Converting Graphical Models to Formal Specifications

by

Alexandre Marques

Student ID: 101189743

Submitted to: Professor Jason Jaskolka

Capstone Project - Final Report

SYSC 4907 - Engineering Project

Department of Systems and Computer Engineering

Faculty of Engineering

Carleton University

April 8, 2025

Abstract

Formal specifications are a useful method of communicating system descriptions due to their lack of ambiguity, but they tend to take time to write and require skills few people possess. In this paper, we propose a way to derive formal specifications from commonly generated informal graphical models. With our tool, the UML to C2KA Converter (U2C), we reduce the barriers of formal methods to engineer quality systems. We managed to read UML State Diagrams to automatically produce Communicating Concurrent Kleene Algebra (C2KA) specifications. These specifications can be fed to C2KA model checkers, like the Implicit Interactions Analysis Tool (IIAT). By improving the accessibility of these model checkers, we can identify vulnerabilities and faults earlier in the design process. This should reduce design costs and reduce operational damages by helping engineers build safer and more secure systems.

Keywords: C2KA, UML, Finite State Machine, state diagram, formal methods, model checking, model driven engineering.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Problem Background

Modern system requirements are becoming increasingly complex over time. To fulfill these requirements, engineers typically go through a modelling phase. Models are artifacts from the modelling phase that represent different views of the system to understand various aspects of it. Producing graphical models in languages such as UML (Unified Modelling Language) is a typical modelling technique to communicate information quickly through visual system interactions. This communication works well across levels of system details between humans, but they tend to be written with informal modelling languages due to their simplicity. This means the semantic meaning of the model is nondeterministic, and computers cannot interpret most graphical models. In contrast, formal modelling languages like C2KA (Communicating Concurrent Kleene Algebra) have a defined semantics. This allows computers to perform rigorous automated model checking on formal models of the system. Critical system properties like safety and security can be proven at the model level before any system implementation starts.

System descriptions can vary in quality, especially across different stages of design. They can range from informal and incomplete natural language descriptions to well-defined formal requirements. Engineers need to make reasonable decisions on how to model systems from these descriptions. This often means going for informal visual models that are easy to produce and communicate with. Even with their known model-checking benefits, formal models are often dismissed. They require more time to make and specialized skills to produce and understand them. This means formal models cannot easily replace informal models, they typically supplement them.

1.2 Problem Motivation

We wanted to take advantage of the benefits of formal modelling methodologies without facing the barriers they typically impose. We believed we could use the informal models typically created to derive formal models with minimal additional time cost. We were specifically interested in the C2KA formalism for a few reasons. The language is suitable for model checking system properties relating to interactions between components, which is an increasing concern in complex systems with many components. The language is new and has poor support from existing tools. Our tool would be a welcome addition to the ecosystem, contributing greatly to the growth of C2KA. Experienced C2KA modellers believed this model transformation was possible in this formalism. This gave us confidence that our project would be feasible and beneficial.

1.3 Problem Statement

Our project aims to convert UML State Diagram visual models into C2KA agent specifications. We are aiming to use these agent specifications in our C2KA model checker of interest, the Implicit Interactions Analysis Tool (IIAT) [7]. By targeting a specific model checker, it is easier to prove the function and usefulness of our tool.

1.4 Proposed Solution

We are building a tool as our solution: the "UML to C2KA Converter", or **U2C** for short. To achieve its goals, we decided to find transformation patterns manually first. Once we understood how to convert it manually, we would encode the rules deterministically. That is, we wanted our program to behave like a pure function. It should map one given input to precisely one reproducible output. The output in question would be the agent specification files that the IIAT uses. The input would be some machine-readable version of a set of UML State Diagrams.

1.5 Accomplishments

1. Found deterministic transformation rules for visual models to C2KA specifications.
2. Managed to read a textual representation of the graphical models.
3. Re-created a diagram structure internally to easily parse state diagram information in code.
4. Used our custom structure to simply implement the transformation rules we determined.
5. Formatted and serialized our specifications in a format the IIAT could use.
6. Created a diff tool to validate our outputs given a trusted C2KA specification written by hand.
7. Validated our tool against one known C2KA system, using our diff tool, and by observing parity in the IIAT.

1.6 Document Overview

The remainder of this report is organized as follows. Section 2.0 outlines how our project meets the objectives of an engineering project. Section 3.0 goes over our methodologies and results in technical detail. Finally, Section 4.0 contains our conclusions, reflexions, and possible future work.

2 Engineering Process

2.1 Health and Safety

Since we are working on a pure software project, we do not interact with any dangerous tools or hardware. The most fearsome tools for us are our office peripherals. We need to be conscious of the ergonomics of our work environment.

Monitor placement is important because it affects eye strain and postural strain (neck and shoulders). It is recommended to keep the monitor approximately forty to seventy centimeters away from the eyes. For a quick reference, this is roughly an arm's length away, but it depends on the person. The monitor height is important as well. In general, research has found that eyes naturally have a downward cast [2]. They strain more looking above than looking down. In practice, guidelines recommend keeping the top of the monitor at eye level or slightly below. In general, these are just guidelines to get a good starting point. It may be worth experimenting with, depending on the individual's body and what feels best. A visual summary of these recommendations is given by the Canadian Centre for Occupational Health and Safety (CCOHS), in Figure 1.

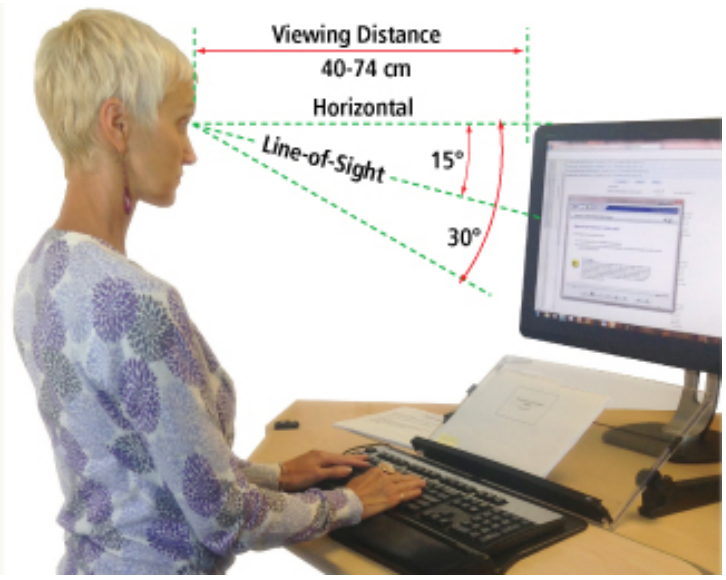

Figure 1: The recommended monitor position guidelines from CCOHS [2].

When working for excessively long periods without breaks, it is possible to get repetitive strain injuries (RSI). It can be from clicking the mouse too much or typing a lot on the keyboard. Certain mice and keyboards are more ergonomic and help reduce these strains. They usually have aggressive curves, forcing your body to adopt more ergonomic poses. Office chairs are also an important part of ergonomics, providing proper support while sitting at a desk.

However, even with the best ergonomic setups, the most important mitigation is taking frequent breaks from the computer. Getting up and walking is good to reduce eye strain, as well as reducing the chances of getting an RSI. To do this reliably, it is best to set timers and respect them. Otherwise, there is a risk of getting too absorbed in the work. After 8 hours of work without breaks writing a report, your hands start aching, and your body will be the one demanding a break.

2.2 Engineering Professionalism

In ECOR4995, we learned that safety was paramount.

Our first step in addressing this was to analyze the security concerns of our tool. We do not believe there to be any security vulnerabilities. This is because it is completely local to the machine, with no internet connection. We do not believe that someone using our tool can harm someone other than themselves by intentionally misusing it. The only concern would be if an individual delivers falsified outputs to provide to someone else. However, we provide no guarantee for this. We create text files that can be spoofed without our tool even existing, we decided this concern to be out of scope.

The next question is, are there safety concerns where users can cause harm unintentionally? We believe there is a way to do so if our outputs provide false information to a designer. If the analyzed system is safety-critical, this can become a safety concern. The analysis done with our faulty outputs would then compromise the analysis of the safety-critical system. To guard against this, the accuracy of our outputs became a critical

requirement in our user requirements (see Section 3.2.1). We even have a second requirement preferring failure over false positive outputs.

We also made sure to properly license our program since we learned that intellectual property was pretty important.

2.3 Project Management

We started the project by defining some processes for baseline communication and work hours. That strategy did not work well because we still faced long periods without any work or communication. We had weekly meetings at first, but we wasted time because we had multiple weeks with no work.

The crux of our recovery from this disastrous start came from a pivot towards asynchronous project management with **GitHub Issues**. We will look at their purpose in more detail in Section 3.2.8. In summary, they were used to formulate task units at a lower level than our requirements. We could also self-assign the tasks we wanted to do to show everyone we started work on it. This is to have a person to contact for status updates or permission to collaborate on it and avoid duplicating work.

In practice, we stopped doing the assignments because only one person was working for a long time. During the last week before the fair, the team became slightly more active. Project Management went back to synchronous meetings with dedicated task assignments and frequent check-ups to ensure no one was blocked for too long.

We also defined timelines, but they were not respected either. They only served as a reminder that we were permanently behind schedule, even when we revised our timeline later. When we switched to Issues, we ended up somewhat ignoring timelines. We were already working as much as possible until we got blocked. Usually this was because we had trouble completing or understanding a C2KA transformation. During our meetings with our supervisor, we would get the opportunity to discuss these issues to unblock us or give us a source to look into for research.

Having said that, we still had a clear roadmap of features to follow and

a target date for a prototype (the fair). When we were planning tasks, we asked ourselves the feasibility of completion. This was done in an ad-hoc manner, based on our knowledge of the current system, and the work left to do. As we went, we also made sure to mark out-of-scope features whenever possible (with a rationale) to increase feasibility.

2.4 Degree Suitability

2.4.1 Alexandre Marques

This project was suitable to my degree because it touches on many concepts I have learned in class. It is an engineering project, and it is a software project, so naturally, it should fit my Software Engineering degree. More specifically, the skills and knowledge I have applied from my courses are as follows:

- I have applied induction rules from COM1805 and COMP2804 to implement the recursive logic needed to convert C2KA diagrams (technically learned in SYSC2100, but discrete math taught me recursion much better).
- I have applied my knowledge of UML intially learned from SYSC2004 to understand the composition of State Diagrams, and their purpose.
- I have applied my knowledge of programming languages from SYSC3101 to select the best language for our needs.
- I have applied Object-Oriented design patterns from SYSC3110 to improve program maintainability and capabilities
- I have applied requirement engineering techniques from SYSC3120 to write a set of functional and non-functional requirements for our program.
- I have applied my Operating System knowledge from SYSC4001 to work on multiple OS hosts and make the IIAT work.

- I have applied Verification & Validation concepts from SYSC4101 to increase quality
- I have applied Project Management techniques from SYSC4106 to increase the chances of completing the project
- I have applied my knowledge of formal methods from SYSC4111 to identify the importance of formal languages and understand C2KA better.
- I have applied program architecture pattern knowledge from SYSC4120 to select and implement the Pipe and Filter pattern as the best architecture for our goals.
- I have applied CI/CD concepts from SYSC4806 to streamline our development process
- I have applied my knowledge of modelling from SYSC5805 and SYSC5104 to understand the importance of models in the engineering process.
- I have applied my experience in writing technical reports from an academic research internship I've done before the degree, combined with the LaTeX skills I have learned during the degree submitting assignments in various classes. Although the internship was not part of my degree, it was valuable, relevant experience in this domain. I was only offered the opportunity because I aimed to do something related to software before even starting the degree.
- I have omitted some pre-requisite classes which were not directly relevant, but they also contributed to giving me the capabilities to complete this project. It feels like we see them all the time in different classes. That was important pre-requisite knowledge, too.

2.4.2 Michael Rochefort

This capstone project integrates and applies a broad range of skills and knowledge from the Software Engineering degree, demonstrating its suitability as a culminating project for the program. At its core, the project

required engineering a complex software system from conception to verification, which aligns perfectly with Software Engineering principles. We began with requirements analysis (identifying what the tool needs to do and under what constraints), a process learned in our Requirements Engineering courses. We then moved to software design, selecting an architecture (pipe-and-filter) and design patterns appropriate for the problem. This reflects knowledge from software architecture and design courses, where evaluating and applying architectural styles is a key learning outcome. The implementation involved writing a substantial amount of code in an organized manner, applying best practices for coding and documentation that we developed throughout my degree. Notably, the project's focus on converting UML models to formal specifications bridged theory and practice: we utilized UML modelling techniques taught in modelling and software design classes, and we engaged with formal methods by generating specifications.

The project also demanded proficient use of software engineering tools and practices. For instance, we used version control (Git/GitHub) extensively – a skill emphasized in our software engineering labs – to collaborate and manage our codebase. We practiced issue tracking and agile-like iteration, echoing project management coursework, to keep the team organized and responsive to changes. The testing strategy we used is a direct application of software testing principles learned in class; we wrote unit tests, integration tests, and system tests much as we would in an industrial setting, reinforcing our understanding of testing frameworks and the importance of test coverage.

Lastly, this project involved teamwork and communication, soft skills that are integral to the Software Engineering program. We had to collaborate effectively, divide tasks, conduct code reviews, and integrate our work, which mirrors the team projects and assignments throughout my degree. The production of a comprehensive technical report and documentation tested my technical communication skills, another key component of the program's outcomes.

2.4.3 Ahmed Babar

The Goal of this project is to streamline and reduce errors by creating an automated input for an existing error analysis tool. This project is suitable for the software development degree program as it demonstrates Object oriented programming picked up from courses like 3303 and 2006 where multithreading is used to improve efficiency and 2006 where coupling and techniques were used to create object oriented project just like this one. Another example of use of knowledge from previous course related to this degree include 4001 software validation and 4120 and 3120.

2.5 Individual Project Contributions

2.5.1 Alexandre Marques

- Created a knowledge base in Notion for the team to share information and document decisions as we progress.
- Defined team process for Communication, ensuring some minimal standard of communication is agreed on.
- Defined team process for Work Hours based on expected total work hours for the project and availability of the team to have established a shared expectation of regular work effort.
- Organized regular weekly meetings until the progress report, writing an agenda beforehand and writing meeting notes afterward.
- Assigned appropriate tasks to delegate work equitably at weekly meetings.
- Presented and incorporated feedback for processes to teammates to approve them and make them official for the whole team.
- Performed preliminary research on how to derive C2KA specifications for a system, and documented a detailed modelling methodology in Notion.
- Defined review criteria for modelling tool selection.

- Tested and reviewed four modelling tools.
- Selected the modelling tool which seemed the most appropriate for our needs by comparing all reviews.
- Wrote a set of unofficial requirements for the project to help determine objectives, and initial architecture.
- Suggested edits for the introduction, background, proposed solution, and risks sections of the proposal.
- Determined and wrote initial project objectives in the proposal.
- Designed the first iteration of our program architecture, and linked it to the project objectives in the proposal.
- Rewrote the initial timeline in the proposal to be more realistic, and congruent with the project objectives.
- Defined review criteria for XMI Library, and methodology to test them and communicate their suitability.
- Tested two XMI parsing libraries, one of which being SDMetrics, the only XMI library we managed to use.
- Wrote the initial code allowing us to use SDMetrics, and show extracted model data from a file.
- Rewrote Summary, Progress Summary, and conclusion in the Progress Report to remove inaccuracies, add relevant details, and improve quality to an adequate level.
- Suggested edits for the Background in the Progress Report.
- Wrote challenges faced section in the Progress Report and used it as a way to reflect and improve methodologies in the project.
- Rewrote timeline to be more realistic with our lost time, and implemented lessons learned from the first timeline.
- Changed project management approach to be asynchronous, established a system based on GitHub issues that can be created by anyone, and self-assigned improving our efficiency massively.

