LhsVar(qn, _), _, _) when qn.ToString().StartsWith("
                __rteAtLine") -> true
220         | Assert(_, _) -> true
221         | _ -> false
222
223     // Generate all state nodes
224     let allStates =
225         efsm
226         |> Array.map (fun node ->
227             {
228                 Name = sprintf "State_%d" node.sccIndex
229                 EntryActions = []
230             }
231         )
232         |> Array.toList
233
234     // Generate all transitions between states
235     let allTransitions =
236         efsm
237         |> Array.toList
238         |> List.collect (fun node ->
239             let thisState = sprintf "State_%d" node.sccIndex
240
241             // For each explicit guarded assignment (not assertion or internal),
                    add a transition
242             let guardedTransitions =
243                 node.surfActs
244                 |> Set.toList
245                 |> List.choose (fun (guard, action) ->
246                     if isAssertAssignment action then None else
247                     match action with
248                     | AssignNow(LhsVar(qn, _), _, _) when isInternal (qn.ToString
                            ()) -> None
249                     | _ ->
250                         Some {
251                             From = thisState
252                             To = thisState
253                             Guard = guard.ToString()
254                             Action = action.ToString()
255                         }
256                 )
257
258             // Add EFSM control transitions (e.g. state-to-state jumps)
259             let controlTransitions =
260                 node.rsdStmts
261                 |> Set.toList
262                 |> List.map (fun (cond, targetNodeId) ->
263                     {
264                         From = thisState
265                         To = sprintf "State_%d" targetNodeId
266                         Guard = cond.ToString()
267                         Action = ""
268                     }
269                 )
```

```
            guardedTransitions @ controlTransitions
        )

    {
        Name = "QuartzToSysMLStateMachine"
        States = allStates
        Transitions = allTransitions
    }



// ------------------- Helpers for SysML Attribute/Type XML -------------------

// Type code mapping for TTool XML attributes
let typeCodeOfQType (qt: QType) =
    match qt with
    | Qint _ -> 8
    | Qbool -> 4
    | Qreal -> 10 // Not checked
    | Qbtv _ | Qnat _ -> 8 // Not checked
    | _ -> 0

// Default value for each type, for XML initialization
let defaultValueForQType (qt: QType) =
    match qt with
    | Qint _ | Qnat _ | Qbtv _ -> "0"
    | Qbool -> "false"
    | Qreal -> "0.0"
    | _ -> ""

// Generate attributes XML for the block diagram section
let generateAttributesXml (decls: (string * Decl) list) =
    decls
    |> List.map (fun (name, decl) ->
        let typeCode = typeCodeOfQType decl.qtype
        let initVal = defaultValueForQType decl.qtype
        sprintf "<Attribute access=\"0\" var=\"0\" id=\"%s\" value=\"%s\" type
            =\"%d\" typeOther=\"\"/>"
                name initVal typeCode
    )
    |> String.concat "\n"

// Escape XML special characters
let escapeXml (input: string) =
    SecurityElement.Escape(input)



// ------------------- Requirements XML Generation ----------------------------

// Write the Avatar Requirement section for all extracted requirements
let generateRequirementsXml (requirements: RequirementInfo list) =
    let sb = StringBuilder()
    sb.AppendLine("  <Modeling type=\"Avatar Requirement\" nameTab=\"AVATAR
        Requirements\">") |> ignore
    sb.AppendLine("    <AvatarRDPanel name=\"AVATAR RD\" minX=\"10\" maxX
        =\"1900\" minY=\"10\" maxY=\"1400\" zoom=\"1.0\">") |> ignore

```

