



Renato Cavalcanti Domingues da Silva Filho

Verificação de propriedades de diagramas de atividade em um ambiente de modelagem aberto com suporte a rastreabilidade

Recife

2022

Renato Cavalcanti Domingues da Silva Filho

**Verificação de propriedades de diagramas de atividade em
um ambiente de modelagem aberto com suporte a
rastreadabilidade**

Monografia apresentada ao Curso de Bacharelado em Ciências da Computação da Universidade Federal Rural de Pernambuco, como requisito parcial para obtenção do título de Bacharel em Ciências da Computação.

Universidade Federal Rural de Pernambuco – UFRPE

Departamento de Computação

Curso de Bacharelado em Ciências da Computação

Supervisor: Lucas Albertins de Lima

Recife

2022

Dados Internacionais de Catalogação na Publicação
Universidade Federal Rural de Pernambuco
Sistema Integrado de Bibliotecas
Gerada automaticamente, mediante os dados fornecidos pelo(a) autor(a)

F481v Filho, Renato Cavalcanti Domingues da Silva
Verificação de propriedades de diagramas de atividade em um ambiente de modelagem aberto com suporte a rastreabilidade / Renato Cavalcanti Domingues da Silva Filho. - 2022.
50 f. : il.

Orientador: Lucas Albertins de Lima.
Inclui referências.

Trabalho de Conclusão de Curso (Graduação) - Universidade Federal Rural de Pernambuco,
Bacharelado em Ciência da Computação, Recife, 2022.

1. SysML. 2. CSP. 3. diagrama de atividade. 4. verificação. 5. ambiente de modelagem aberto. I. Lima, Lucas Albertins de, orient. II. Título

CDD 004



**MINISTÉRIO DA EDUCAÇÃO E DO DESPORTO
UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO (UFRPE)
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**

<http://www.bcc.ufrpe.br>

FICHA DE APROVAÇÃO DO TRABALHO DE CONCLUSÃO DE CURSO

Trabalho defendido por Renato Cavalcanti Domingues da Silva Filho às 09 horas do dia 27 de maio de 2022, no link <https://meet.google.com/rvb-secz-udk>, como requisito para conclusão do curso de Bacharelado em Ciência da Computação da Universidade Federal Rural de Pernambuco, intitulado “Verificação de propriedades de diagramas de atividade em um ambiente de modelagem aberto com suporte a rastreabilidade”, orientado por Lucas Albertins de Lima e aprovado pela seguinte banca examinadora:

Lucas Albertins de Lima
DC/UFRPE

Sidney de Carvalho Nogueira
DC/UFRPE

Dedico esse meu trabalho a todas as pessoas que me ajudaram durante toda a minha carreira acadêmica, principalmente no período da faculdade.

Acknowledgements

Primeiramente, agradeço à minha família por sempre ter cuidado de mim. Agradeço também aos meus amigos por sempre estarem ao meu lado nos momentos difíceis. Por último mas não menos importante, agradeço ao meu orientador o professor Lucas Albertins por ter me dado a oportunidade de ingressar no projeto de pesquisa que serviu como base para esse trabalho.

“Não importa o quão poderoso você se torne, nunca tente fazer tudo sozinho. Caso contrário irá falhar.”
(Itachi Uchiha)

Resumo

À medida que a tecnologia avança, os modelos e sistemas tornam-se cada vez mais complexos, assim como o esforço para verificá-los. À medida que um projeto avança, o custo de correção de erros aumenta exponencialmente. Assim, técnicas que auxiliem na identificação antecipada de tais erros são cada vez mais importantes. Dentre essas técnicas, a verificação de modelos tem se apresentado como uma abordagem interessante. No entanto, ela requer a manipulação de notações formais que são difíceis de operar por projetistas de sistemas. Portanto, a criação de ferramentas que abstraem os aspectos formais dessas abordagens de verificação tem se mostrado um caminho promissor. Outro aspecto relevante é que alguns problemas podem surgir devido à natureza concorrente desses sistemas. Problemas como *deadlock* e não determinismo estão bastante presentes nessa perspectiva. No entanto, a maioria das ferramentas atuais não tem a capacidade de lidar com esses problemas. Além disso, as que conseguem geralmente são ferramentas que exigem licenças pagas. Neste trabalho, expandimos a ferramenta criada em trabalhos anteriores para que ela possa ser utilizada em ambientes de modelagem abertos, sem deixar de ser de código aberto e não comercial. Nossa ferramenta agora tem a capacidade de verificar propriedades como *deadlock* e não determinismo de diagramas de atividades que são criados usando a linguagem SysML em um ambiente de modelagem aberto chamado OpenMBEE. Embora alguns outros trabalhos realizem a verificação de *deadlock*, poucos são aqueles que realizam a verificação de não-determinismo, menos ainda são aqueles que podem realizar ambos. Nossa ferramenta possui uma mecanização formal subjacente que nos permite realizar verificações automatizadas. Além disso, a ferramenta também traz a vantagem de que seus usuários não precisam entender ou manipular essa linguagem formal, pois fornecemos um módulo de rastreabilidade que rastreia os resultados da verificação formal de volta à notação do ambiente de modelagem, que é baseada no formato JSON. As principais contribuições deste trabalho são o aumento da expressividade da ferramenta e a adição de suporte para a verificação de diagramas de atividades de um ambiente de modelagem aberto. Avaliamos nossa abordagem usando um modelo real da indústria relacionado ao desenvolvimento de um Telescópio de Trinta Metros (TMT), que é fornecido pela comunidade OpenMBEE.