- Researched collaboration diagrams to understand what they are used for, what are their model elements, and how can we use those elements to map them to C2KA specifications.
- Defined and implemented a pull request template in GitHub to guide our review communications and establish rules.
- Created an interface to configure the XMI Parsing library for different inputs, and parsing modes.
- Implemented initial test strategy for unit testing, by creating tests for the XMI Parser config.
- Implemented a proof of concept to find all system agents, and stimuli in a collaboration diagram.
- Unit tested functions to find agents and stimuli in the diagram.
- Set up a continuous integration pipeline with GitHub actions to automatically test our code before merging.
- Code reviewed all the pull requests in the repository, providing meaningful feedback to improve code quality.
- Suggested comments as a way to read annotations compensating for the lack of behavior information in collaboration diagrams.
- Modified parser config files to be able to read comments.
- Analyzed state diagrams and identified elements of interest for our C2KA transformations. Later, revisited this idea by trying to formalize it through C2KA base representations.
- Rewrote design completely after the parsing stage when we realized State Diagrams were a better fit for our uses after discussing with our Supervisor.
- Created initial state diagrams to explore C2KA mappings.
- Revised our initial architecture to classify our program into pipes and filters.

- Wrote and presented slides for Collaboration Diagrams, Implementation (Current, Future), and Project Statem.
- Edited the challenges section to make sense in the newer context of our project, as they were originally copied from my work in the progress report.
- Took notes of presentation feedback, to improve future presentation and reports.
- Implemented a filter to extract stimuli from state diagram transitions.
- Read a paper on the foundations of C2KA [8] to develop a deep understanding of possible ways of representing systems in C2KA.
- Documented rules and possible mappings of C2KA in an internal wiki, using the Base Representations for reference.
- Revised our architecture to add a transformation step before we interpret C2KA transformations: the "diagram linking" step 3.2.7.
- Broke down C2KA interpretation step into filters for each part of the specification, and changed their inputs to be the linked diagram instead of primitives.
- Frequently discussed with our supervisor to find the best ways to represent concrete behavior, and composition of behaviors in state diagrams, as well as finding directions for research to answer these questions.
- Implemented all pipes, and all filters incrementally, adding integration tests and unit tests for them before merging them.
- Defined test strategy, describing when to do unit tests, integration tests, or system tests.
- Refactored integration tests and streamlined testing by creating reusable pipelines.
- Fixed draft Manufacturing Cell diagrams in Papyrus to be accurate and testable according to our input specifications

- Refactored tests to import test paths and the diff tool as reusable utilities.
- Wrote specification formatting classes to output our collected information in the same syntax as the IIAT.
- Debugged and fixed specific niche cases relating to sequential states, or other assumptions which did not hold during system tests. One example, we assumed only sequential states had the same input/output stimuli. This did not hold for the waste water system, instead we used the missing "/" of sequential transitions to identify them.
- Managed the team during the last two week before the fair to frequently check status, set deadlines, and assign tasks when possible.
- Pushed team to write a draft final report to get crucial feedback for creation of the final report.
- Designed poster layout, and formated slides.
- Wrote an updated architecture figure with inputs / outputs of filters, incorporating feedback from the presentation.
- Wrote a subset of our formal requirements to include in our poster, and the Future Work slide.
- Designed motivation and solution slide to be visually descriptive and concise.
- Reviewed and gave feedback to teammates to help focus and reduce our text, and punctuate with visual aids (Presentation, and Poster Fair).
- Managed and tracked GitHub issues, making sure to carefully label them and create them with enough detail for anyone to complete them adequately.
- Created Question issues to track questions we were unable to answer to progress, allowing us to track research progress and reminding us to ask them during meetings.

- Closed all relevant issues before our final V1.0 release, marking less critical open issues as out of scope.
- Executed the IIAT tool with our outputs to prove our tool worked, documented how to do so for future readers in the report.
- Wrote the Final Report.

2.5.2 Michael Rochefort

- Helped figure out comment parsing in diagrams (dropped feature). We used this knowledge later to understand how to configure the parser for our needs.
- Did research on Dijkstra's Guarded Command Language (GCL) and the transformations between its syntax and the IIAT. Documented these findings in an internal wiki.
- Drew the C2KA base model diagram in Papyrus related to concrete behavior containing a conditional statement written in GCL.
- Wrote an integration test case for StateDiagramLinker using the C2KA base model with the GCL concrete behavior.
- Created the accuracy validation diff tool, which matches agent specification irrespective of order, with the same semantic meaning.

2.5.3 Ahmed Babar

- Created draft Manufacturing Cell diagrams in Papyrus for all agents (with many errors)
- Did research on State Diagrams, with no clear goal or conclusion. Documented a lot of irrelevant information, making it unclear how to use state diagrams for C2KA derivations.
- Started adding integration test cases to StateDiagramLinker according to a template written by Alexandre, but could not complete them to an adequate level of quality in time.

- Started (but failed to complete) creation of Wastewater system diagram inputs in Papyrus for a second system test.

2.6 Individual Final Report Contributions

Ahmed wrote Section 2.4.3 Micheal wrote Section 2.4.2. However, the work to port both sections to LaTeX from the draft report was done by Alexandre Marques.

Everything else was written by Alexandre Marques. This includes all formatting, figure creation, setting up the LaTeX environment, etc. Figure creation does include creating new diagrams, running the IIAT tool, figuring out why it was breaking, etc. All the work relating to formally defining requirements and tracing their fulfillment was done as a part of this report.

All sections from the draft unless otherwise mentioned above were rewritten from scratch by Alexandre Marques to incorporate the feedback. As a disclaimer, this includes project contributions in Section 2.5 which were edited from the draft report for accuracy by Alexandre Marques.

3 Technical Details

3.1 Background & Terminology

3.1.1 C2KA Agents

Communicating Concurrent Kleene Algebra (C2KA) is a formal language intended to support the analysis of concurrent systems with communicating agents. **Agents** are any systems whose **behaviour** consists of discrete actions [8]. **Stimuli** are messages passed between agents to communicate and cause changes in their behaviours. We won't delve much further into the formal definition of C2KA and its goals, as there are better resources for that [8, 4].

Instead, we will identify relevant terms from a C2KA analysis that was performed on a real system [3]. Our system of interest is a Manufacturing Cell, shown by the collaboration diagram in Figure 2.

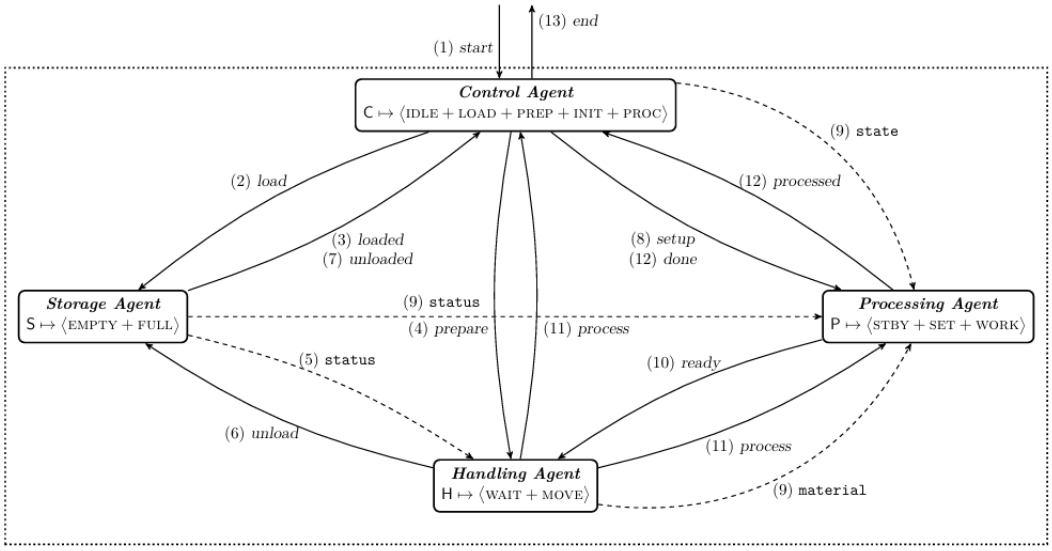

Figure 2: The collaboration diagram representing the Manufacturing Cell [3]

In the boxes, we see the possible discrete agent behaviours, denoted by the "+" choice operator. This is also how we define the **Abstract Behaviour Specification**, as seen in Figure 3.

$$
\begin{aligned}
\mathsf{C} &\mapsto \langle \text{IDLE} + \text{LOAD} + \text{PREP} + \text{INIT} + \text{PROC} \rangle \\
\mathsf{S} &\mapsto \langle \text{EMPTY} + \text{FULL} \rangle \\
\mathsf{H} &\mapsto \langle \text{WAIT} + \text{MOVE} \rangle \\
\mathsf{P} &\mapsto \langle \text{STBY} + \text{SET} + \text{WORK} \rangle
\end{aligned}
$$

Figure 3: The abstract behavior specification for the system [3]

The solid communication arrows represent stimuli. The stimuli are relevant for the Stimulus-Response Specification. We break it down into its components and refer to them in this report as **Next Behaviour Mapping** (Figure 5) and **Next Stimulus Mapping** (Figure 4). As the names may imply, these are functions which map an initial behaviour, and an input stimuli to the Agent's next behaviour, or next stimulus (respectively). These have to be complete functions, but some papers may leave *neutral mappings* empty. These are mappings where a *neutral stimulus* with no impact on agents is sent out, or the behavior remains the same.

λ	*start*	*load*	*loaded*	*prepare*	*done*	*unload*	*unloaded*	*setup*	*ready*	*process*	*processed*	*end*
IDLE	*load*	n	n	n	n	n	n	n	n	n	n	n
LOAD	n	n	*prepare*	n	n	n	n	n	n	n	n	n
PREP	n	n	n	n	n	n	*setup*	n	n	n	n	n
INIT	n	n	n	n	n	n	n	n	n	*done*	n	n
PROC	n	n	n	n	n	n	n	n	n	n	*end*	n

Figure 4: The next stimulus mapping function for agent C [3]

$\circ$	*start*	*load*	*loaded*	*prepare*	*done*	*unload*	*unloaded*	*setup*	*ready*	*process*	*processed*	*end*
IDLE	LOAD	IDLE	IDLE	IDLE	IDLE	IDLE	IDLE	IDLE	IDLE	IDLE	IDLE	IDLE
LOAD	LOAD	LOAD	PREP	LOAD	LOAD	LOAD	LOAD	LOAD	LOAD	LOAD	LOAD	LOAD
PREP	PREP	PREP	PREP	PREP	PREP	PREP	INIT	PREP	PREP	PREP	PREP	PREP
INIT	INIT	INIT	INIT	INIT	INIT	INIT	INIT	INIT	INIT	PROC	INIT	INIT
PROC	PROC	PROC	PROC	PROC	PROC	PROC	PROC	PROC	PROC	PROC	IDLE	PROC

Figure 5: The next behavior mapping function for agent C [3]

The dashed arrows represent shared environmental variable information being passed. The way these values get checked and set is defined in the **Concrete Behaviour Specification**. These are concrete programs associated with specific behaviour, defined in Dijkstra's Guarded Command Language (GCL). An example of this specification can be seen in Figure 6 for agent C of the Manufacturing Cell.

$$\mathsf{C} \mapsto \begin{cases} \text{IDLE} & \stackrel{\text{def}}{=} & \texttt{state} := 0 \\ \text{LOAD} & \stackrel{\text{def}}{=} & \texttt{state} := 1 \\ \text{PREP} & \stackrel{\text{def}}{=} & \texttt{state} := 2 \\ \text{INIT} & \stackrel{\text{def}}{=} & \texttt{state} := 3 \\ \text{PROC} & \stackrel{\text{def}}{=} & \texttt{state} := 4 \end{cases}$$

Figure 6: The Concrete Behavior Specification for agent C [3]

3.1.2 Agents as State Diagrams

UML State Diagrams are an informal way to represent finite state machines. These are abstract machines that can only be in one **state** among a finite set of possible states. To change between states, the machine establishes possible transitions triggered by **inputs**.

When we interchange state with behaviour and inputs with stimuli, we observe many similarities with C2KA. A set of states of a state diagram corresponds quite well with the Abstract Behaviour Specification. Transitions between states indicate which behaviours a stimulus can induce on a behaviour. Paired with the ability to show outputs on a transition, we can also show the next stimulus directly on the transition.

There is even a concept for concrete programs in UML State Diagrams. **doActivities** refer to a concrete action that is executed while in the state. This means we can encode the GCL in the doActivity of a state to describe its Concrete Behaviour.

3.1.3 XMI Model Exports

To start processing a graphical model, we need a way for the computer to read it deterministically. This is where the XML Metadata Interchange (XMI) format helps. It is a standard used to exchange metadata information through XML, mainly for UML models.

The first goal was to be able to model in one tool and be able to pass it around to different tools. Unfortunately, in practice, the imports do not always work cleanly. Even our chosen modelling tool, Papyrus, encodes

additional information outside the ".uml" files to be able to import the diagram.

The second and more relevant goal for us is its use in model-driven engineering. This is the idea that the models should be machine-readable to be able to generate code from them. Although this is not our purpose, we still need this property to perform our analysis and generate specifications instead.

3.2 System Development

3.2.1 User Requirements

These User Requirements represent our original set of requirements, with minimal implementation details and domain knowledge. They are elicited from initial discussions with our supervisor, and interpretations of the problem statement. We did not establish a procedure for explicit lower-level requirements after this stage.

Some limitations of this process are a lack of explicit detail, priorities, or levels of satisfaction for non-functional requirements. Instead, we will use later stages of the system development to trace design decisions taken for these requirements. We will also touch on priorities and possible improvements when we explain which requirements were skipped for our current release in Section 3.2.11.

The table of requirements contains a list of User Requirements for our system, with green states indicating satisfactory completion, and gray indicating unsatisfactory completion:

Table 1: Set of User Requirements for U2C

ID	Requirement	Type	Description	State
1	Input	Functional	U2C shall accept visual system models as input.	

Table 1: Set of User Requirements for U2C

ID	Requirement	Type	Description	State
2	C2KA Specifications	Functional	U2C shall interpret given inputs to derive a complete C2KA agent specification.	
3	IIAT Parameters	Functional	U2C shall interpret given inputs to derive the other required IIAT inputs.	
4	Output	Functional	U2C shall output its computed inputs as text files.	
5	Minimal User Actions	Usability	U2C shall require no additional inputs from the user apart from those required to provide system models.	
6	Modelling Tool Support	Compatibility	U2C shall support models produced by at least one chosen modelling tool (Papyrus, LucidChart, StarUML).	
7	Modelling Language Support	Compatibility	U2C shall support visual models drawn in at least one type of modelling language (ex: UML, SysML, BPMN).	
8	Diagram Type Support	Compatibility	U2C shall support diagrams corresponding to at least one chosen type (ex: state, collaboration, etc.).	

Table 1: Set of User Requirements for U2C

ID	Requirement	Type	Description	State
9	Cross Tool Integration	Compatibility	U2C should provide options to integrate directly into tools it depends on (input producer, and output consumers).	
10	OS Support	Portability	U2C shall run on at least one chosen OS distribution.	
11	Simple System Analysis Speed	Performance	U2C shall produce outputs for a simple system within one minute of execution on a chosen platform specification. A simple system is defined as having up to five agents, twenty stimuli, and five behaviours per agent.	
12	Worst Case Mitigation	Scalability	U2C should scale at a smaller rate than $O(n^3)$. The expected scaling factors are agents, stimuli, and agent behaviours.	
13	User Documentation	Maintainability	The U2C code repository shall contain a user manual detailing expected user interactions and input formats.	

Table 1: Set of User Requirements for U2C

ID	Requirement	Type	Description	State
14	Maintainer Documentation	Maintainability	The U2C code repository shall contain documentation detailing implementation details for maintainers.	
15	Verification Testing	Maintainability	The U2C code repository shall contain automated regression tests providing adequate coverage of the program source code.	
16	Accurate Outputs	Accuracy	U2C shall provide deterministic outputs representing accurately what the inputs contain.	
17	No False Positives	Reliability	U2C shall not provide outputs if it cannot find a deterministic interpretation of the given inputs.	

3.2.2 Chosen Input Type

The first decision to take concerning inputs was the specific diagram type to support. This decision impacts the language since the diagram type may be specific to one or a small set of languages. It also impacts the modelling tool because it needs to support creating that diagram type. Most importantly, if the diagram does not have the information we require, there is no purpose in processing it.

Recall that the information we require is related to creating C2KA agent specifications. Our initial diagram choice was *collaboration diagrams* due

to their simplicity. When we learned more about C2KA, we realized they do not model agent behaviour very well. Collaboration diagrams are more suited to describe a specific scenario in an entire system. Instead, we found that having one **UML State Diagram** per agent is enough to capture agent specifications. More details on this later when we break down relevant components of state diagrams in Section 3.2.3.