```
325     let mutable compId = 0
326     let mutable connectorId = 1000
327
328     for req in requirements do
329         let refUid = System.Guid.NewGuid().ToString()
330         let reqUid = System.Guid.NewGuid().ToString()
331         let satisfyUid = System.Guid.NewGuid().ToString()
332
333         // Element Reference Block
334         sb.AppendLine($"""      <COMPONENT type="5207" id="{compId}" index="{
                compId}" uid="{refUid}">""") |> ignore
335         sb.AppendLine($"""        <cdparam x="{150 + compId*100}" y="150"/>""")
                |> ignore
336         sb.AppendLine($"""        <sizeparam width="150" height="30" minWidth
                ="10" minHeight="30" maxWidth="2000" maxHeight="2000" minDesiredWidth
                ="107" minDesiredHeight="0"/>""") |> ignore
337         sb.AppendLine($"""        <hidden value="false"/>""") |> ignore
338         sb.AppendLine($"""        <cdrectangleparam minX="10" maxX="1900" minY
                ="10" maxY="1400"/>""") |> ignore
339         sb.AppendLine($"""        <infoparam name="AvatarElementReference" value
                ="StateMachine"/>""") |> ignore
340         sb.AppendLine($"""        <new d="false"/>""") |> ignore
341         sb.AppendLine("      </COMPONENT>") |> ignore
342
343         // Requirement Block
344         sb.AppendLine($"""      <COMPONENT type="5200" id="{compId + 1}" index="{
                compId + 1}" uid="{reqUid}">""") |> ignore
345         sb.AppendLine($"""        <cdparam x="{150 + compId*100}" y="220"/>""")
                |> ignore
346         sb.AppendLine($"""        <sizeparam width="180" height="70" minWidth="1"
                 minHeight="30" maxWidth="2000" maxHeight="2000" minDesiredWidth
                ="136" minDesiredHeight="0"/>""") |> ignore
347         sb.AppendLine($"""        <hidden value="false"/>""") |> ignore
348         sb.AppendLine($"""        <cdrectangleparam minX="10" maxX="1900" minY
                ="10" maxY="1400"/>""") |> ignore
349         sb.AppendLine($"""        <infoparam name="Requirement" value="Req_{req.
                ElementName}_{req.Id}"/>""") |> ignore
350         sb.AppendLine($"""        <new d="false"/>""") |> ignore
351         sb.AppendLine($"""        <extraparam>""") |> ignore
352         sb.AppendLine($"""          <textline data="{escapeXml req.Description
                }"/>""") |> ignore
353         sb.AppendLine($"""          <kind data="{req.Kind}"/>""") |> ignore
354         sb.AppendLine($"""          <criticality data="{req.Criticality}"/>""")
                |> ignore
355         sb.AppendLine($"""          <reqType data="{req.ReqType}" color
                ="-1773070"/>""") |> ignore
356         sb.AppendLine($"""          <id data="{req.Id}"/>""") |> ignore
357         sb.AppendLine($"""          <satisfied data="{req.Satisfied.ToString().
                ToLower()}"/>""") |> ignore
358         sb.AppendLine($"""          <verified data="{req.Verified.ToString().
                ToLower()}"/>""") |> ignore
359         sb.AppendLine($"""        </extraparam>""") |> ignore
360         sb.AppendLine("      </COMPONENT>") |> ignore
361
362         // Satisfy Connector
363         sb.AppendLine($"""      <CONNECTOR type="5208" id="{connectorId}" index
                ="{connectorId}" uid="{satisfyUid}">""") |> ignore
```

