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:

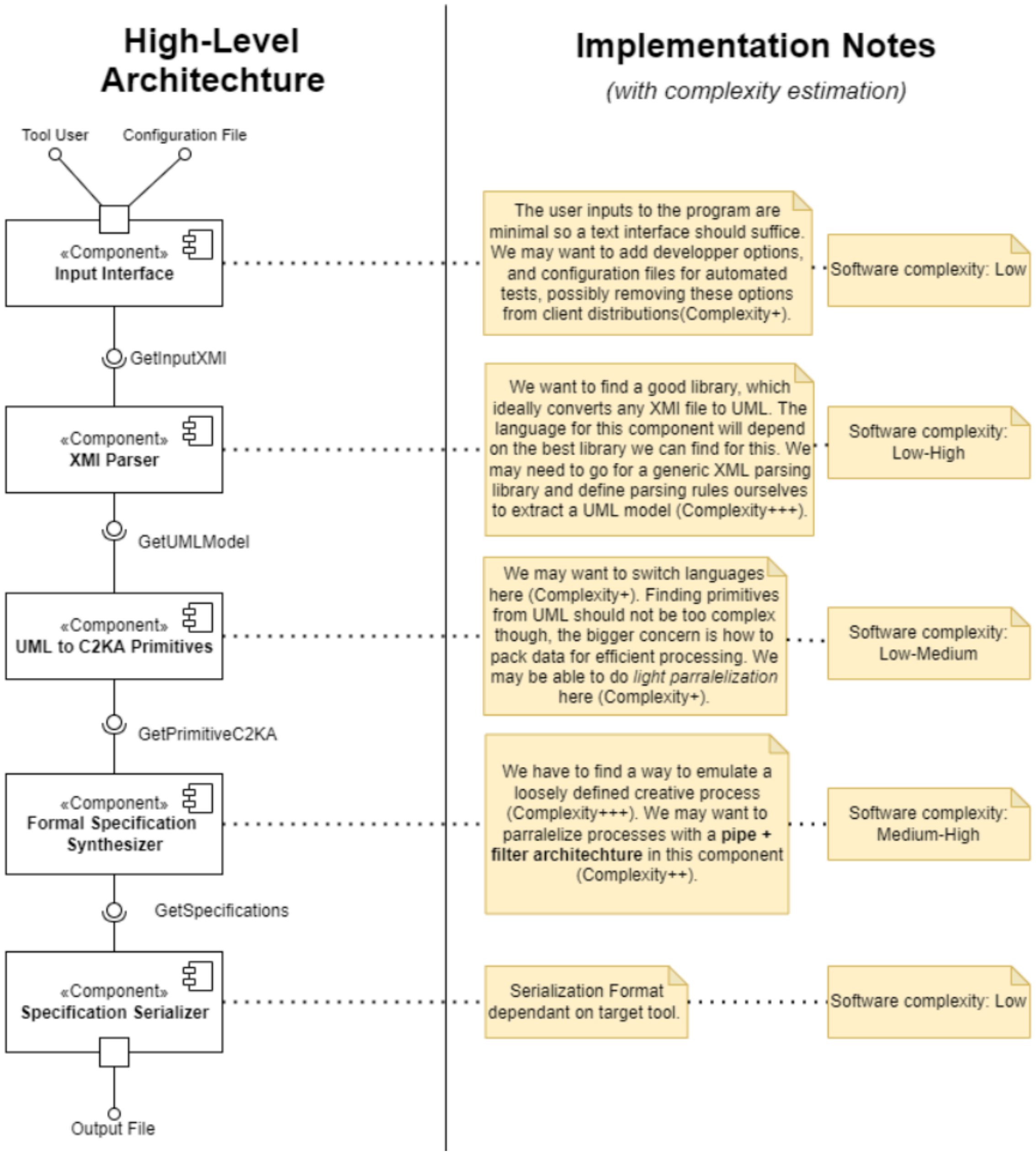
- Improving tool confidence (adding error detection, error correction, increasing accuracy with additional inputs).
- Improving tool convenience. User experience, deployment options, and tool interoperability are some examples in this category.

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.