By choosing UML State Diagrams, we also choose **UML** as our modelling language. SysML could have been a viable alternative, but we decided to choose a language with which we were more familiar. This makes it easier to implement properly for us, but it is also related to our problem statement, aiming to use modelling languages already familiar to system designers to reduce the learning curve. Admittedly, outside of software system design, SysML is more prevalent, and it is feasible to support both, but we had to limit our scope.

For our modelling tool, we chose **Papyrus**. We established evaluation criteria to evaluate different tools and compared the most popular ones we found. The most important requirement was machine readability. We needed to be able to export some representation of the diagram so our system could read outside the modelling tool. Additionally, it was an advantage if it was free, popular, and had ongoing support. We did not want the modelling tool we depended on to become a hurdle to adopting our program. Ideally, we chose a tool that had widespread appeal. Other tools we looked at were Lucidchart, draw.io, Creately, Gliffy, UMLLet, and PlantUML (all had poor export functions). Modelio (felt hard to use), Astah UML (unpopular), StarUML and Visual Paradigm seemed like good alternatives, but we figured out how to use Papyrus first.

To export diagrams from Papyrus, there is an option to export **XMI files**. They technically have a .uml file extension in Papyrus, but they function the same as XMI files from other tools. Once a diagram for an agent is finished in papyrus, we can add its XMI file to our program's **input folder**. Once all agents are done, we can execute the program. The

user does not need to provide additional inputs. A full workflow using our program is described in Section 3.3.

The table below is a traceability summary mapping our original user requirements to the concepts refining them. Refinements can be correlated by the bolded terms above.

Table 2: Input Decision Requirement Traceability

ID	User Requirement	Refinement
1	Input	Visual Diagrams are given to the system through XMI file exports from supported modelling tool(s).
5	Minimal User Actions	The user puts files in an input folder, then executes the program.
6	Modelling Tool Support	The chosen modelling tool is Papyrus.
7	Modelling Language Support	The chosen modelling language is UML.
8	Diagram Type Support	The chosen input diagram type is UML State Diagrams.

3.2.3 Input Specifications

Although we accept UML State diagrams, there are specific elements that our interpreter cares about. The primary goal was to make use of essential model elements only, specifically states and transitions, to avoid extra modelling requirements. Unfortunately, modelling expressiveness was a smaller priority, as such other elements may cause unexpected behaviour. Ideally, they are ignored. In the next best case is a descriptive error gets raised and crashes the program, preventing a faulty output. In the worst case, an inaccurate specification may be generated without warning.

For the safest behaviour, we have defined a known set of valid model elements, along with usage guidelines. At a high level, it can be summarized with the class diagram in Figure 7.

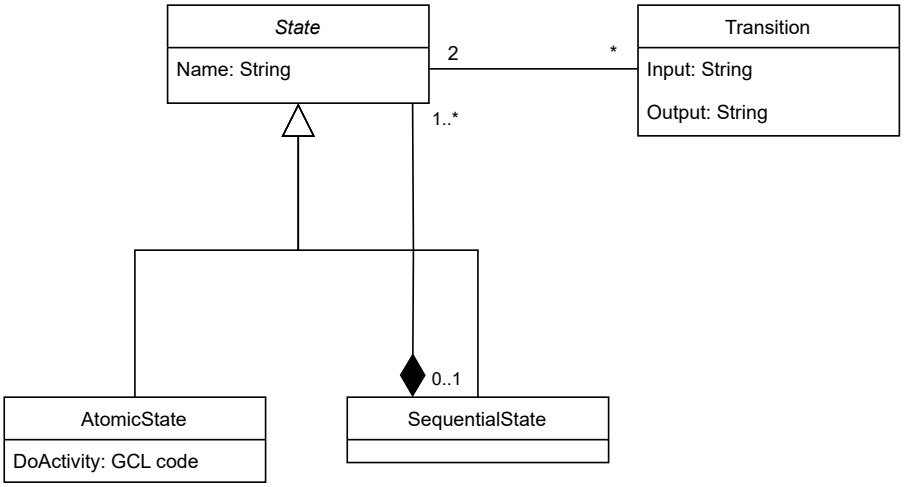

Figure 7: The class diagram representation of supported diagram elements

The first element in our state diagram is a **state**. States describe a discrete behaviour where an invariant condition holds. They map directly to *Abstract behaviours* in C2KA. **Atomic States** are states that do not contain other states. We can see an example of atomic behaviour in Figure 8.

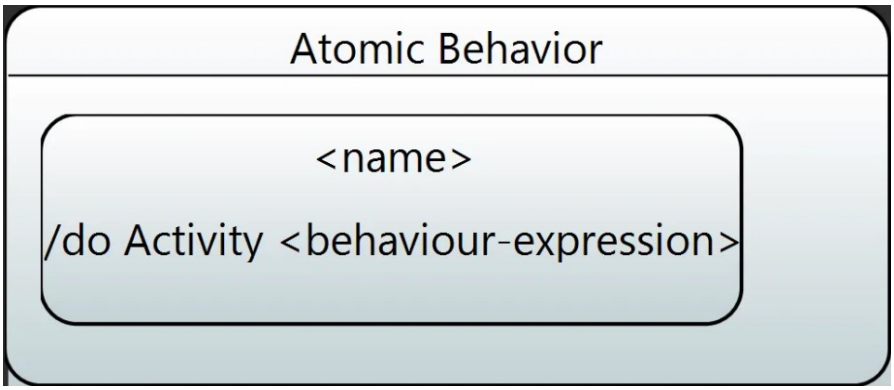

Figure 8: A generic atomic behaviour representation

Atomic States always have a **doActivity** property, as they are needed for C2KA *Concrete behaviours*. doActivities need to follow Dijkstra's Guarded Command Language (GCL) notation [1]. They are used to assign environmental shared variable values, with the option to use conditional flows based on the current environment state. Due to typesetting constraints, we had to convert the GCL operands listed in table 3.

Table 3: GCL Conversions for Modelling Tools

Original GCL	**Modelling Tool**
$\wedge$	&&
$\square$	|
$\vee$	||
$\rightarrow$	$->$

Transitions are links between states that describe how state changes occur. We use these transitions and their associated behaviours to compute *next behaviour* and *next stimulus* functions. For C2KA, we have defined a precise labelling format to follow {input-stimulus}/{output-stimulus}. This follows state diagram convention, but it does not allow for guards; instead, it enforces input and output on all transitions. *sequential transitions* are special transitions with a single stimulus, and their labelling format is: {in-and-out-stimulus}. This is because sequential states use the output of one state as the input of the next state in the sequence. This allows us to uniquely identify sequential compositions by using a single stimulus as the key identifying property. We can see an example of transitions and how we interpret mappings from them in Figure 9.

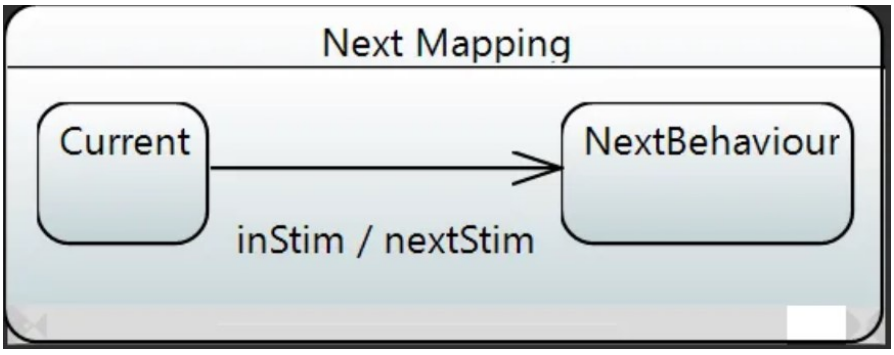

Figure 9: A representation of next mappings and their relation to transtions

Sequential States are super states (state compositions) with a specific composition pattern. There are at least two inner states. All inner states are linked by *sequential transitions*. The sequence has no cycles. There is a clear initial state with no incoming transitions. There is a clear final state with no outgoing transitions. We can see an example of a sequential state in Figure 10.

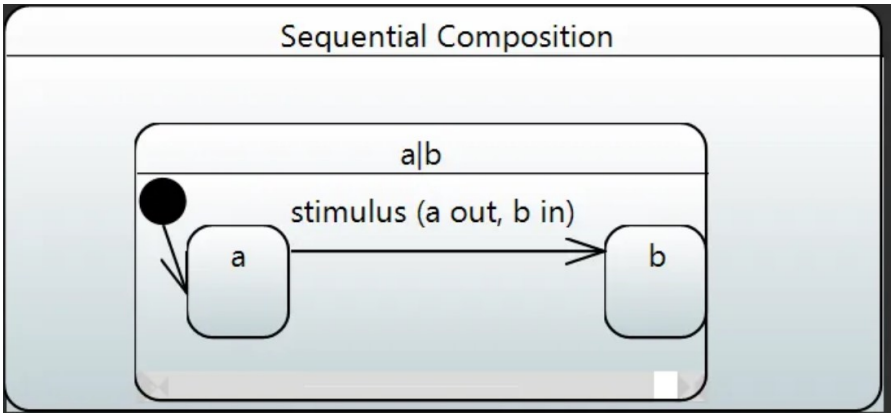

Figure 10: A C2KA sequential composition represented in a state diagram

Note: We explicitly decided not to cover C2KA *parallel compositions* to simplify modelling and parsing. To model parallel behaviour, it is possible to model two different agents that communicate with each other instead.

3.2.4 XMI Parsing

Having chosen our input format, we needed to find a way to parse it to be able to process it. Although XMI is supposed to be a standardized format, many modelling tools will have slight differences for specific components. Therefore, creating a parser that can be extended generally to support many tools can be challenging.

Thankfully, this is a general problem others have attempted to solve before, and third-party free libraries exist. We searched for libraries that supported the most recent UML standards. Making them work was the second hurdle, and it was a significant one. The libraries we tested were *SDMetrics OpenCore*, *eclipse UML2*, *Apache Xerces*, *XMIParser* (by cqframework), *xmiparser* (a Python library). We only managed to get the **SDMetrics**

Java library to work within our research period. Since it adequately met our needs, we did not extend the research period to find other alternatives.

The table below traces user requirements to the parsing choices, refining them in this section.

Table 4: Parser Requirement Traceability

ID	User Requirement	Refinement
1	Input	Use SD Metrics OpenCore library to parse XMI file inputs.

3.2.5 Target Deployment

One of our user requirements requires us to target an Operating System. Just like the choice of modelling tool, we want to avoid the OS target to be a barrier to our tool. We also want to avoid incurring costs attempting to port our tool across multiple Operating Systems. **Java** is a great language that allows us to run the same program on any OS due to the Java Virtual Machine.

We considered *C*, but it is not as simple to make platform-independent. *Python* was also a valid alternative because it is also platform-independent. The deciding factor was the availability of third-party libraries. As mentioned in Section 3.2.4, the SD Metrics library we use for XMI parsing in our system was implemented in Java. We were not able to make any alternative library work in Python. With no particular need for other languages, we decided to keep it simple by only using Java in our program.

The table below traces user requirements to the deployment choices, refining them in this section.

Table 5: Deployment Requirement Traceability

ID	User Requirement	Refinement
10	OS Support	Use Java for platform independence.

3.2.6 Architecture Choice

Choosing how to structure our code depends strongly on our functional requirements, but quality attributes can play a role as well. The most relevant quality attributes for the architecture are scalability and performance. The other requirements are addressed at different steps of the design process.

To begin with, we can think about what our program does not need, like user interaction during execution. This already gets rid of the need for user interaction-focused patterns like *Model-View-Controller* and its variants. We also did not need a server or decentralized processing unless we failed to meet our performance goal on one computer. We did not expect performance to be an issue big enough to require decentralized processing. Thus, we can eliminate any server patterns, including *microservice* and *service-oriented* architectures.

The initial architecture we chose was the *Layer Pattern* because it fits quite well with the flow of our program. We are provided an input and do a series of unidirectional transformations on it to produce an output at the end. We believed these transformations made sense as different layers of our program. Layers could pass their outputs through defined input interfaces of the next layer. This is great for separation of concerns and maintainability. It is also a great way to reason about the system, allowing us to convert our understanding of C2KA transformation to code easily.

That said, we then realized we were starting to describe a **Pipe and Filter** architecture. Instead of layers to transform data, we use filters. We use pipes as our interface to communicate between them. We still have great separation of concerns and the same straightforward reasoning for implementation. The advantage we gain from this pivot is the ability to increase the parallelization of processing compared to the layered architecture. This allows us to directly improve our **performance** and **scalability** by reducing the impact of increased data set sizes.

The table below traces user requirements to the architecture choices, refining them in this section.

Table 6: Architecture Choice Requirement Traceability

ID	User Requirement	Refinement
11	Simple System Analysis Speed	Use parallelization enabled by Pipe and Filter to improve performance.
12	Worst Case Mitigation	Use parallelization enabled by Pipe and Filter to reduce scaling costs.

3.2.7 Architecture Description

To describe the concrete implementation of our design, we will break down the responsibilities of our filters and the pipes in our system through a visual representation. In these representations, the filters are the named rounded rectangles. The directional associations show data flow between filters. Annotations on the data flows specify what data is contained in the pipe. We also have forks, which are flows going into a black vertical bar and splitting, showing when data is re-used in parallel. Joins are the opposite operation, collecting data from independent parallel operations back into one centralized point. Finally, we have the cloud to indicate a simplification in the diagram. It explains which details were omitted from the model to make it easier to understand.

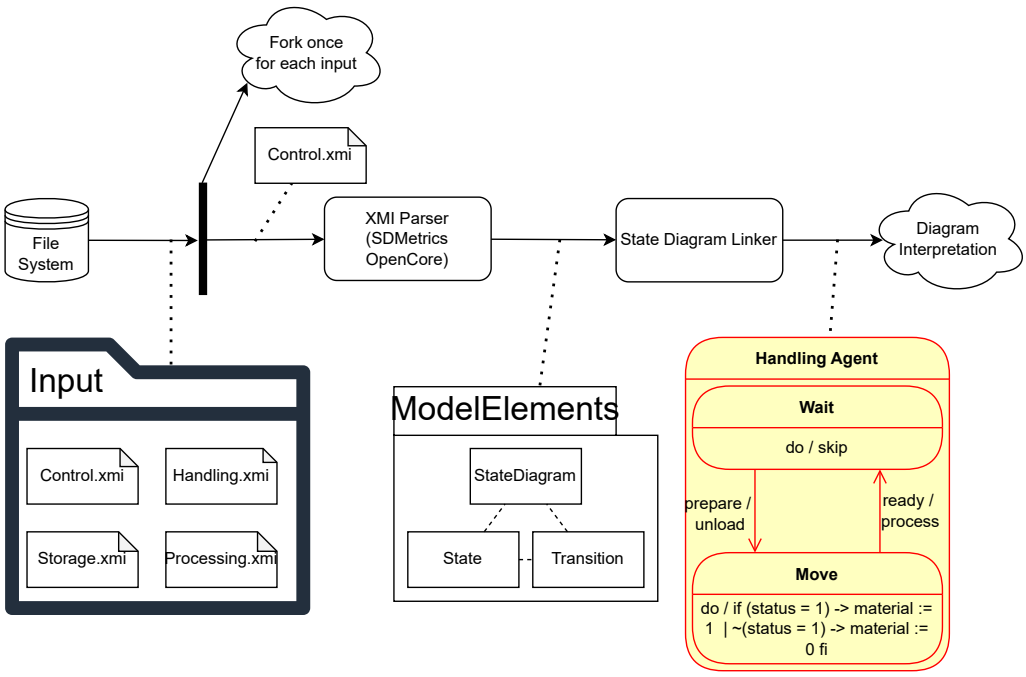

Figure 11: Converting an XMI file to an internal state diagram representation

Figure 11 shows the first part of our architecture model, which focuses on converting our expected input to a state diagram internally. When the program starts execution, it reads all the model files at once from a known input address. It forks once for each model to process them all in parallel. The first transformation step is to parse the *XMI file* with our **XMI Parser**. The parser uses the SDMetrics library to extract generic UML *ModelElements*.

These elements need to be passed to our **State Diagram Linker** to build an internal representation of a *state diagram* for our program. Specifically, we strongly type model elements according to our internally defined types (states and transitions). Super states can be thought of as tree roots, and a state diagram can also be represented as a super state. This allows us to recursively interpret our diagram just like a tree during the following **diagram interpretation** stage.