```
364         sb.AppendLine($"""        <cdparam x="{150 + compId*100 + 20}" y
                ="200"/>""") |> ignore
365         sb.AppendLine($"""        <sizeparam width="0" height="0" minWidth="0"
                minHeight="0" maxWidth="2000" maxHeight="2000" minDesiredWidth="0"
                minDesiredHeight="0"/>""") |> ignore
366         sb.AppendLine($"""        <infoparam name="connector" value="&lt;&lt;
                satisfy&gt;&gt;"/>""") |> ignore
367         sb.AppendLine($"""        <P1 x="{150 + compId*100}" y="180" id="{compId
                }"/>""") |> ignore
368         sb.AppendLine($"""        <P2 x="{150 + compId*100 + 20}" y="240" id="{
                compId + 1}"/>""") |> ignore
369         sb.AppendLine($"""        <AutomaticDrawing data="true"/>""") |> ignore
370         sb.AppendLine($"""        <new d="false"/>""") |> ignore
371         sb.AppendLine("      </CONNECTOR>") |> ignore
372
373         compId <- compId + 2
374         connectorId <- connectorId + 1
375     sb.AppendLine("    </AvatarRDPanel>") |> ignore
376     sb.AppendLine("  </Modeling>") |> ignore
377     sb.ToString()
378
379
380
381 // ------------------------- Main XML Generation -----------------------------
382
383 // Generate the full SysML XML including requirements, block diagram, and state
        machine
384 let generateSysMLXml (sysmlModel: SysMLBlock) (requirements: RequirementInfo list
        ) =
385     let sb = StringBuilder()
386     sb.AppendLine("<?xml version=\"1.0\" encoding=\"UTF-8\"?>") |> ignore
387     sb.AppendLine("<TURTLEGMODELING version=\"1.0 beta\"
            ANIMATE_INTERACTIVE_SIMULATION=\"false\">") |> ignore
388
389     // Add requirements section first (if any)
390     if not (List.isEmpty requirements) then
391         sb.AppendLine(generateRequirementsXml requirements) |> ignore
392
393     // Block Diagram Panel for the main state machine block
394     sb.AppendLine("  <Modeling type=\"AVATAR Design\" nameTab=\"QuartzSysML\"
            tabs=\"Block Diagram\">") |> ignore
395     sb.AppendLine("    <AVATARBlockDiagramPanel name=\"BlockDiagram\" minX=\"10\"
             maxX=\"2500\" minY=\"10\" maxY=\"1500\" zoom=\"1.0\">") |> ignore
396     sb.AppendLine("       <MainCode value=\"void __user_init() {\"/>") |> ignore
397     sb.AppendLine("       <MainCode value=\"}\"/>") |> ignore
398     sb.AppendLine("       <Optimized value=\"true\"/>") |> ignore
399     sb.AppendLine("       <considerTimingOperators value=\"true\"/>") |> ignore
400     sb.AppendLine("       <Validated value=\"StateMachine;\"/>") |> ignore
401     sb.AppendLine("       <Ignored value=\"\"/>") |> ignore
402     let blockName = "StateMachine" // Considered
403     let allVars: (string * Decl) list =
404         parsedModule.declStmt.decL
405         |> Seq.toList
406         |> List.map (fun (qname, decl) -> (qname.ToString(), decl))
407     let blockUid = System.Guid.NewGuid().ToString()
408     sb.AppendLine($"      <COMPONENT type=\"5000\" id=\"0\" index=\"0\" uid=\"{
            blockUid}\">") |> ignore
409     sb.AppendLine("        <cdparam x=\"445\" y=\"19\"/>") |> ignore
```