Palavras-chave: SysML, CSP, diagrama de atividade, verificação, ambiente de modelagem aberto.

Abstract

As technology advances, models and systems become increasingly complex, as does the effort to verify them. As a project progresses, the cost of correcting errors increases exponentially. Thus, techniques that help identify such errors in advance are increasingly important. Among these techniques, model checking has been presented as an interesting approach. Nevertheless, it requires manipulation of formal notations that are difficult to operate by system designers. Therefore, the creation of tools that abstract the formal aspects of these verification approaches has been shown as a promising way forward. Another relevant aspect is that some problems may arise due to the concurrent nature of these systems. Problems such as deadlock and non-determinism are quite present in this perspective. However, most of the current tools lack the capability to handle such problems. In addition, those that succeed are often tools that require paid licenses. In this work, we expand the tool created in previous works so that it can be used in open modeling environments, while still being open source and non-commercial. Our tool now has the ability to verify properties such as deadlock and non-determinism of activity diagrams that are created using the SysML language in an open modeling environment called OpenMBEE. Although some other works perform deadlock verification, few are those that perform non-determinism verification, even fewer are those that can perform both. Our tool has an underlying formal mechanization that allows us to perform automated checks. Furthermore, the tool also brings the advantage to its users that they do not need to understand or manipulate such a formal language, because we provide a traceability module that tracks the results of the formal verification back to the modeling environment notation, which is based on the JSON format. The main contributions of this work are the increase in the expressiveness of the tool and the addition of support for the verification of activity diagrams of an open modeling environment. We evaluate our approach using a real industry model related to the development of a Thirty-Meter Telescope (TMT), which is provided by the OpenMBEE community.

Keywords: SysML, CSP, activity diagram, verification, open modeling environment.

List of Figures

Figure 1 – Cost to fix a problem in each phase based on (HASKINS et al., 2004).	11
Figure 2 – Example of an SysML activity diagram	21
Figure 3 – Diagram invoked by the Calibrate Reference Beam Map node	23
Figure 4 – Framework architecture	24
Figure 5 – Framework process	26
Figure 6 – Shortened version of a returned JSON file	27
Figure 7 – OpenMBEE sub module architecture	28
Figure 8 – CSP specification structure	31
Figure 9 – Find and Identify diagram available on the MMS	32
Figure 10 – Specification of the SH_Image datatype	32
Figure 11 – Select Filter SH diagram available on the MMS	34
Figure 12 – Specification of the enumeration SHFilter	35
Figure 13 – Counterexample JSON	36
Figure 14 – Swagger interface	38
Figure 15 – Diagram Setup APS, Acquire and Start Guiding	40
Figure 16 – Get CCD Temperature diagram	42
Figure 17 – Visual representation of the path	42
Figure 18 – Modified version of the counterexample JSON	43
Figure 19 – Decision node with identical guards	44
Figure 20 – Modified JSON from non-determinism counterexample	44

List of Tables

- Table 1 – Action nodes representations 29
- Table 2 – Control nodes representations 30
- Table 3 – Edges representations 30
- Table 4 – Object nodes representations 31

List of abbreviations and acronyms

UML	Unified Modeling Language
SysML	System Modeling Language
CSP	Communicating Sequential Processes
MMS	Model Management System
MDKs	Model Development Kits
VE	View Editor
LTS	Labelled Transition System
JVM	Java Virtual Machine
JDK 11	Java SE Development Kit 11
APS	Alignment and Phasing System
PIT	Pupil Image Tracking loop
LGSPN	Labeled Generalized Stochastic Petri Net