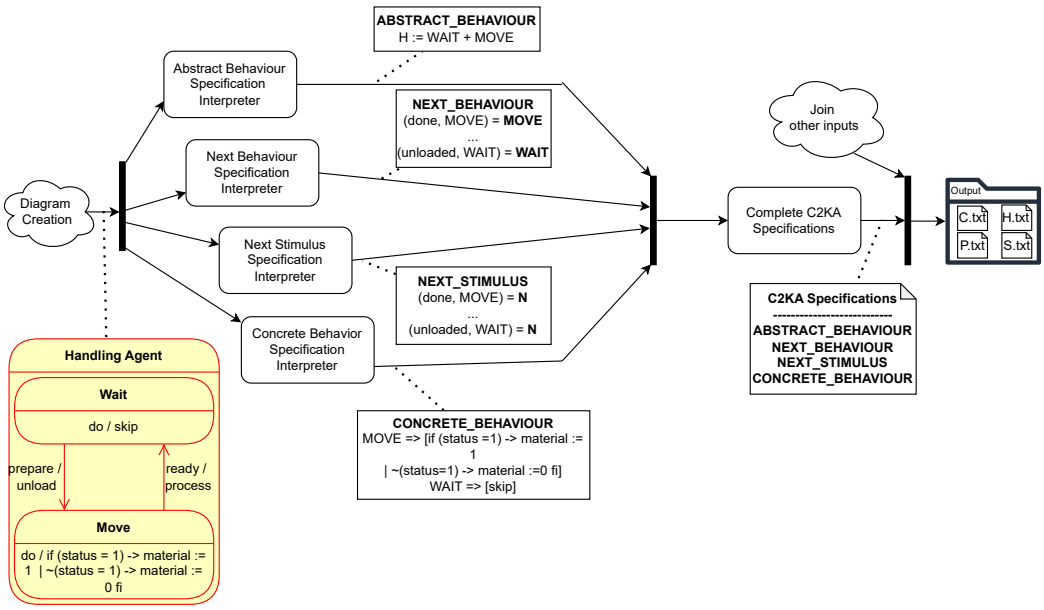

Figure 12: Interpreting a state diagram to derive a complete C2KA specification

Figure 12 shows the second part of our model completes the process, going from the internal state diagram to the desired output. From our *state diagram*, we can perform different analyses in parallel to find parts of the C2KA specification. The **abstract behaviour interpreter** looks at

super states to find sequential compositions and atomic state names to build a C2KA *abstract behaviour specification*. States that are not sequentially linked are composed by choice.

The **concrete behaviour interpreter** simply collects all the atomic behaviours and their doActivities. The doActivities should already be formatted as the concrete behaviour according to our input specification (Section 3.2.3). The C2KA *concrete behaviour specification* produced is just a formatted collection of the doActivities of the agent.

The **next behaviour interpreter** checks transitions to find the next behaviour mappings. A mapping is an initial behaviour, an input stimulus, and the resulting behaviour of the transition. The *next behaviour specification* is the set of all next behaviour mappings found for that agent.

The **next stimulus interpreter** is nearly the same as the next behaviour interpreter, but instead of the resulting behaviour, we check the output stimulus on the transition. The *next stimulus specification* is the set of all next stimulus mappings found for that agent.

Once all the interpretation is complete, it gets joined into a complete *C2KA Specification*. We then need to wait at a barrier for all other models to complete analysis before writing our final *output* to a file.

The reason for the barrier is due to a limitation inherent in C2KA of the next stimulus and next behaviour functions. They need to be complete functions, meaning behaviours in an agent need to have defined neutral mappings for all stimuli in the system. Neutral mappings are outcomes where nothing changes; a neutral stimulus is sent, or the behaviour remains the same. However, since the view of our system is fragmented across our input set, we cannot identify all stimuli in the system until we analyze all agent inputs to get a complete view of our system.

The table below traces user requirements to the architecture implementations, refining them in this section.

Table 7: Architecture Description Requirement Traceability

ID	User Requirement	Refinement
1	Input	XMI Parser, and State Diagram Linker used to convert input to a diagram format we can interpret.
2	C2KA Specifications	Interpreter filters are used to convert a diagram to partial C2KA specifications. They are then combined into a full C2KA specification.
4	Output	The complete C2KA Specification to output conversion allows us to extract agent text files as output.

3.2.8 Project Management

After user requirements, we described our design choices to cover them. From design, we needed to create tasks to trace the design to specific code implementation. To do so, we used **GitHub Issues** to build a list of upcoming tasks for the project. In Figure 13, we show a small sample of these issues completed throughout the project.

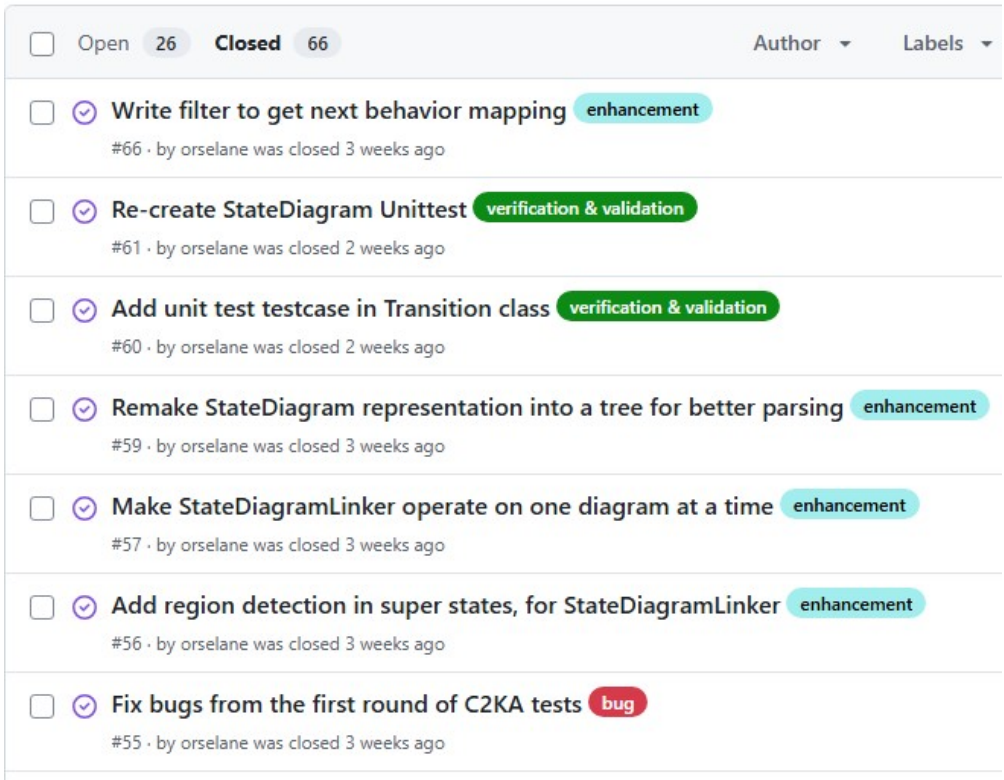

Figure 13: Sample Set of completed issues in our repository

These issues allowed us to assign them when resources were available

and communicate task dependencies easily. Work done by individuals can be tracked through issues, preventing multiple people from working on a task simultaneously unaware. The transition from weekly meetings to asynchronous communication through issues improved our efficiency immensely. We also customized issue tags to categorize issues according to our major development concerns, as seen in our repository in Figure 14.

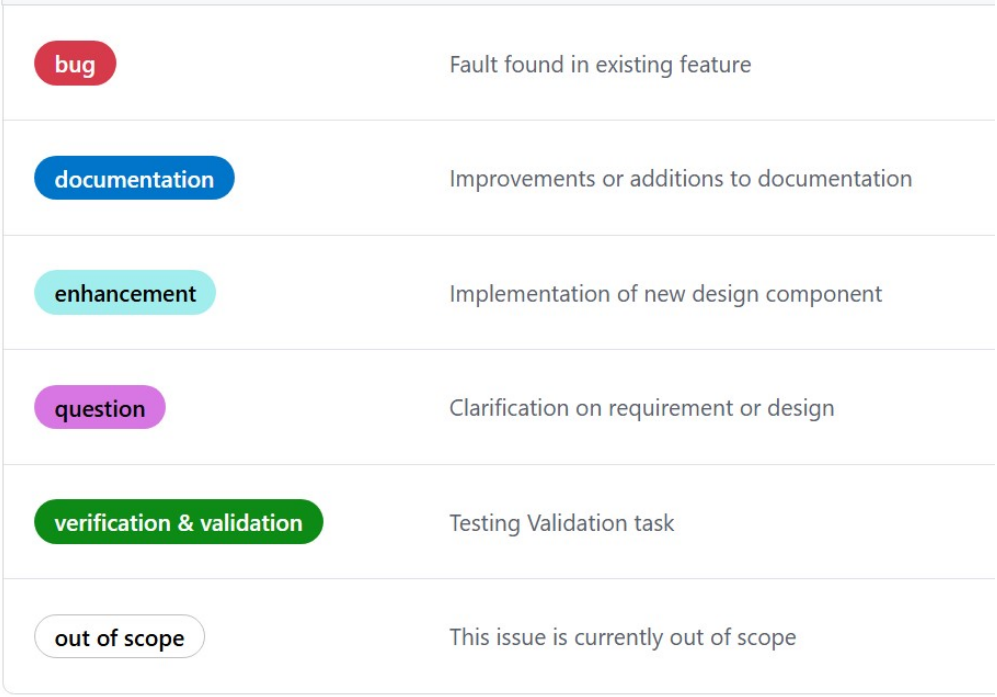

Figure 14: Types of issues defined in our GitHub repository

Enhancements and *bugs* both relate to implementing our design, but bugs are errors we found after merging the initial enhancement implementation.

Verification & Validation (V&V) are tasks related to verifying our code and validating it. Usually through creating new tests, but it can also include creating tools for testing, or establishing and documenting V&V requirements or strategies.

Questions are typically related to clarifications on requirements, or designs that require research or input from our supervisor. Once a question is answered, we typically reply directly to the question issue and close it. Usually, questions have a purpose and can create new issues or unblock

existing ones as well.

Documentation are tasks related to documenting our project for any audience. This includes documentation like this report, function descriptions in code, and a user manual.

Out of scope marks tasks that we identified as out of scope for our current release. These can include requirements we create to explicitly exclude from the start or existing issues we drop later due to time constraints. In the case where an issue is deemed irrelevant, we do not mark it out of scope. Instead, we close the issue with a comment for rationale and stop tracking it. On release, we reset the scope constraints by removing this label from all issues. After an evaluation of the goals of the next release, maintainers can decide which issues should stay within the scope of the next release.

We did not develop a great way to automatically trace issues to design, and we have too many issues to go through them all (over 60 closed issues currently). Unfortunately, this means we cannot rely on design to code traceability. Instead, we rely only on testing for validation. We can, however, demonstrate how we could attempt to show traceability manually through a specific issue. In Figure 15, we can identify from the title (1) the

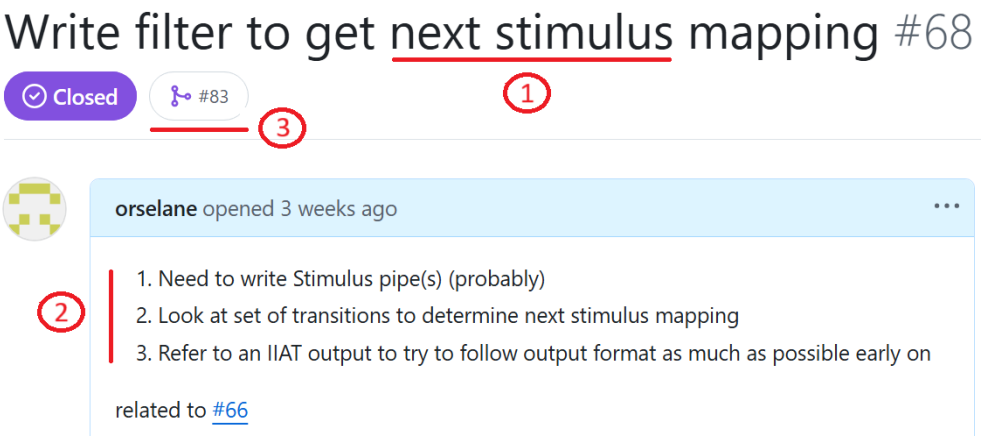

Figure 15: An annotated enhancement type issue

user requirement the issue contributes to. In this case, it contributes to the C2KA Specification transformations (User Requirement 2). Then we can look at the description (2) to see a lower level, but informal explanation to guide the code implementation. Finally, we see the merged branch (3)

linking the issue to merged code changes.

3.2.9 Testing Strategy

Although we never defined formal test selection criteria, We defined informal guidelines to define how to verify our code.

For our simplest code unit, *pipes*, we can **unit test** it. After construction, we verify that public attributes are as expected.

For *filters*, we did **integration tests** to simulate a real program environment more closely. For these tests, we called all the filters up to and including the filter under test and then evaluated the output. This allowed us to do iterative development when we developed new filters, ensuring new filter implementations were not breaking any previously implemented filters. Figure 16 shows the design behind the implementation of an integration test for the State Diagram Linker. The oracle object is a logical representation of our test cases, which define the expected outputs and compare them against the actual produced outputs.

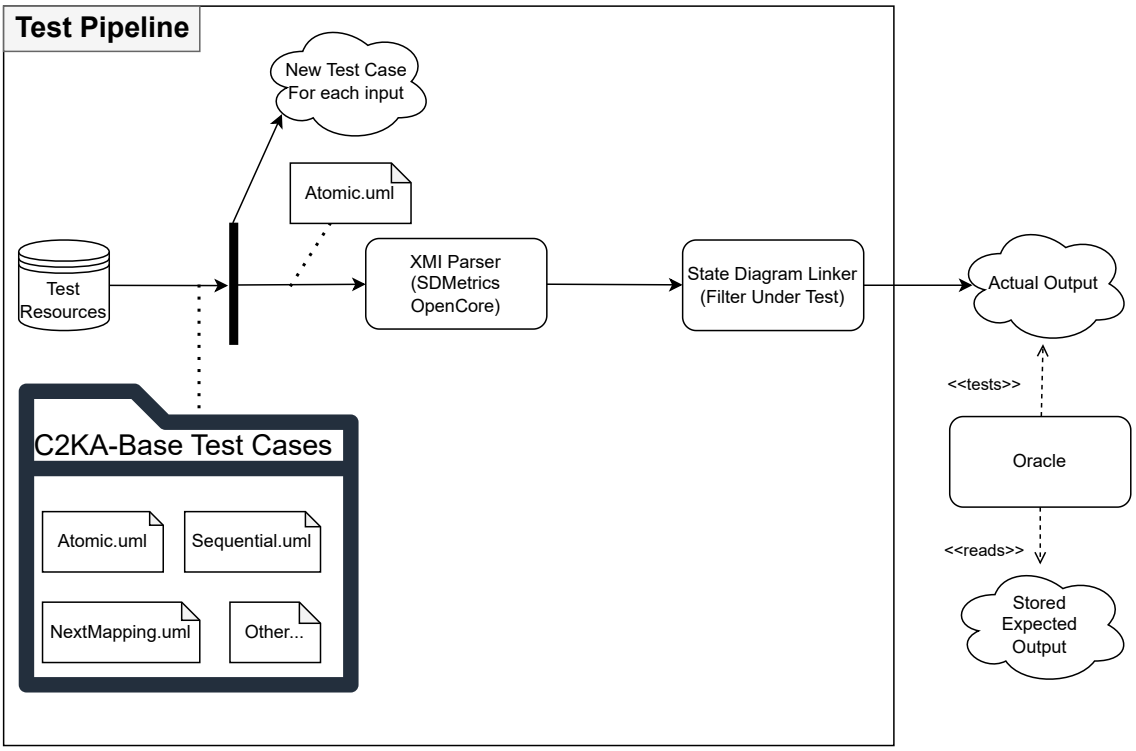

Figure 16: A visualization of the integration test design for State Diagram Linker

For integration test inputs, we defined a group of elementary C2KA representations that implement all the elements specified as part of our input specification. The key diagrams used are the same diagrams included in the input specification Section 3.2.3.