```
410     sb.AppendLine("        <sizeparam width=\"461\" height=\"358\" minWidth=\"5\"
            minHeight=\"2\" maxWidth=\"2000\" maxHeight=\"2000\" minDesiredWidth
            =\"0\" minDesiredHeight=\"0\"/>") |> ignore
411     sb.AppendLine("        <hidden value=\"false\"/>") |> ignore
412     sb.AppendLine("        <cdrectangleparam minX=\"10\" maxX=\"1400\" minY
            =\"10\" maxY=\"900\"/>") |> ignore
413     sb.AppendLine($"       <infoparam name=\"block\" value=\"{escapeXml blockName
            }\"/>") |> ignore
414     sb.AppendLine("        <new d=\"false\"/>") |> ignore
415     sb.AppendLine("        <extraparam>") |> ignore
416     sb.AppendLine("          <blockType data=\"block\" color=\"-4072719\"/>") |>
            ignore
417     sb.AppendLine("          <CryptoBlock value=\"false\"/>") |> ignore
418     sb.AppendLine(generateAttributesXml allVars) |> ignore
419     sb.AppendLine("        </extraparam>") |> ignore
420     sb.AppendLine("      </COMPONENT>") |> ignore
421     sb.AppendLine("    </AVATARBlockDiagramPanel>") |> ignore
422
423     // State Machine Panel (includes all states and transitions)
424     sb.AppendLine("    <AVATARStateMachineDiagramPanel name=\"StateMachine\" minX
            =\"10\" maxX=\"2500\" minY=\"10\" maxY=\"1500\" zoom=\"1.0\">") |> ignore
425     let numPerRow = 4
426     let statePositions =
427         sysmlModel.States
428         |> List.mapi (fun i state ->
429             let x = 100 + 300 * (i % numPerRow)
430             let y = 100 + 220 * (i / numPerRow)
431             (state.Name, (x, y))
432         )
433         |> Map.ofList
434     let mutable compId = 0
435     let mutable connId = 0
436     let mutable subcompId = 10000
437     // Emit all state nodes as COMPONENTS
438     for state in sysmlModel.States do
439         let uid = System.Guid.NewGuid().ToString()
440         let (x, y) = statePositions.[state.Name]
441         sb.AppendLine($"      <COMPONENT type=\"5106\" id=\"{compId}\" index=\"{
                compId}\" uid=\"{uid}\">") |> ignore
442         sb.AppendLine($"        <cdparam x=\"{x}\" y=\"{y}\"/>") |> ignore
443         sb.AppendLine($"        <sizeparam width=\"250\" height=\"200\" minWidth
                =\"5\" minHeight=\"2\" maxWidth=\"2000\" maxHeight=\"2000\"
                minDesiredWidth=\"0\" minDesiredHeight=\"0\"/>") |> ignore
444         sb.AppendLine($"        <hidden value=\"false\"/>") |> ignore
445         sb.AppendLine($"        <cdrectangleparam minX=\"10\" maxX=\"2500\" minY
                =\"10\" maxY=\"1500\"/>") |> ignore
446         sb.AppendLine($"        <infoparam name=\"state\" value=\"{escapeXml
                state.Name}\"/>") |> ignore
447         if not (List.isEmpty state.EntryActions) then
448             let entryStr = String.concat "; " (state.EntryActions |> List.map
                    escapeXml)
449             sb.AppendLine($"        <entry>{entryStr}</entry>") |> ignore
450         sb.AppendLine("      </COMPONENT>") |> ignore
451         compId <- compId + 1
452
453     // Emit all transitions as CONNECTORS and SUBCOMPONENTS
454     let getStatePos name =
455         match Map.tryFind name statePositions with
```