Contents

	List of Figures	7
1	INTRODUCTION	11
1.1	Objectives	14
1.2	Methodology	15
1.3	Work organization	15
2	BACKGROUND	17
2.1	CSP	17
2.1.1	Sequential Processes	17
2.1.2	CSP language	18
2.2	Activity Diagram	21
3	ACTIVITY DIAGRAM CHECKER FOR OPENMBEE ENVIRON-	
	MENT	24
3.1	Framework architecture	24
3.2	OpenMBEE sub module	25
3.3	Translation into CSP semantics	29
3.3.1	Datatype	31
3.3.2	Enumeration	33
3.4	Verification and traceability	34
4	EVALUATION	37
4.1	Environment preparation	37
4.2	Case study	39
4.2.1	Structural verification	39
4.2.2	Deadlock and non-determinism verification	41
5	CONCLUSION	45
5.1	Related works	45
5.2	Limitations and future work	46
	BIBLIOGRAPHY	47

1 Introduction

As a project progresses, its models become increasingly detailed and complex, thus making it difficult to identify errors. This difficulty can cause this identification step to be postponed to later stages. However, the more advanced the project phase, the more expensive it becomes to solve a problem. In this context, (HASKINS et al., 2004) does a study on the cost escalation to fix errors over the course of a project. As can be seen in Figure 1, if the base cost in the requirements phase is 1x, in the design phase it could cost up to 8x more, 16x more in manufacturing phase, 78x in Integration and test phase and can reach up to absurd 1500x in Operation phase. Thus, techniques that can identify errors in advance become even more important. Among these techniques, model verification has been presented as an interesting approach, due to the fact that it is able to identify these errors even in the design phase.

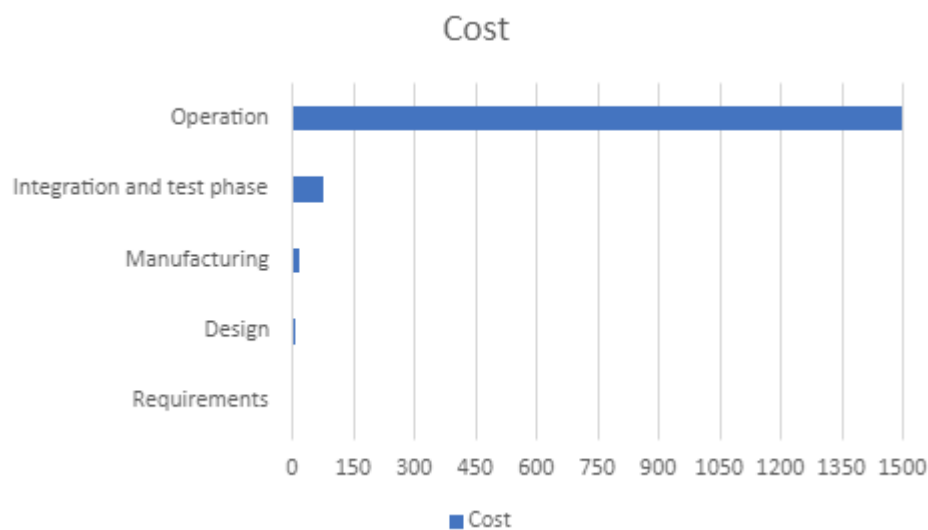


Figure 1 – Cost to fix a problem in each phase based on (HASKINS et al., 2004).

The semantics of these models can be formal or informal. The formal models are unambiguous and have several tools that allow you to perform automatic verification on them, but to take advantage of these features it is necessary to use the adequate mathematical concepts. The usage of formal models require a greater amount of related technical knowledge both to understand and to manipulate them. Meanwhile, informal models such as Unified Modeling Language (UML) (OMG, 2017) are easier and more intuitive to understand as they do not require those mathematical concepts. However, due to the lack of such concepts, the models can become ambiguous and, thus, present the need for a designer to judge whether or not such models have any kind of flaws. The designer relies mostly on his intuition and experience, which can lead to errors due

to the human factor.

Among the models that present concurrency, we can mention two major problems, deadlock and non-determinism. The deadlock problem occurs when a part of the system, or even the entire system, is waiting for a certain resource or signal that will never arrive, thus causing a system crash. A classic problem when we talk about deadlock is the dining philosophers problem, in which five philosophers numbered from 0 to 4 sit at a round table to eat their dinner. Each philosopher has his own place at the table and in front of him his plate of food. The problem is that each plate has a spaghetti that needs two forks to be eaten and each plate has a fork on its left and one on its right. If each philosopher picks up the fork to the right of his plate, all the forks will be busy but no one will be eating.

On the other hand, the problem of non-determinism is a little more subtle, it happens when you run the same system, with the same inputs n times and in the $n+1$ attempt it presents a completely different and/or unexpected result, causing a loss of control of the system features. In critical systems projects where a single mistake can lead to disastrous situations such as major financial losses or even loss of human life, the verification process is critical and, in general, becomes quite expensive for companies due to the lack of available tools that allow this verification.