For the complete *C2KASpecification*, we did end-to-end **system tests** with a **custom diff tool**. To do this, we needed a system with known C2KA outputs. We got analyzed systems from our supervisor, which we treated as the source of truth. Then, we also needed to model diagrams representing those specifications. Finally, we create a test with the diagrams as inputs and the analyzed system as expected outputs and compare them with our diff tool.

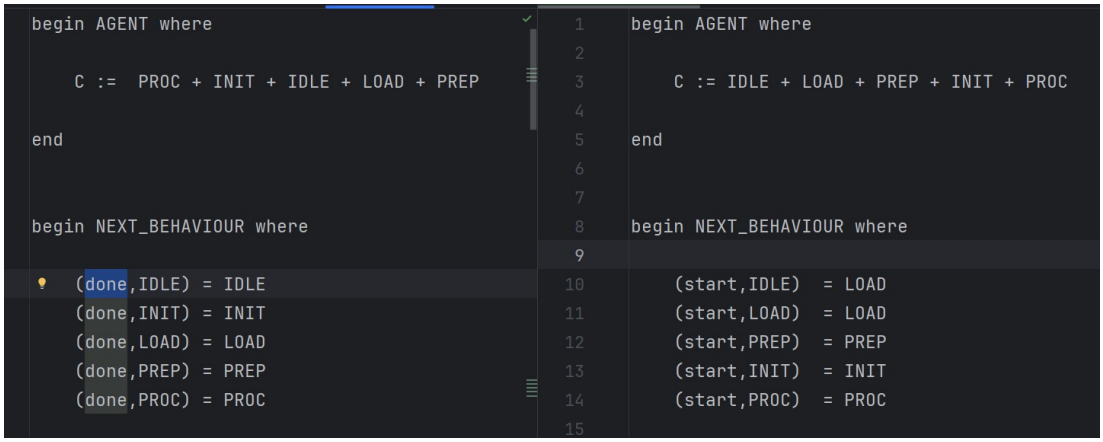

Figure 17: An example of the formatting differences between an actual output (left), and its expected output (right)

In Figure 17, we see that our produced output files are not the same as the expected output files. The most glaring difference is the ordering of lines in the NEXT_BEHAVIOUR spec. Our tool respects a strict alphabetical order, while the given expected output follows a different formatting rule. This difference is why we need the diff tool. Manual C2KA analysis does not follow deterministic formatting rules. There may be some whitespace differences or a different ordering of lines, which are both semantically irrelevant to the specification. This is why our diff tool checks for an exact match for lines in a specification type, regardless of their location within the block or any whitespace.

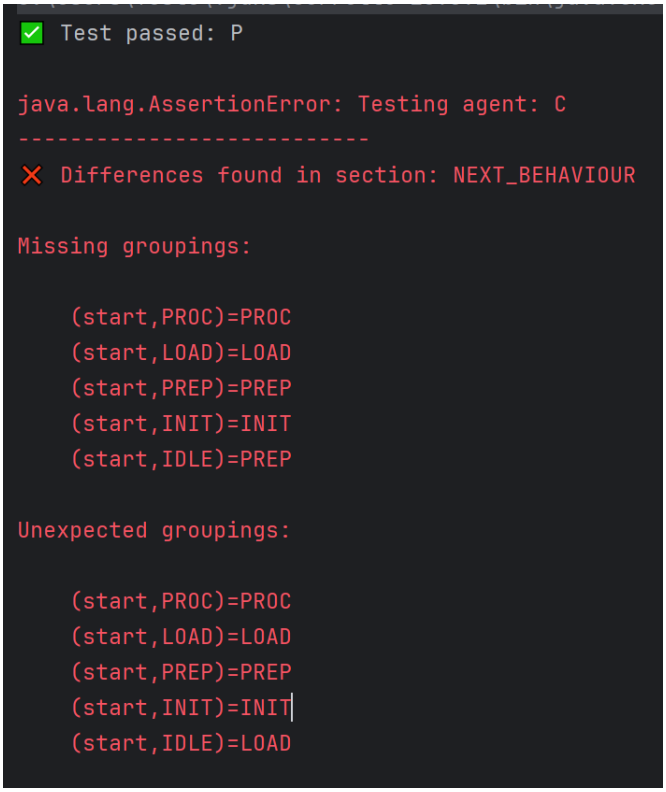

Figure 18: The test result after injecting a fault by modifying a line of the expected output

Figure 18 displays what happens when we inject a fault to force a failure. We can observe the success case at the top, displaying a checkmark for agent P. When failing, if our actual produced output is missing a line, the diff mentions a grouping is missing. If we have an additional unexpected line, the diff adds it to the unexpected grouping display. In the case of a modified line, it will appear in both categories. A limitation of the current diff tool is that it displays errors in groupings instead of displaying specific lines. Only the (start, IDLE) line was modified in Figure 18, but all start responses got flagged at once.

The Table 8 traces user requirements to our testing strategy concepts, refining them in this section.

Table 8: Testing Strategy Requirement Traceability

ID	User Requirement	Refinement
15	Verification Testing	Unit tests and integration tests provided confidence that our code was continuously functional as we progressed.
16	Accurate Outputs	The system tests and diff tool were part of our validation activities to verify output accuracy.
17	No False Positives	The system tests should show failures if an output could not be completed, otherwise, the diff tool should raise an error.

3.2.10 Validation Results

For our actual system *functionality*, it only makes sense to test it **transitively**. That is, we test it indirectly, in the context of other non-functional requirements. They serve as the metrics against which to evaluate how well that functionality is fulfilled. As such, we will analyze our validation strategies for the requirements we managed to fulfill.

We can safely assume our *Portability* is working if we manage to run the application. Ideally, we should have set up a continuous deployment environment for multiple target operating systems to ensure this property. For our current scope, we were satisfied with the assumption that **Java's JVM** was enough to ensure cross-platform support.

For our *Usability* requirement, we can verify it by looking at the usage instructions. As we will describe in Section 3.3.3, the user only needs to **double-click** the executable to run the program. To provide the models, they simply drop them in an Input folder before executing the program. We deemed this requirement fully fulfilled from this simple analysis.

For *Compatability* requirements, we will verify them by generating our **test inputs** from our supported choices. As long as our tests work using these inputs and we have sufficient tests, we prove the compatibility. It is only when we decide to support multiple tools that we need to generate more test case inputs from those other tools. We were not able to add support for multiple options for any of the compatibility requirements.

For the *Accuracy* requirement, this is where our **diff tool** comes into

play. We created accuracy-focused **system tests** as described in Section 3.2.9. With a trusted expected output, if we prove an equivalency with the files our tool generates, we can trust it is accurate (at least for that test case). Unfortunately, we only managed to complete one system test. Although we marked accuracy as adequately fulfilled, one sample test is not enough to be strongly assured of this property. Still, at least our accuracy validation is reproducible with our automated system test located in our repository under:

`src/test/java/sinks/ManufacturingCellTest`

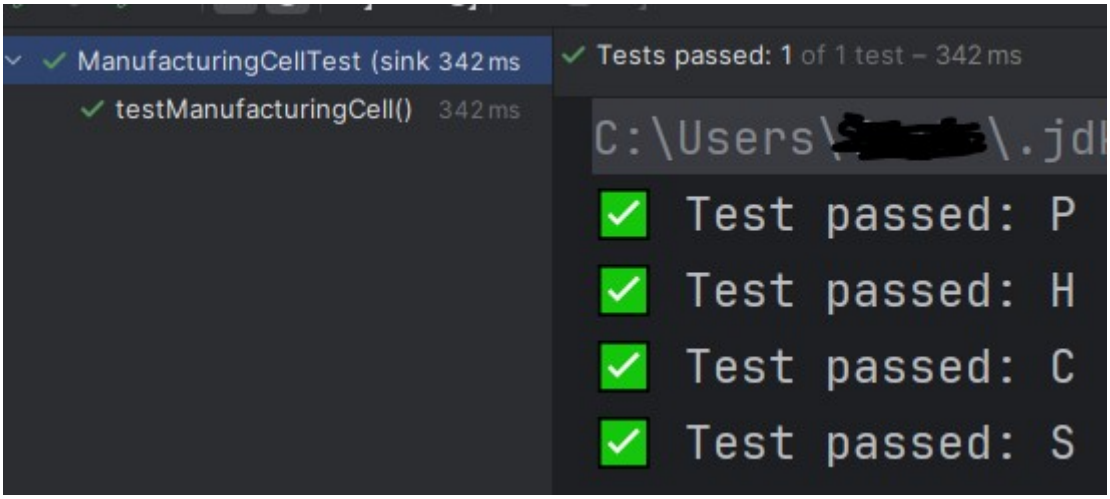

Figure 19: The accuracy system test for the Manufacturing Cell reporting a full success.

In Figure 19, we can see our diff tool reporting a successful C2KA generation for the Manufacturing Cell system. We did not manage to write test cases for our diff tool itself, therefore, it is valid to doubt the report accuracy. When debugging or when we were trying to initially gain confidence in the diff tool, we did manual diffs. This is a tedious process, and it is very easy to miss things. To verify the raw diagrams, we can look in the test folder's resources as uml files under ManufacturingCell. The test folder has the expected outputs as text files, which are the same as the ones from the IIAT repository [7]. We have a diagrams repository to store the additional files required to load them in Papyrus for visual inspection if needed [5]. If

we wish to compare the actual outputs generated in testing, they are generated under the Output folder. It is unsustainable for new users to do this as well to gain confidence. Tests for the diff tool itself are important. It is a critical part of the tool chain, and validating it should not be ignored in future work.

The table below traces user requirements to our testing results, refining them in this section.

Table 9: Validation Requirement Traceability

ID	User Requirement	Refinement
1	Input	Tested transitively as part of other validation activities.
2	C2KA Specification	Tested transitively as part of other validation activities.
4	Output	Tested transitively as part of other validation activities.
5	Minimal User Actions	User simply double-clicks to fully execute the program once configured.
6	Modelling Tool Support	Test inputs generated using chosen modelling tool.
7	Modelling Language Support	Test inputs generated using chosen modelling language.
8	Diagram Type Support	Test inputs generated using chosen diagram type.
10	OS Support	Rely on the Java JVM for built-in cross-platform support.
16	Accurate Outputs	System tests using the diff tool for comparison checks against expected verified outputs.

3.2.11 Unsatisfied Requirements

As we progressed in the project, some lower-priority requirements had to be cut to reduce the scope.

For functional requirements, this meant the **IIAT Parameters**. Although these are nice to have, they may have required support for an additional diagram type for small benefits. The core goal of our program was to produce the C2KA Specification transformation.

For non-functional requirements, we completely dropped **cross tool in-**

tegration. It seemed complex, yet not essential to prove that one version of our program was functional. We also cannot claim **performance** nor **scalability** of our program due to lack of testing or proofs. In theory, these properties should hold with our current design, but we did not take the time to prove it. We did not prove that we prevent **False Positives** either. False positives are concerned with preventing indeterministic outputs from being generated. This means that to satisfy this requirement, we needed to prove that errors get raised in diagrams, which lead to indeterministic interpretations. Our system tests are only focused on accuracy; they assume an output is generated but may be inaccurate.

We also completed some non-functional requirements but in a way that could be improved. Our parser allows for multiple **modelling tool support**, but it still needs to be tested. Similarly, we could have potentially supported multiple **modelling languages** with little extra effort. The same may apply to **diagram types**. All of this requires additional tests and may lead to small tweaks, increasing our scope beyond what we could handle.

User Documentation could be improved, but we prioritized writing a comprehensive report instead. For now, we believe a user could use our tool with the readme provided, but may struggle to use it without aid from the Usage in Section 3.3, and Input specification in Section 3.2.3.

Maintainer documentation is good, but it is not systematic. There is no automatic way to detect missing documentation to ensure full functional coverage. It may also have been good to have a maintainer manual alongside the user manual. It could explain design decisions and the program architecture at a lower level than the report for overarching context. For now, this report serves adequately for context behind the design and goals of the system.

Our custom diff tool was also not tested on its own. This means the reports we rely on for accuracy validation may be inaccurate themselves due to an unknown fault in the diff tool. If we had more time, getting better assurance of our diff tool itself would increase our assurance in the

program's **Accuracy**.

In summary, OS Support, Usability, and the main functions are the only requirements we have fully covered (green). All other requirements are either adequately covered (also green) or not covered (gray), as seen in our user requirement table, table 1.

3.3 Usage

3.3.1 Demonstration Scope

The demonstration will be using inputs generated to simulate the Manufacturing Cell system seen in Section 3.1.1. The IIAT tool is pre-configured for this system, and the sample inputs from our V1 release is for this system as well.

To use our tool, we will show the full process from how to generate the inputs up to how to provide them to our target model checker (IIAT) to see the output. We will assume the reader is familiar with the input specification from Section 3.2.3, and how to create their desired state diagram following the specification. Therefore, we will start by showing how to export an existing state diagram in Papyrus.

Recall that we already deemed the IIAT-specific properties out of scope in table Section 3.2.11. For this version of our program, we do not take responsibility for IIAT configurations to run any Model. We only wish to demonstrate the C2KA specifications are accurate, and can be used irrespective of the modelling tool. Showing parity with the existing samples for IIAT is the limit of our demonstration.

3.3.2 Exporting Inputs from Papyrus

The first step in providing inputs is to export them from Papyrus. Figure 20, Figure 21 and Figure 22 will demonstrate the steps required to do so:

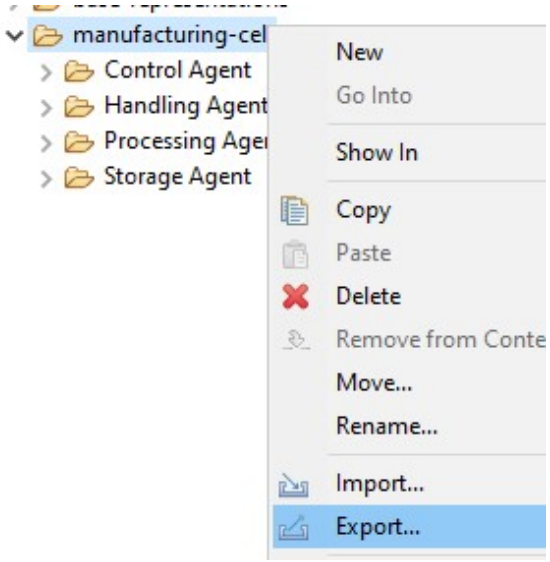

Figure 20: Step 1, open export wizard in Papyrus

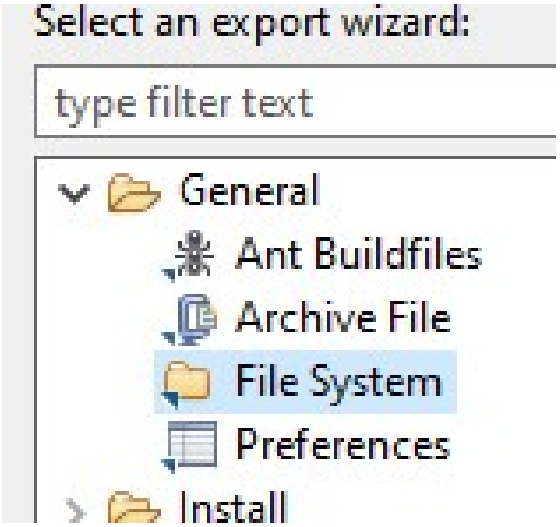

Figure 21: Step 2, select file system export

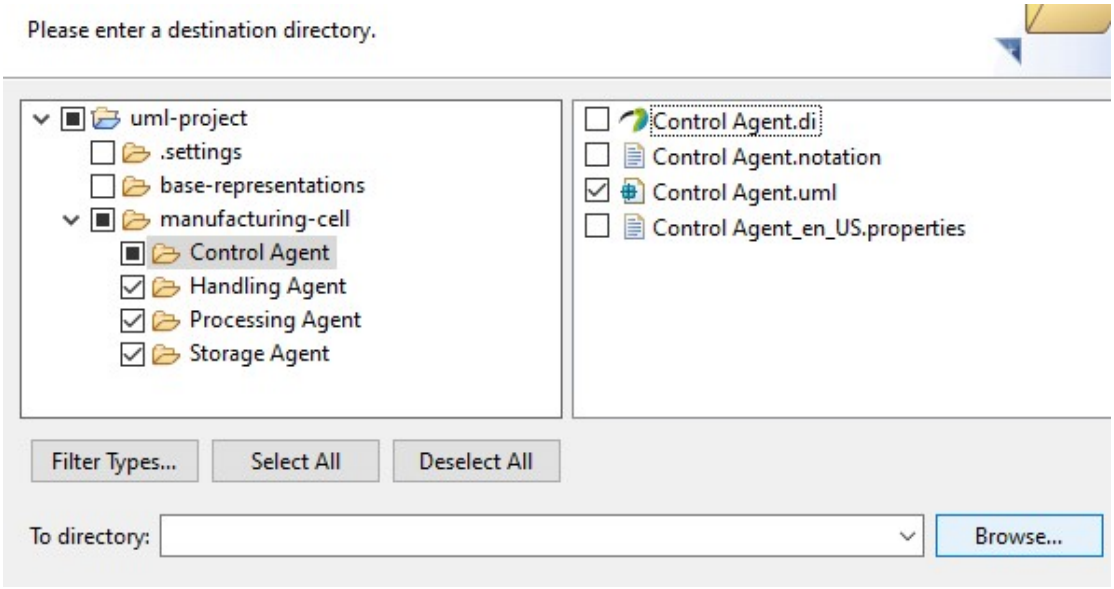

Figure 22: Step 3, Select an output directory, and optionally unneeded files.