```
456         | Some (x, y) -> (x, y)
457         | None -> (100, 100)
458     for trans in sysmlModel.Transitions do
459         let uid = System.Guid.NewGuid().ToString()
460         let subUid = System.Guid.NewGuid().ToString()
461         let stateWidth = 250
462         let stateHeight = 200
463         let (srcX, srcY) = getStatePos trans.From
464         let (dstX, dstY) = getStatePos trans.To
465         let (p1x, p1y, p2x, p2y) =
466             if trans.From = trans.To then
467                 (srcX + stateWidth, srcY + stateHeight / 2,
468                  srcX + stateWidth + 60, srcY + stateHeight / 2 + 60)
469             elif dstX > srcX then
470                 (srcX + stateWidth, srcY + stateHeight / 2,
471                  dstX, dstY + stateHeight / 2)
472             elif dstX < srcX then
473                 (srcX, srcY + stateHeight / 2,
474                  dstX + stateWidth, dstY + stateHeight / 2)
475             elif dstY > srcY then
476                 (srcX + stateWidth / 2, srcY + stateHeight,
477                  dstX + stateWidth / 2, dstY)
478             else
479                 (srcX + stateWidth / 2, srcY,
480                  dstX + stateWidth / 2, dstY + stateHeight)
481
482         let midX = (p1x + p2x) / 2
483         let midY = (p1y + p2y) / 2
484
485         sb.AppendLine($"      <CONNECTOR type=\"5102\" id=\"{connId}\" index=\"{
                connId}\" uid=\"{uid}\">") |> ignore
486         sb.AppendLine($"        <cdparam x=\"{midX}\" y=\"{midY}\"/>") |> ignore
487         sb.AppendLine($"        <sizeparam width=\"0\" height=\"0\" minWidth
                =\"0\" minHeight=\"0\" maxWidth=\"2000\" maxHeight=\"2000\"
                minDesiredWidth=\"0\" minDesiredHeight=\"0\"/>") |> ignore
488         sb.AppendLine($"        <infoparam name=\"connector\" value=\"null\" from
                =\"{escapeXml trans.From}\" to=\"{escapeXml trans.To}\"/>") |> ignore
489         sb.AppendLine($"        <TGConnectingPoint num=\"0\" id=\"{connId +
                1}\"/>") |> ignore
490         sb.AppendLine($"        <P1 x=\"{p1x}\" y=\"{p1y}\" id=\"{connId +
                2}\"/>") |> ignore
491         sb.AppendLine($"        <P2 x=\"{p2x}\" y=\"{p2y}\" id=\"{connId +
                3}\"/>") |> ignore
492         sb.AppendLine($"        <AutomaticDrawing data=\"true\"/>") |> ignore
493         sb.AppendLine($"        <new d=\"false\"/>") |> ignore
494         sb.AppendLine("      </CONNECTOR>") |> ignore
495
496         let guardPart = trans.Guard
497         let actionPart = trans.Action
498
499         sb.AppendLine($"      <SUBCOMPONENT type=\"-1\" id=\"{subcompId}\" index
                =\"{connId}\" uid=\"{subUid}\">") |> ignore
500         sb.AppendLine($"        <father id=\"{connId}\" num=\"0\"/>") |> ignore
501         sb.AppendLine($"        <cdparam x=\"{midX}\" y=\"{midY}\"/>") |> ignore
502         sb.AppendLine($"        <sizeparam width=\"50\" height=\"25\" minWidth
                =\"0\" minHeight=\"0\" maxWidth=\"2000\" maxHeight=\"2000\"