High-Level Architechture
Implementation Notes
(with complexity estimation)
Tool User
Configuration File
«Component» Input Interface
GetInputXMI
«Component» XMI Parser
GetUMLModel
«Component» UML to C2KA Primitives
GetPrimitiveC2KA
«Component» Formal Specification Synthesizer
GetSpecifications
«Component» Specification Serializer
Output File
The user inputs to the program are minimal so a text interface should suffice. We may want to add developper options, and configuration files for automated tests, possibly removing these options from client distributions(Complexity+).
Software complexity: Low
We want to find a good library, which ideally converts any XMI file to UML. The language for this component will depend on the best library we can find for this. We may need to go for a generic XML parsing library and define parsing rules ourselves to extract a UML model (Complexity+++).
Software complexity: Low-High
We may want to switch languages here (Complexity+). Finding primitives from UML should not be too complex though, the bigger concern is how to pack data for efficient processing. We may be able to do light parralelization here (Complexity+).
Software complexity: Low-Medium
We have to find a way to emulate a loosely defined creative process (Complexity+++). We may want to parralelize processes with a pipe + filter architechture in this component (Complexity++).
Software complexity: Medium-High
Serialization Format dependant on target tool.
Software complexity: Low

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^2KA Primitives

This component computes the “easier” initial steps of the C^2KA 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^2KA 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.

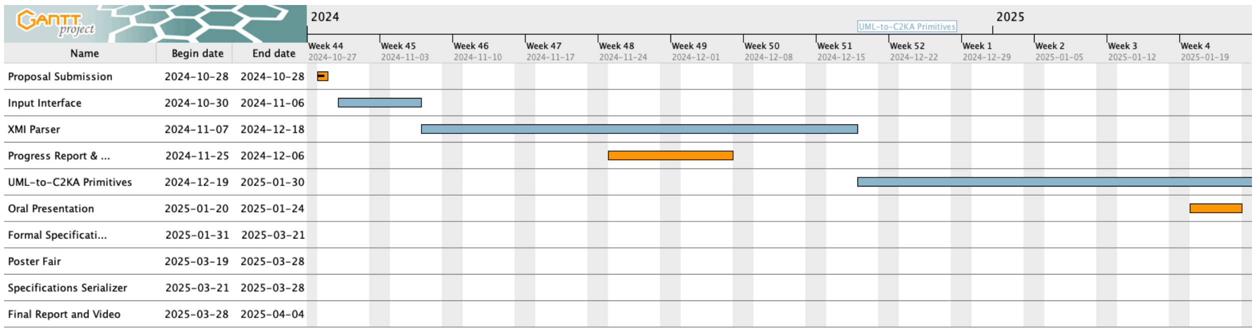
6.6.2 Mitigation Strategy

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

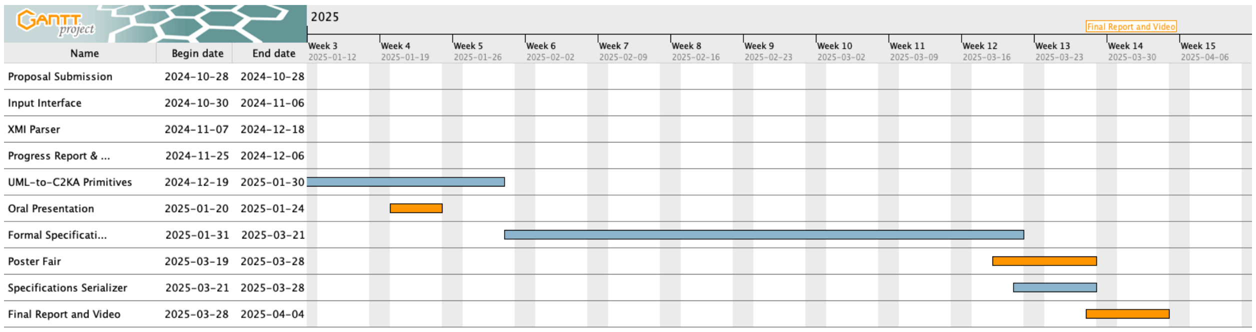
7.1 Gantt Chart

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:



Name	Begin date	End date
Proposal Submission	2024-10-28	2024-10-28
Input Interface	2024-10-30	2024-11-06
XMI Parser	2024-11-07	2024-12-18
Progress Report & ...	2024-11-25	2024-12-06
UML-to-C2KA Primitives	2024-12-19	2025-01-30
Oral Presentation	2025-01-20	2025-01-24
Formal Specificati...	2025-01-31	2025-03-21
Poster Fair	2025-03-19	2025-03-28
Specifications Serializer	2025-03-21	2025-03-28
Final Report and Video	2025-03-28	2025-04-04

Figure 2 - Gantt Chart for Fall Term



Name	Begin date	End date
Proposal Submission	2024-10-28	2024-10-28
Input Interface	2024-10-30	2024-11-06
XMI Parser	2024-11-07	2024-12-18
Progress Report & ...	2024-11-25	2024-12-06
UML-to-C2KA Primitives	2024-12-19	2025-01-30
Oral Presentation	2025-01-20	2025-01-24
Formal Specificati...	2025-01-31	2025-03-21
Poster Fair	2025-03-19	2025-03-28
Specifications Serializer	2025-03-21	2025-03-28
Final Report and Video	2025-03-28	2025-04-04

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.