3.3.3 Executing U2C

To use our program, download the 1.0 zip release from our GitHub [6]. Then, extract the zip as is to a folder. Double-click the jar file to execute it (Java 16+ required). Open the Output folder, there should be four output files corresponding to the sample input.

If the output does not get generated, the program did not succeed. This is most likely due to an improper Java installation. Attempt to run the Jar file in the console; for example, on Windows, the command is:

```
java -jar U2C.jar
```

This should display some error message which is hopefully helpful enough to troubleshoot. Otherwise, raise a bug on GitHub with steps to reproduce it. Optionally, contact an individual currently maintaining the tool for help.

If you had no issues, collect the output for use in later steps of this demonstration. We will use the sample inputs to make the demonstration reproducible. To analyze your own system, you would remove all files in Input, Output then put your own ".uml" files in Input. Then, execute the program and collect the outputs, just as we did with the sample.

3.3.4 Running the target Model Checker, IIAT

To build and run the IIAT, follow the instructions on the IIAT repository [7] to install its dependencies first. We recommend avoiding Windows, it tends to be more difficult to work with. The IIAT has native options for both macOS, and Linux. For this demonstration, we ran the IIAT on an Arch Linux distribution. The next section assumes you have completed the *Installation* section of the IIAT repository's readme file.

Before running the program, it needs to be compiled locally. When we tried to do so on our Linux installation, the compiler was having trouble loading *hidden* packages. These were packages that were installed but not explicit dependencies of the project. To fix this issue, we went into the Makefile and changed the HSFlags variable to add the hidden packages:

```
HSFlags = -O2 --make -w -package parsec -package vector -package split
```

Admittedly, there must be better fixes, but this was the first time we touched Haskell. This solution worked well enough to compile and run the program.

Before execution, we also need to configure Maude. Again, we followed the instructions in the source repository. The only trouble we had this time was finding the file. It is not specified, but it is located at /src/MaudeInterface.lhs.

For our particular Arch Linux installation, we also had to add a missing library. Specifically, we were missing "libtinfo.so.5". We had a "libtinfo.so.6" library as a dependency of ncurses (which we already had installed). It is backward-compatible with the old version, so the solution was to create a symbolic link like so:

```
sudo link /lib/libtinfo.so.6 /lib/libtinfo.so.5
```

To verify that the program works, run the executable with:

```
./ImplicitsInteractionsAnalysisTool
```

The pre-configured system should be the manufacturing cell. If it worked, it should display the intended interactions. Afterward, the IIAT starts computing the implicit interactions and printing them as it goes. The tool is very verbose; as such, we only show the output up to the intended interactions in Figure 23. Note although the figure does not capture the shell, it shows the contents of the real output. We were lacking a screenshot feature in our host, we copied the output text instead.

Note: We did not manage to pipe the output to a file; it currently has to be observed as it prints. It seems to spawn subprocesses that direct the output to the console directly after the Intended Interactions are printed. What we did manage to pipe lost its formatting and became difficult to read. Whichever the case, we believe it to be a limitation of the Linux implementation. There may be a way through shell commands to redirect the output, but we are not familiar enough with bash.

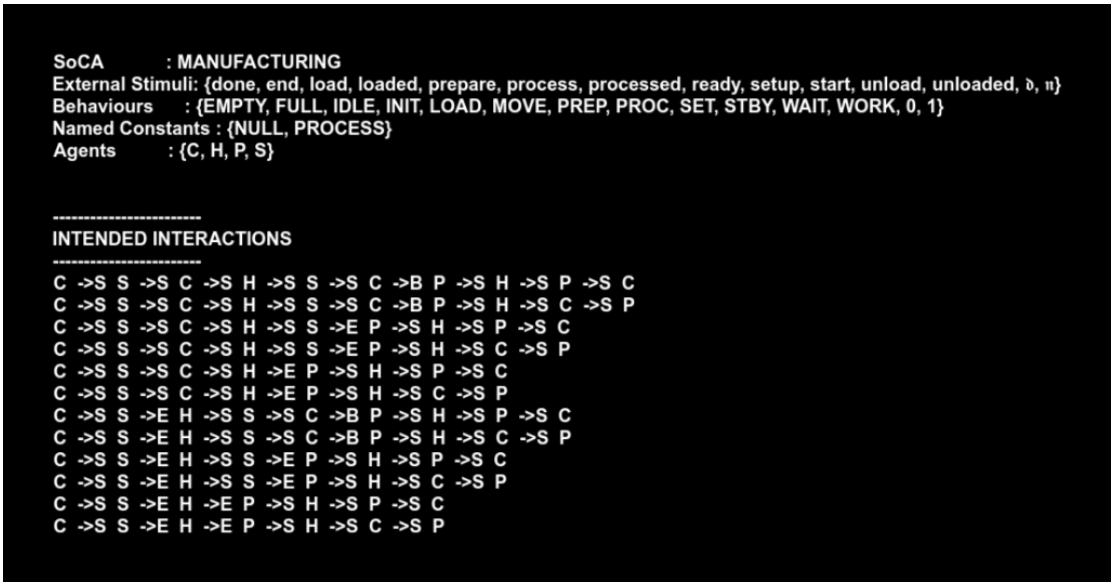

Figure 23: The intended interactions found by the IIAT on the expected input (text copied)

3.3.5 Providing generated specifications to IIAT

Since the IIAT is pre-configured for the Manufacturing Cell, We can take advantage of that for this demonstration. Grab the outputs we generated in Section 3.3.3. Replace the files under specs/ManufacturingCell with the outputs generated from our sample input diagrams. The sample input models the Manufacturing Cell, therefore, they should be semantically the same. You should be able to observe the same outputs from Section 3.3.4 when executing the program again. Again, due to the verbosity, we only check and display the intended interactions in Figure 24. The output observed was identical to the expected output in Figure 23. To analyze different models, additional inputs need to be provided to the IIAT. It does not seem that the IIAT tool explains how to configure the system in the readme. As such, doing it ourselves for demonstration may require significant exploration. This is why we deemed it out of scope when we discussed the demonstration scope in Section 3.3.1.

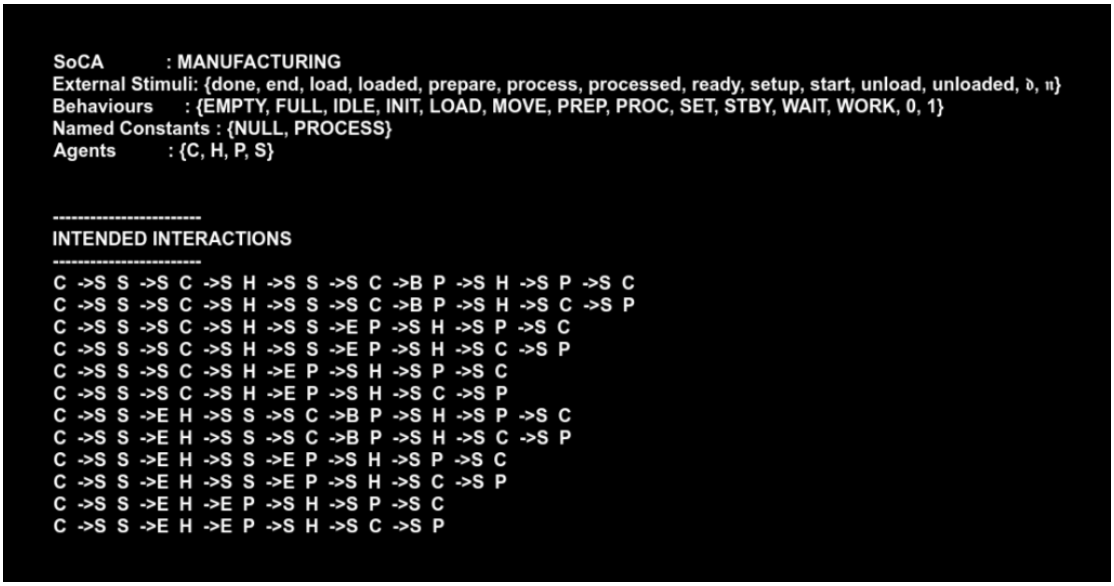

Figure 24: The intended interactions found by the IIAT on our generated specifications (text copied)

4 Conclusion

We wanted to find a way to produce C2KA system specifications more easily. C2KA specifications are a useful formalism to analyze systems, but they take time and knowledge to produce. As such, other methods to gain assurance in a system are used. These are typically lower in cost but cannot provide the same degree of assurance (like unit testing code).

We believed that we could leverage a more universal skill than C2KA specification writing: Creating visual diagram representations of the system. Visual diagrams have the added benefit of being artefacts that are useful on their own already. This means that users of our tool may already be using them or benefitting from their creation.

The prototype we've built takes in UML state diagrams and can produce C2KA specification files for each agent in the system. On their own, they can provide a comprehensive overview of the possible behaviors of an agent in the system. When fed to other analysis tools like the IIAT, these specifications can also be used to analyze the system for implicit interactions. These implicit interactions may be the source of hidden vulnerabilities in the system, which we can now help detect easily.

4.1 Further Recommendations

There are a few gaps in our prototype compared to our requirements. For an up-to-date detailed account, our GitHub issues should be the best source in case of any changes after this report was written. For consistency, we will follow the issue categories we used in GitHub to discuss our recommendations. See section 3.2.8 for definitions of these categories.

4.1.1 Verification & Validation

We are lacking a formal definition of coverage criteria. The test strategy we defined does a good job of capturing the idea behind how we try to cover tests. Formalizing coverage criteria would be taking these ideas and formulating them in a way that we can systematically test our code coverage and build a report to see if we conform to our desired coverage. This report would also be an important baseline for assurance in our software. Checking coverage is done manually at the moment, we create GitHub issues if we notice missing coverage (easy to miss).

We also lack system test (validation tests) variety. It would have been nice to have system tests to test other metrics than the "Accuracy" of our system. We have no stress tests for performance, for example. This is partially due to a late definition of our non-functional requirements, as explained later in section 4.2.1.

In general, more tests are needed. We are currently lacking coverage (verification). More system tests increase assurance in our system (validation); we have very few of them at the moment.

4.1.2 Enhancements

The main feature we lack is collaboration diagram processing for IIAT parameter extraction. We cannot produce an intended sequence of inputs for the IIAT, which was part of our initial set of functional requirements.

Another notable mention was performance improvement-related fixes. Our testing did not yet support proper analysis for these changes; thus,

work on them did not get started either.

We entertained the idea of having a separate model checker module as part of our tool. We could have a separate pipeline that could be called independently to check the validity of the inputs. We would define rules in the module to check the assumptions we defined in section 3.2.3. This would help prevent potential unexpected behaviors (false positive outputs), and it could help improve the error messages we give to users to make fixing user diagrams easier. The side benefit is that by trying to make the artefacts work for our tool, we may improve diagram quality by automatically reviewing them through our model checker.

4.1.3 Bugs

There are none (that we know of!).

In this context, we will count bugs as faults that could lead to an incorrect output being produced. If the program crashes before producing the output, we may count it as a missing feature, depending on the cause. A well-formed input that crashes means we have not developed the tools to process it yet. Collaboration diagrams could fall in this category, although we have not yet defined what is a well-formed collaboration diagram either. Otherwise, it is good if the program crashes because it means that it detected a fault in the input. If it did not, it would be a bug because the output is faulty (false positive output).

Our philosophy as we were finishing our prototype was to focus on a minimum viable product with well-tested essential features. We did not want to add many unessential features. We knew we would not have the proper time to identify faults if we increased our scope. This could have led to unsatisfied critical non-functional requirements, like accuracy. The goal is that we may not be able to output everything, but what we do produce can be relied on (no false positive).

4.2 Reflections

4.2.1 Requirement Elicitation Phase

This section refers to how we collected background information and determined our requirements from there. This does not refer to how we managed requirements after we started the implementation. The issue with our approach is two-fold. We took too much time (we did not start implementation work until after the progress report), and it was not productive given the time spent on it. Although the User Requirement list in section 3.2.1 is adequate, it was never formally defined until the report was written.

At multiple times during implementation, progress halted due to a lack of C2KA knowledge, or we had to rewrite code from a poor initial understanding. We also did not approve or trace any requirements other than our functional objectives. These functional objectives were based on a specific pipeline configuration, which we decided to change later, and thus were too low-level to base our functional requirements on them. The non-functional objectives of our system became even more implicit and were subject to change since there was no traceability for them. This means any decisions regarding trade-offs in the program were decided based on what felt best at the time, not an objective metric measured against non-functional requirements.

Upon reflection, we learned that our requirement elicitation should have been more targeted and produced a Software Requirement Specification document that we verified and agreed to conform to. We should have made a process for ongoing requirement elicitation during development. In the case of a knowledge gap being identified, we would research or ask questions to fill this gap, then produce requirements and update our SRS accordingly. This turnaround should not take more than a week to avoid blocking or reverting changes in the implementation. The initial requirement elicitation phase may warrant a bit more time allocated. It takes time to develop this process and learn about the problem domain. It should not have taken us

more than a month, it is better to fail and identify knowledge gaps earlier than be as inefficient as we were.

4.2.2 Timelines

There were a few problems with the initial timeline from the proposal (see Appendix A). The first one was keeping an optional objective and excluding it from our timeline. Our timeline represented the worst-case scenario to still deliver a prototype. This meant our target of a high-quality project required us to outperform the timeline we proposed.

We also did not include any slack-time in our project, we included it by allocating extra time instead. This meant components that had six weeks allocated to them were intended to be implemented in potentially a week or two in reality. This, and keeping the optional objective implicit, made us quite lax in the early stages of the project. It ultimately affected the timeline negatively and impacted the feasibility of the project.

The timeline was also based on two assumptions that did not hold and made it impossible to achieve. We had agreed as a team to spend a set amount of hours regularly on the project. This assumption was never held throughout the project. We also based our timeline on a specific scope, architecture, and components. The moment implementation started, we rewrote most of it because we changed our target diagram type.

4.2.3 Well-Defined Testing Criteria

As mentioned in our further recommendations in section 4.1.1, our testing could use improvements. Ideally, after designing our architecture, we could have defined test selection and coverage criteria. If we agreed on it, we could have had an automated mechanism to check our test coverage to see if we're missing tests. This should have helped us improve our test coverage and build a better case for assurance in our tool.

This would massively improve our verification process. It would also help with validation because it is much easier to trace well-defined test

criteria. It should also indicate to us early on when we are missing system tests for validation. In hindsight, we realized quite late what sorts of tests we could make that are not directly linked to the functional output. These are related to properties like portability, compatibility, and scalability, for example.

References

[1] Edsger W. Dijkstra. *Guarded Commands, Nondeterminacy and Formal Derivation of Programs*. 1975. URL: http://www.cs.toronto.edu/~chechik/courses05/csc410/readings/dijkstra.pdf.

[2] Canadian Centre for Occupational Health Government of Canada and Safety. *Office ergonomics - positioning the Monitor*. May 2024. URL: https://www.ccohs.ca/oshanswers/ergonomics/office/monitor_positioning.html.