                minDesiredWidth=\"0\" minDesiredHeight=\"0\"/>") |> ignore
503         sb.AppendLine($"        <hidden value=\"false\"/>") |> ignore
```

```
504         sb.AppendLine($"        <enabled value=\"true\"/>") |> ignore
505         sb.AppendLine($"        <cdrectangleparam minX=\"10\" maxX=\"2500\" minY
                =\"10\" maxY=\"1500\"/>") |> ignore
506         sb.AppendLine($"        <infoparam name=\"List of all parameters of an
                Avatar SMD transition\" value=\"\"/>") |> ignore
507         sb.AppendLine($"        <new d=\"false\"/>") |> ignore
508         sb.AppendLine($"        <TGConnectingPoint num=\"0\" id=\"{subcompId +
                1}\"/>") |> ignore
509         sb.AppendLine($"        <TGConnectingPoint num=\"1\" id=\"{subcompId +
                2}\"/>") |> ignore
510         sb.AppendLine($"        <TGConnectingPoint num=\"2\" id=\"{subcompId +
                3}\"/>") |> ignore
511         sb.AppendLine($"        <TGConnectingPoint num=\"3\" id=\"{subcompId +
                4}\"/>") |> ignore
512         sb.AppendLine($"        <extraparam>") |> ignore
513         sb.AppendLine($"          <guard value=\"[{escapeXml guardPart}]\"
                enabled=\"true\"/>") |> ignore
514         sb.AppendLine($"          <afterMin value=\"\" enabled=\"true\"/>") |>
                ignore
515         sb.AppendLine($"          <afterMax value=\"\" enabled=\"true\"/>") |>
                ignore
516         sb.AppendLine($"          <extraDelay1 value=\"\" enabled=\"true\"/>") |>
                 ignore
517         sb.AppendLine($"          <extraDelay2 value=\"\" enabled=\"true\"/>") |>
                 ignore
518         sb.AppendLine($"          <delayDistributionLaw value=\"0\" enabled=\"
                true\"/>") |> ignore
519         sb.AppendLine($"          <computeMin value=\"\" enabled=\"true\"/>") |>
                ignore
520         sb.AppendLine($"          <computeMax value=\"\" enabled=\"true\"/>") |>
                ignore
521         sb.AppendLine($"          <probability value=\"\" enabled=\"true\"/>") |>
                 ignore
522         if actionPart <> "" then
523             sb.AppendLine($"          <actions value=\"{escapeXml actionPart}\"
                    enabled=\"true\"/>") |> ignore
524         sb.AppendLine($"        </extraparam>") |> ignore
525         sb.AppendLine($"      </SUBCOMPONENT>") |> ignore
526
527         connId <- connId + 10
528         subcompId <- subcompId + 10
529
530     sb.AppendLine("    </AVATARStateMachineDiagramPanel>") |> ignore
531     sb.AppendLine("  </Modeling>") |> ignore
532     sb.AppendLine("</TURTLEGMODELING>") |> ignore
533     sb.ToString()
534
535
536
537 // --------------------------- MAIN: Put it all together ---------------------
538
539 // Apply EFSM transformation pipeline
540 let transformedEFSM = applyTransformationWithCommit completeEFSM
541
542 // Work on Requirements
543 let requirements = extractRequirementsFromEFSM transformedEFSM
544 let efsmNoAssert = removeAssertActionsFromEFSM transformedEFSM
545
```