[3] Jason Jaskolka. *Distributed Manufacturing Cell Control System - C2KA System Specification*. 2020. URL: https://gitlab.com/CyberSEA-Public/ImplicitInteractionsAnalysisTool/-/blob/master/doc/ManufacturingCell/ManufacturingCell.pdf.

[4] Jason Jaskolka and John Villasenor. *Identifying Implicit Component Interactions in Distributed Cyber-Physical Systems*. 2017. URL: https://scholarspace.manoa.hawaii.edu/server/api/core/bitstreams/ee616add-f1b6-46fa-85d6-cd1069aef9c0/content.

[5] Alexandre Marques. *U2C test diagrams*. 2025. URL: https://github.com/4907-49-2024/diagrams.

[6] Alexandre Marques. *UML to C2KA Converter*. 2025. URL: https://github.com/4907-49-2024/U2C.

[7] CyberSEA Research Lab. *Distributed Manufacturing Cell Control System - C2KA System Specification*. 2021. URL: https://gitlab.com/CyberSEA-Public/ImplicitInteractionsAnalysisTool.

[8] Jason Jaskolka, Ridha Khedri, and Qinglei Zhang. *Foundations of Communicating Concurrent Kleene Algebra*. Feb. 2015. URL:

https://carleton.ca/jaskolka/wp-content/uploads/CAS-13-07-RK.pdf.

Appendices

A Project Proposal

Converting Graphical Models to Formal Specifications

Students: Ahmed Babar (101154651), Michael Rochefort (101080282), Alexandre Marques (101189743)
Supervisor: Prof. Jason Jaskolka

Table of Contents

1 - Introduction

1.1 Complex Systems and the Use of Graphical Models

Complex systems rely on the coordination of multiple interacting components to function as intended. Designers commonly use graphical models such as UML (Unified Modeling Language) and SysML (Systems Modeling Language) to specify these interactions. These models provide a high-level visual representation of a system's intended design, outlining the expected communication pathways between components. However, even with careful planning, implicit interactions—unforeseen sequences of communication between components—can arise.

1.2 Problem Statement: The Challenge of Implicit Interactions

Implicit interactions, which are interactions that deviate from a system's intended design, are difficult to detect and can lead to vulnerabilities, especially in distributed and concurrent systems. These hidden interactions may go unnoticed during the design phase, potentially resulting in system instability or exposing vulnerabilities that attackers can exploit [1]. Detecting implicit interactions typically requires formal analysis, but translating graphical models into the necessary formal specifications is often a manual, labor-intensive process that introduces errors and limits scalability [1].

This manual process creates a significant barrier to the widespread adoption of formal methods and tools in system design. System designers often lack the time or expertise to convert their high-level graphical models into the formal specifications required for implicit interaction analysis. As a result, implicit interactions may go undetected until much later in the development lifecycle, increasing the risk of vulnerabilities and system failures [1].

2 - Background

2.1 Communicating Concurrent Kleene Algebra (C^2KA)

To rigorously analyze implicit interactions, the Communicating Concurrent Kleene Algebra (C^2KA) framework extends classical algebra by incorporating communication actions and concurrency. This framework allows system designers to model agent behaviors as formal specifications, making it possible to capture both intended and unintended interactions [2]. C^2KA's mathematical structure is well-suited for distributed systems where concurrency plays a critical role, enabling the detection of implicit interactions and identifying potential vulnerabilities early in the design process [2].

2.2 Partial Solutions: The Implicit Interactions Analysis Tool

While C^2KA provides the theoretical basis for addressing implicit interactions, practical tools are necessary to make its application feasible for designers. The Implicit Interactions Analysis Tool can analyze formal specifications for implicit interactions, but it requires formal inputs that system designers rarely produce. Most rely on UML/SysML models, meaning the conversion to C^2KA-compatible specifications is typically manual. This gap between graphical modeling and formal analysis presents a barrier to effective implicit interaction detection, as designers often lack the resources for this translation step.

2.3 Parsing XMI Files for Data Extraction

XMI (XML Metadata Interchange) files offer a standardized format for representing UML/SysML models, which is crucial for transforming graphical models into formats suitable for analysis. However, variations in XMI implementations across different UML tools can complicate parsing, making it challenging to create a universal parser that consistently handles diverse outputs.

To address this, various parsers and libraries have been developed to process XMI files by converting them into structured representations that allow for easier extraction of essential model data. These libraries typically work by deserializing XMI data into object-oriented representations, enabling smoother handling of different UML tool outputs and facilitating further processing. By leveraging such parsing tools, we can automate data extraction from XMI files, laying the groundwork for their conversion into formal C^2KA specifications.

2.4 The Need for Automation

The current reliance on manual translation from graphical models to formal specifications limits the accessibility and scalability of implicit interaction analysis tools. Automating this translation process is crucial, as it will:

- Reduce the time and effort required to generate formal specifications.
- Increase accuracy by minimizing human error.
- Encourage adoption of formal analysis tools by improving usability.
- Strengthen security by making it easier to detect implicit interactions during the design phase.

3 - Proposed Solution

Our project aims to create a software solution that automates the extraction of interaction data from UML/SysML models and converts it into formal specifications using C^2KA. The program will serve as a bridge between high-level graphical models and the Implicit Interactions Analysis Tool, which requires formal inputs for analysis. By automating this conversion process, our solution will enable system designers to move from model creation to formal analysis efficiently, reducing the risk of undetected implicit interactions and providing a more secure system design process.
Key components of the solution:

- Input: An XMI file exported by a UML/SysML modeling tool, which contains the model's structure and interactions.
- Processing: Automated parsing of the XMI file using libraries to extract relevant interaction sequences and convert them into C^2KA specifications.
- Output: A formal specification compatible with the Implicit Interactions Analysis Tool, enabling systematic analysis for implicit interactions.

4 - Objectives

O1: Parsing XMI

Parse an XMI input into a semantically identical model internally, ignoring aesthetic details. This objective includes choosing a method to generate an XMI input, a strategy to parse it, and a simple way to view the result.

Measuring: We would start by building a model containing all the model elements we may expect to find in a **behavioral model**. We would then build a list of expected components, and compare the parsed model with the expected components. By the end of this objective, the test should be automated if possible for continuous integration.

O2: Deriving intermediate model traits

These are traits that are implicit in the model, and used as a means to build our final outputs. Model traits refer specifically to *agents*, *behaviors*, and *stimuli*.

Measuring: We would re-create a model with a known C^2KA analysis, based on academic papers analyzing existing systems with this method. We would then do manual comparisons during the initial prototypes, but requiring an automated test strategy before completing the objective.

O3: Derive specifications.

Using the accrued model information, build the desired specifications. These are the *Abstract Behavior Specification, Stimulus-Response Specification, Concrete Behavior Specification*.

Measuring: We can re-use the known C^2KA example to compare specifications. We should encode the known specifications into automated tests.

O4: Format Output

Format output for Implicit Interactions Analysis Tool (*IIAT*). This requires packing the specifications in a format that the IIAT can use. The initial part of this objective relates to figuring out what format the IIAT requires.

Measuring: To test this step, we should compare our output file against a known correct input file for the same pre-analyzed C^2KA model. If it is semantically equivalent, and syntactically correct, we have completed **O4,** and all previous objectives. Additional confidence can be achieved by running the IIAT with our produced input, and re-producing a known correct analysis for the input model.

O5: Improve Tool (Optional)

Optional objective, commence if and only if we complete O4 before February.

Re-asses project time, and tool state to define tool improvement objectives. We keep the objective vague to force us to elicit new requirements and prioritize new objectives later. That said, there are important concerns we are likely to want to address at this point:

- Improving tool confidence (adding error detection, error correction, increasing accuracy with additional inputs).
- Improving tool convenience. User experience, deployment options, and tool interoperability are some examples in this category.

Measuring: Depends strongly on project state, but we will review our plan with our mentor. Measuring strategy will depend on chosen additional features: Tool Confidence will increase the existing test set comprehensiveness. Performance requirements would require stress testing. User experience tests will likely be tested manually, ideally with our target clientele.

5 - Solving Methods

5.1 High Level Architecture

Figure 1 shows the high-level simple pipe-filter architecture of our software components on the right, and implementation details on the right. We can identify our initial approach to the problem with this figure, breaking it down into sequential steps with minimal interfaces for clear separation of responsibilities between components.

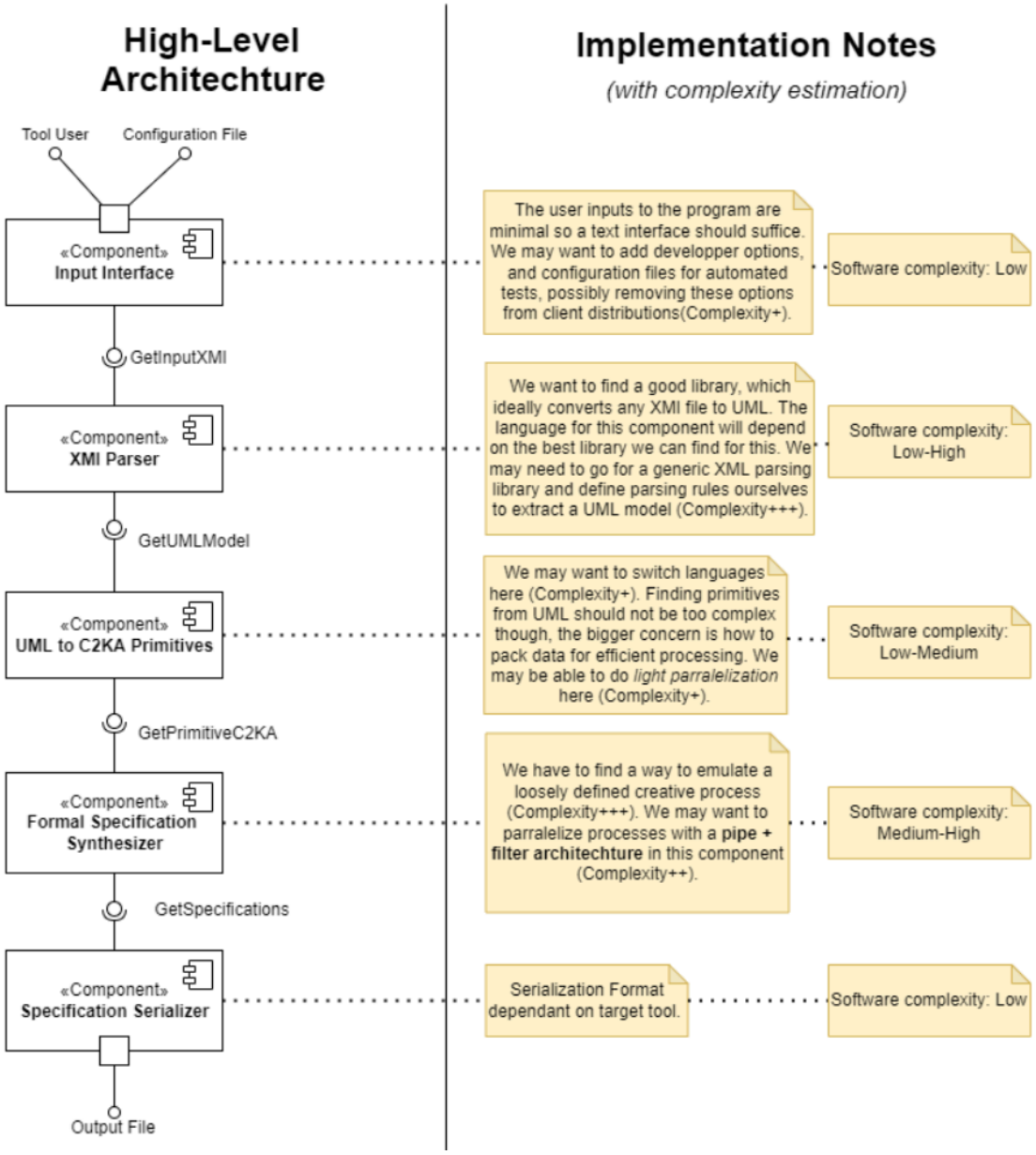

Figure 1 - (Left) High Level Component Diagram Architecture. (Right) Current implementation ideas, and how they relate to expected complexity.

5.2 Linking Architecture To Objectives

The high-level architecture follows the objectives by mapping them to components one-to-one, with the exception of the optional objective, since it is outside the initial scope. It lets us achieve objectives progressively, by building our components sequentially due to their dependency on each other. The proposed set of tasks to build these components is as follows:

Input Interface

We want minimal complexity for this part initially. We may be able to cleanly integrate all components in each other. Alternatively we might need to separate them into individual processes. The script would have to pass the input to the parser component to start the program. A small bash script should suffice for this purpose, offering flexibility as well.

Tasks:

1. Setup development environment: Choose OS, setup VCS, setup CI pipelines.
2. Define interface requirements: Required inputs, and which programs are the inputs sent to.
3. Create a prototype script: It should be able to take input, and start a placeholder program which prints back the input.
4. Add a test trigger option: The test for this script is step #3.
5. Implement the next component, then update this script to start the component with its relevant arguments. (further implementation details omitted, component language may affect how it gets started).
6. Repeat step 4, step 5 as new components are developed. May require adding test trigger support, for more granularity options or linking tests to existing options (step 4). May require a new process to start and pass arguments (step 5).

XMI Parser

This step is the most likely to have some level of work done by others in the past, and we want to leverage this possibility by looking at libraries (likely a Python one). Depending on how useful existing libraries are, we may be able to simply integrate an existing library, or we may have to do an entirely custom solution.

Tasks:

1. Look for XMI parsing libraries. The more modeling tools it supports the better. Ensure it captures behavioral or sequence diagram modeling behavior. Base the program language for this component on the library requirements. **Alternative:** If we cannot find a good XMI parser, fall back to an XML parser. Parse UML model components ourselves, this may be more complex.
2. Now that language is known, add onto the infrastructure to support it. Setup build, CI integration, start script integration, test options.
3. Break down the component with a low level design: A possible way to do this would be identifying specific parsing modules which each interpret specific XML tokens, or look for specific allowed UML model elements.
4. Split up and prioritize tasks based on design: We will increase priorities on modules which are dependencies of other modules (ideally minimizing those in our design), then on the modules corresponding to the most common UML model elements. Rare model elements may even be skipped on the first implementation if deemed to be low value.
5. After fulfilling all design requirements, show off a prototype which can accurately create a semantically identical model to the graphical model used.

UML to C^{2}KA Primitives

This component computes the “easier” initial steps of the C^{2}KA analysis, but it’s still a new computing problem. We will most likely implement components from here on entirely on our own. We believe there may be room for parallelization of the problem resolution so we would like to support concurrency in our component’s architecture.

We will try to avoid complex communications between entities, object-oriented programming does not fit our program as well. Therefore, the best fitting languages from here onwards are likely simpler languages which support procedural, functional, and concurrency programming. Some candidates we are considering are C, Go, and Rust.

The role of this component can be summarized as building useful data structures such that the next component can do its job efficiently.

Tasks:

1. Break down the component with a low level design: We define C2KA primitives as “**agents**, **stimuli**, and **behaviors**”. We may decide to create a module for each, with a high degree of independence. Later during iterative development, we may add some communication between them to verify the model through iterative estimations.
2. Split up and prioritize tasks based on design: As we get more familiar with C2KA, the three main modules defined above may be split further into steps. The earlier dependent steps take priority. The main modules themselves should be independent ideally, but we may find behaviors to be dependent on the other two (meaning it may be started last).
3. Settle on a language to use based on language benefits, the component design, and the feasibility of integration with the rest of the program.
4. Add onto infrastructure to support the chosen language and the new component. Build system, testing integration, and integration with previous components.
5. After fulfilling all design requirements, show off a prototype which can derive the intended C^{2}KA primitives from a model (the same ones a human could derive).

Formal Specifications Synthesizer

This component makes use of the C^2KA analysis we did so far to compute our intended output (Formal Specifications). It should share similar architecture patterns, in theory, but we know little of the synthesization process at this point in time.

For simplicity, we may decide to extend our C^2KA Primitives component even though they are logically separate components. If we do this we can mostly skip task 2, and 3.

Tasks:

1. Break down the component with a low level design. Here we're looking to pipe all the output from the primitives into three different specification modules. Modules to determine: *Abstract Behavior Specification, Stimulus-Response Specification, and Concrete Behavior Specification.* As we did with the C2KA Primitives, we will try to maximize their independence at first, and break them down further into steps once we have more knowledge of the process.
2. Split up and prioritize tasks based on design: Within the main modules mentioned, we will prioritize any dependencies first. We will prioritize Concrete Behavior, then Stimulus-Response specifications because they are higher risk implementations. We would rather be aware of the issues early than completing the easier parts of the component first for better time management.
3. Settle on a language to use based on language benefits, the component design, and the feasibility of integration with the rest of the program.
4. Add onto infrastructure to support the chosen language and the new component. Build system, testing integration, and integration with previous components.
5. After fulfilling all design requirements, show off a prototype which can derive the intended Formal Specifications from a model (the same ones a human could derive).

Specifications Serializer

A simple component which is concerned with making sure the IIAT can read our data as intended.

Tasks:

1. Analyze required input format for the IIAT.
2. Construct expected output file based on input specification for testing.
3. Build a serializer component in the tool with the target format in mind.
4. When the serialization from a test run matches the expected output file, we've proven the tool works with at least one model.
5. Do additional validation tasks: Acceptance testing with clients, testing other model profiles if we haven't already.
6. Assuming validation passes, ensure we baseline our first full working tool and transition to iterative development to improve it as outlined in objective 5. Section 5.3 gives detail into how the optional objective integrates into our development cycle.

5.3 Development Lifecycles

The optional objective mostly serves as a way to clearly limit our scope for a minimum viable product (which is our bare-minimum target for the poster fair). Many aspects mentioned to improve the tool are important, but not essential for a working prototype. To improve our tool as much as possible, and to minimize risks, we will adopt two stages to development.

The first stage will be inspired by the Rapid Application Development style. We will try to complete each component as fast as we can fitting our defined minimum requirements. We continue with this style until we've completed every component and get a full working prototype. We'd like to stick to this style because working on completely new software, and in a new problem domain, makes estimations difficult. Having a full working prototype will allow us to manage our time and risks better during the second stage.

The second stage would be more iterative development. We will identify areas of improvements through frequent prototype demos and focusing on those to create an improved prototype. We would repeat this process as much as we can depending on the time we have left.

5.4 Future Low-Level Architecture Variations

As we implement our low level design, we may split interfaces into several parts and the internal composition of components into parts as well. This may be to separate concerns further, to help support concurrency through the architecture, or to facilitate separating tasks internally (but this should be a secondary goal to avoid compromising the architecture for convenience).

Depending on our implementation strategies for C^2KA, it is quite possible for us to use an architecture like pipe + filter to efficiently process data. In fact, depending on how we do the parsing component, the same architecture and design philosophies could be useful there as well. However, due to our current limited knowledge on the subject, we want to avoid making lower level design decisions until later to make better decisions when they become relevant.

5.5 Target Deployment

Target Operating System

Some of the technologies used may be operating system (OS) dependent, so we will limit our scope to a single OS as a minimum. We may use bash, C, Rust which makes Windows less convenient. Windows also requires a license, we would prefer to avoid forcing our users to use it.

We will choose to prototype on Linux but may port to QNX if we have to make use of libraries with GPL3 licenses in Linux. Since they are both posix compliant, if we don’t use OS specific libraries, porting should not be too difficult.

The licensing is important because if there are industry uses, we may want to commercially release the product for companies, with the option of potentially having academic free licenses as well.

Tool Integration

Our tool serves as the glue between two different tools. For most convenience to the user, the three tools should be able to be used at once, in a **single location** (A single tool, single menu, or even a single button).

To automate the input integration, we can integrate our tool in the supported modeling tools directly through an extension which produces the IIAT file directly from the model.

To automate the output integration, we can try to provide a way to integrate our tool with the IIAT. This could be a simple script which detects our generated input file, and starts the IIAT tool. On its own, this might be redundant from a user's perspective.

If feasible, we could attempt to include the script into our extensions to have an option to automatically start the IIAT process instead of just generating the input file. Achieving our ideal "single location" solution for user convenience.

6 - Risks & Mitigations

Given the complexity of automating the translation of UML/SysML models into formal specifications, this project comes with several risks that could affect its progress and successful implementation.

6.1 Difficulty Extracting Accurate Information from UML/SysML Models

6.1.1 Description

UML/SysML tools vary in their support for exporting models in XMI/XML format, which is critical for the automated extraction of interaction data. Inaccurate or incomplete extraction could result in faulty formal specification.

This could lead to errors in the conversion process, making the generated formal specifications unreliable or incomplete, thereby undermining the tool's primary objective.

6.1.2 Mitigation Strategy

1. We will prioritize selecting UML/SysML tools with robust XMI/XML export functionality, such as Modelio, conducting early tests to validate the quality of exported data.
2. If a selected tool fails to produce acceptable XMI/XML output, we will research libraries to manually parse and correct XMI outputs.

6.2 Integration Challenges with Implicit Interactions Analysis Tool

6.2.1 Description

The software solution must generate formal specifications that can be fed into the Implicit Interactions Analysis Tool. A mismatch in formats or interfaces could prevent successful integration.

If integration fails, the project's objective of analyzing implicit interactions would be compromised.

6.2.2 Mitigation Strategy

1. We will research the input/output formats and interfaces of the Implicit Interactions Analysis Tool early in the project. This will include identifying any transformation steps required to convert our output into a format compatible with the tool.
2. Instead of waiting until the end to integrate, we will implement the integration incrementally—testing small sections of the project as they are developed.
3. Regular consultation with the professor and external resources (if needed) will ensure that we are correctly interpreting the tool's requirements.

6.3 Time Management and Balancing with Other Course Work

6.3.1 Description

This project requires a significant time investment, and balancing it with other courses, especially during midterms and final exams, may be challenging.

Lack of proper time management could lead to missed deadlines, rushed work, or incomplete project phases.

6.3.2 Mitigation Strategy

1. We will create a detailed project timetable **(as outlined in Section 6)** with milestones and clear deadlines to ensure that work is spread out over the term. Each team member will be assigned tasks based on availability and strengths, ensuring that we stay on track even during high-stress periods like exams.
2. We will build in buffer periods around midterms and finals where the workload is reduced, allowing us to focus on exams while keeping the project moving forward.

6.4 Overly Complex System Design and Performance Bottlenecks

6.4.1 Description

The scope of the project involves creating a complex software solution that integrates multiple tools and frameworks. The system architecture could become overly complex, making development, debugging, and maintenance difficult.

Our tool may encounter performance issues when processing large or complex UML/SysML models, particularly during the parsing and conversion stages. Performance bottlenecks, combined with complexity, could lead to excessive processing times and make the tool impractical for real-world use.

We will consider processing times over 1 minute for small models and 5 minutes for large models to be excessive. These thresholds are high, as real-time performance is not a requirement. Initial performance assessments will be qualitative, with a formal definition of model size once benchmarking is implemented.

6.4.2 Mitigation Strategy

1. We will design the software in a modular way, ensuring that each component (such as a UML parser, C^{2}KA converter, analysis tool integrator) can be developed, tested, and debugged independently. This will reduce the complexity of the system and make it easier to manage.
2. Every two weeks, we will evaluate our progress against the defined requirements to ensure we stay on track and haven't unintentionally deviated. If complexity increases unnecessarily, we will consider reducing the scope or simplifying the system architecture.
3. We will rely on informal manual testing to assess performance with small, medium, and large UML models. We will only investigate optimization strategies if processing times exceed our defined thresholds (1 minute for small models, 5 minutes for large models). This approach prevents premature optimization.
4. If performance issues arise, we will consider techniques such as parallel processing and lazy loading to improve the efficiency of processing large models. These strategies will be implemented incrementally as needed.
5. Regular code reviews will ensure that the codebase remains clean, well-documented, and manageable. Peer reviews will also help catch issues early before they escalate.
6. Scope management from our mitigation strategy in Section 5.6 will play a critical role in managing complexity, helping us detect deviations early and adjust accordingly.

6.5 Limited Experience with C^{2}KA and Formal Methods

6.5.1 Description

While the team has some background in software architecture and modeling, formal methods such as C^{2}KA may be new to some team members, leading to a steep learning curve.

This could slow down progress and introduce errors in the formal specification process if the team struggles to apply the C^{2}KA framework effectively.

6.5.2 Mitigation Strategy

1. We will dedicate the initial project weeks to understanding C^2KA and formal methods, leveraging online resources, textbooks, and academic papers. By frontloading this research, we can ensure that we have the required knowledge before diving into development.
2. The team will manually reproduce C^2KA analyses to better understand the associated processes. This will involve manually working through smaller UML models and performing C^2KA- bases analysis. Each team member will compare their approach, taking into consideration input from our professor, and together we will identify the most appropriate strategies to automate. By doing this, we can ensure that our understanding of the C^2KA framework translates smoothly into the computer-automated solution, reducing risk of misinterpretation during implementation.

6.6 Poor Requirement Management

6.6.1 Description

The project's requirements could evolve over time as the team gains insight or based on feedback from the professor. If requirements change unexpectedly, or the scope expands without clear communication, the project risks scope creep and misaligned objectives.

We will consider a change in requirements to be a significant risk if it leads to a delay of more than one major milestone or if it results in fundamental changes to core features that would require substantial redesign of the project.

6.6.2 Mitigation Strategy

1. We will conduct bi-weekly check-ins with the professor and team to ensure that any changes to the scope are carefully evaluated and documented. Changes will only be accepted if they fit within the project timeline and are achievable with the available resources.
2. When presenting new prototypes, we will evaluate whether we are adhering to the current requirements. During these presentations, we will confirm that the features align with the designed scope and ensure that no deviations have been introduced.
3. If we failed to catch a deviation from the original scope before introducing it, we will review the change to ensure it fits within the overall project goals. We may then choose to accept or revert these deviations.
4. We will prioritize the core objectives of the project (conversion of UML/SysML to C^2KA and integration with the Implicit Interactions Analysis Tool). Any additional features or extensions will be treated as secondary, only being developed if time permits.

7 - Project Timeline

7.1 Gantt Chart

Due to the nature of the objectives, components will mostly be developed sequentially. The following gantt charts show the expected amount of time spent on every objective and deliverable:

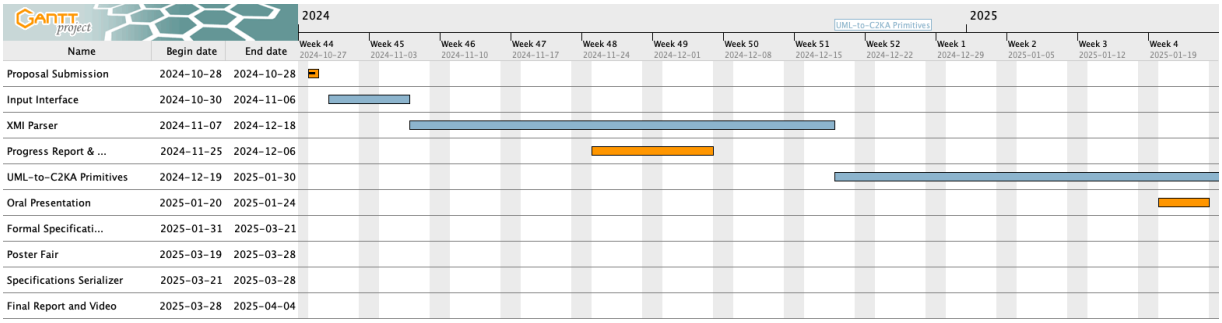

Figure 2 - Gantt Chart for Fall Term

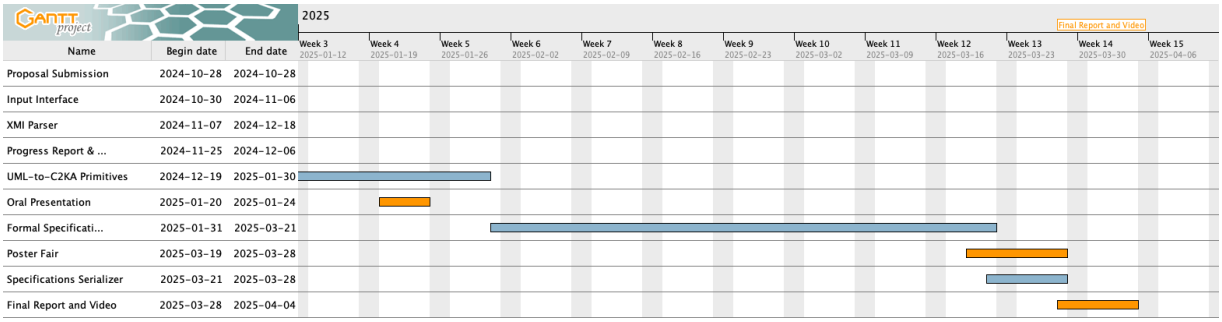

Figure 3 - Gantt Chart for Winter Term

8 - Degree Relation

Ahmed Babar:

I would say the very essence of the project is rooted in software engineering as the product is a tool for automating part of the software engineering process. All team members are in the software engineering program, and this project draws heavily from key concepts we've learned. Requirements engineering helps us separate functional and non-functional requirements, ensuring the project's behavior is well-defined. Architectural design patterns, like the Layer Pattern, describe component interaction. Courses like Software Verification and Validation taught us techniques such as unit testing, regression testing, and using mocks to manage dependencies during development. We'll apply these along with software lifecycle models (e.g., Waterfall) to ensure thorough development. Using the techniques learned in this degree, development of the software product will be smooth

Michael Rochefort:

As a Software Engineering student, this project directly applies the skills and knowledge I've developed throughout my degree program, especially in courses such as SYSC 3120: Software Requirements Engineering, SYSC 4120: Software Architecture and Design, and SYSC 3110: Software Design Project. These courses provided a strong foundation in defining, analyzing, and designing software systems to meet complex requirements. My role in developing the core software logic and ensuring accurate UML-to-C^2KA translations draws heavily on these courses, where I learned to model requirements, design architecture, and implement systematic design processes. In SYSC 3120, I gained experience with requirements analysis and learned how to capture functional and non-functional requirements, which will be instrumental in defining our project's goals. SYSC 4120 advanced my understanding of software architectures, focusing on modular and scalable designs, which I will apply to structure the system for seamless integration with the Implicit Interactions Analysis Tool. Additionally, SYSC 3110 taught me the practical aspects of executing a software design project from concept to implementation, preparing me to manage both the technical and collaborative aspects of this Capstone project. By building a software tool that integrates formal methods with practical engineering workflows, I am able to extend these technical skills into a real-world context directly relevant to my software engineering degree.

Alexandre Marques:

The way we are approaching the problem, we are planning to have a purely software based solution. There are no perceived dependencies on hardware, or other engineering disciplines as far as we can tell at this stage. There is some theoretical knowledge we have to learn outside the boundaries of our degree (Mainly the C2KA framework), but it builds upon some theoretical concepts of system design we learn through software engineering. As we are developing software, we will apply many concepts from our classes. Whether it'd be our classes on project management, architecture, and the experience we have accumulated on other software projects. The conversion work we are doing is directly related to a model familiar to us from our classes (UML) to the unfamiliar model of C2KA specifications. As such, we do have some problem domain specific knowledge, as well as general software skills to build the tool.

9 - Group Skills

Our team collectively possesses the skills necessary to undertake this project, drawing on a solid foundation of experience in both academic and professional software development contexts. Over our years at Carleton, we have become proficient in programming languages such as Python, Java, and C, and we have hands-on experience with essential tools, including Git and SysML/UML modeling software. Each member has completed courses in software development, software architecture, and project management, where we gained practical knowledge of methodologies like Agile.

In previous group projects, we successfully implemented applications such as a full Java version of UNO Flip and a Scrabble game, demonstrating our ability to manage complex software projects from design through deployment. These projects have sharpened our understanding of collaborative development, large-scale project management, and the end-to-end software process. With these skills and experiences, we are well-equipped to meet the technical and logistical demands of our Capstone project.

10 - Special Dependencies

Not applicable, we do not have any needs for special components and facilities to complete the project.

11 - References

[1] Jason Jaskolka and John Villasenor. An Approach for Identifying and Analyzing Implicit Interactions in Distributed Systems. *IEEE Transactions on Reliability*, 66(2):529-546, June 2017.

[2] J. Jaskolka, R. Khedri, and Q. Zhang. Foundations of communicating concurrent Kleene algebra. Technical Report CAS-13-07-RK, McMaster University, Hamilton, ON, Canada, November 2013.