```
546 // Generate SysML state machine model
547 let sysMLModel = EFSMtoSysML efsmNoAssert
548
549 // Generate full XML including requirements, block diagram, and state machine
550 let sysmlXml = generateSysMLXml sysMLModel requirements
551
552 // Output the XML to file (change path as needed)
553 let outputFile = @"E:\Master's Study Material\Thesis\Code\QuartzToSysML_Robot2.
        xml"
554 System.IO.File.WriteAllText(outputFile, sysmlXml)
555
556 printfn "Quartz EFSM translated to SysML XML!"
```

Bibliography

[1] "Embedded systems market." Accessed: 17.07.2025. (2025), [Online]. Available: https://www.fortunebusinessinsights.com/embedded-systems-market-108767.

[2] L. Cunha, J. Sousa, J. Azevedo, S. Pinto, and T. Gomes, "Security first, safety next: The next-generation embedded sensors for autonomous vehicles," *Electronics*, vol. 14, no. 11, p. 2172, 2025.

[3] K. Schneider and J. Brandt, "Quartz: A synchronous language for model-based design of reactive embedded systems," in *Handbook of Hardware/Software Codesign*, Springer, 2017, pp. 29–58.

[4] K. Schneider and T. Schuele, "Averest: Specification, verification, and implementation of reactive systems," in *Conference on Application of Concurrency to System Design (ACSD)*, 2005.

[5] I. Dragomir, I. Ober, and D. Lesens, "A case study in formal system engineering with sysml," in *2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*, 2012, pp. 189–198. DOI: 10.1109/ICECCS20050.2012.6299214.

[6] L. W. Li, D. Genius, and L. Apvrille, "Formal and virtual multi-level design space exploration," in *International Conference on Model-Driven Engineering and Software Development*, Springer, 2017, pp. 47–71.

[7] N. Halbwachs, "Synchronous programming of reactive systems: A tutorial and commented bibliography," in *International Conference on Computer Aided Verification*, Springer, 1998, pp. 1–16.

[8] K. Schneider, "The synchronous programming language quartz," Internal Report 375, Department of Computer Science, University of Kaiserslautern, Tech. Rep., 2009.

[9] "Omg systems modeling language." Accessed: 13.07.2025. (2015), [Online]. Available: https://www.omg.org/spec/SysML/1.4/PDF.

[10] K. Schneider. "Averest framework." Accessed: 12.07.2025. (2025), [Online]. Available: http://www.averest.org/.

[11] L. Apvrille. "Ttool website." Accessed: 16.07.2025. (2013), [Online]. Available: https://ttool.telecom-paris.fr/.

[12] K. Schneider. "Api reference." Accessed: 13.07.2025. (), [Online]. Available: http://www.averest.org/AverestLibDoc/reference/index.html.

[13] M. A. B. Khadra, Y. Bai, and K. Schneider, "Synthesis of distributed synchronous specifications to systemoc," in *MBMV*, 2014, pp. 71–81.

[14] J. Brandt, "Synchronous models for embedded software," Ph.D. dissertation, Technische Universität Kaiserslautern, 2013.

[15] C. Gündogan and A. M. Sahin, "Interaktive verifikation von synchronen systemen,"

[16] K. Schneider, T. Schuele, and M. Trapp, "Verifying the adaptation behavior of embedded systems," in *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, 2006, pp. 16–22.

[17] G. Pedroza, L. Apvrille, and D. Knorreck, "Avatar: A sysml environment for the formal verification of safety and security properties," in *2011 11th Annual International Conference on New Technologies of Distributed Systems*, IEEE, 2011, pp. 1–10.

[18] Y. Roudier, M. S. Idrees, and L. Apvrille, "Sysml-sec: A sysml environment for the design and development of secure embedded systems," *XP055249349*,

[19] L. Apvrille and Y. Roudier, "Towards the model-driven engineering of secure yet safe embedded systems," *arXiv preprint arXiv:1404.1985*, 2014.

[20] Y. Roudier and L. Apvrille, "Sysml-sec: A model driven approach for designing safe and secure systems," in *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, IEEE, 2015, pp. 655–664.

[21] L. Apvrille and A. Becoulet, "Prototyping an embedded automotive system from its uml/sysml models," in *Embedded Real Time Software and Systems (ERTS2012)*, 2012.

[22] L. Apvrille and Y. Roudier. "Sysml-sec website." Accessed: 16.07.2025. (), [Online]. Available: https://sysml-sec.telecom-paris.fr/.

[23] L. Apvrille, "Ttool for diplodocus: An environment for design space exploration," in *Proceedings of the 8th international conference on New technologies in distributed systems*, 2008, pp. 1–4.

[24] A. Enrici, L. Li, L. Apvrille, and D. Blouin, "A tutorial on ttool/diplodocus: An open-source toolkit for the design of data-flow embedded systems," Tech. rep, Tech. Rep., 2022.

[25] L. Apvrille and T. Alessandro. "Avatar model checker." Accessed: 16.07.2025. (2020), [Online]. Available: https://ttool.telecom-paris.fr/docs/ttool_avatarmodelchecker.pdf.

[26] D. Genius, L. Li, and L. Apvrille, "Model-driven performance evaluation and formal verification for multi-level embedded system design," in *5th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, 2017.

[27] L. Apvrille. "Ttool development infrastructure." Accessed: 12.07.2025. (2025), [Online]. Available: https://gitlab.telecom-paris.fr/mbe-tools/TTool/-/blob/master/doc/dev_infrastructure/ttool_development_infrastructure.tex?ref_type=heads.

[28] L. Apvrille. "Coffee machine avatar." Accessed: 13.07.2025. (), [Online]. Available: https://ttool.telecom-paris.fr/networkmodels/CoffeeMachine_Avatar.xml.

[29] L. Apvrille and A. Enrici. "Tg component manager." Accessed: 12.07.2025. (2025), [Online]. Available: https://gitlab.telecom-paris.fr/mbe-tools/TTool/-/blob/master/src/main/java/ui/TGComponentManager.java.

[30] R. E. S. Group. "Tools for quartz programs." Accessed: 12.07.2025. (2022), [Online]. Available: https://es.cs.rptu.de/tools/TeachingTools/Quartz.html.

Tools

Note on the Use of Generative AI

Parts of this thesis were written with the assistance of the language model Chat GPT by OpenAI and Grammarly. The AI was used as a tool to support the drafting process through language generation, rephrasing, or stylistic suggestions. All AI-assisted content has been critically reviewed, edited, and verified by the author to ensure accuracy, academic integrity, and compliance with the standards of scholarly work.