In this work we seek an approach that combines the best of both worlds. The UML language is well known as a standard for creating systems design models, whether hardware or software. It can be used for visualization, specification, construction, documentation of artifacts and is mainly intended for complex systems such as banking and financial services, telecommunications, transport and distributed services. From the UML language, other languages were developed, such as the System Modeling Language (SysML). This language can be considered as an extension of UML, however, while UML is more software oriented, SysML is intended to systems engineering. As a less software-focused language, SysML diagrams can be more expressive and flexible. Currently, the UML language has fourteen types of diagrams that can be separated into two large groups: structural diagrams and behavioral diagrams, each of these contains seven types of diagrams. SysML covers 7 of these diagrams along with the addition of 2 more, requirement and parametric diagrams. Here, we use one type of diagram that is covered from both, the activity diagram. According to (REGGIO et al., 2013), the activity diagram is one of the most used diagrams in the industry, it is usually used to represent functionalities and flows of systems, including concurrent systems.

As time goes on, projects are becoming broader and more complex, however, such scope hampers the level of validations that we can perform in SysML models due to their informal semantics, which are textually described in natural language. The Communicating Sequential Processes(CSP) language (HOARE, 1985) provides

a process algebra for specifications of concurrent systems. This language has been practically applied in industry as a tool for specifying and verifying the concurrent aspects of a variety of different systems, such as the T9000 Transputer, as well as a secure e-commerce system. It has formal (unambiguous) semantics and refinement calculation mechanisms, which are automated through the FDR4 tool ([GIBSON-ROBINSON et al., 2014](#)). In addition to these refinements, the FDR4 also provides mechanisms for checking properties such as deadlock and non-determinism. A CSP specifications can be defined in text files and with the use of its process algebra, therefore, it becomes more complex due to more rigorous mathematical notations and consequently making it more difficult and less intuitive than diagrammatic level specifications.

By describing the semantics of the SysML language using a language like CSP we are defining a formal semantics for SysML. In this way, in addition to being able to define unambiguous models, we can use the tooling support of a formal language to perform automated logical reasoning in the models.

Several previous works aim to automatically verify properties of activity diagrams, helping designers to build more reliable systems. For instance, the following works: ([LIMA; DIDIER; CORNÉLIO, 2013](#); [LIMA, 2016](#); [LIMA; TAVARES; NOGUEIRA, 2020](#)) were used, to define formal semantics for UML diagrams in terms of the CSP language. Such works served as a basis for the creation of a strategy for translating activity diagrams to CSP specifications. Using the FDR tool it is possible to verify properties of such specifications. Most famous modeling environments like Simulink ([MATHWORKS, 2022](#)) and SCADE ([ANSYS, 2022](#)) are commercial tools that require a paid license, which makes it difficult for some designers to access verification features. In addition, they are tool-dependent modeling environments. Then, in ([LIMA; TAVARES; NOGUEIRA, 2020](#)) a plug-in to the Astah UML modeling environment ([VISION, 2019](#)) has been proposed to check deadlock freedom and non-determinism on activity diagrams. This plug-in automatically translates an activity diagram to a CSP specification and uses FDR in background to verify these properties. Traceability is also provided at the diagrammatic level. Thus, at no time during the process do system designers need to have any knowledge of the formal semantics.

The approach proved to be efficient in terms of verification, however, it was also tool dependent. So, in ([DOMINGUES, 2021](#)) the architecture of the framework was refactored to allow the creation of an API and thus allow the inclusion of new modeling tools, consequently mitigating the framework's tool dependency. In this follow-up work we decided that the Astah framework support would still be maintained and new tools could be supported. Nevertheless, supporting tools creates a dependency between the approach and how these tools represent the UML/SysML models. Therefore, instead of supporting tools, the ideal scenario would be to support an open modeling

environment that could be the baseline for several tools. That is the idea behind the OpenMBEE ([OPENMBEE, 2021](#)). Which is a community supported by companies and institutes aiming to provide an open source modeling environment for model-based system engineering. That has already made available some tools that help in the creation of such an environment, like:

- Model Management System (MMS) which is a central repository that provides services for managing models and is a version control system for real industry projects that can be consulted through REST requests.
- Model Development Kits (MDKs), these kits are tool-specific integrations that its primary purposes are to sync models with the MMS, and implement the DocGen language, which allows modelers to dynamically generate documents in a model-based approach using the view and viewpoint concept.
- View Editor (VE) that enables users to interact with SysML models within a web-based environment. It implements the MMS REST API to provide a web environment to create, read, and update model elements, including Documents and Views.

By creating an adapter to the MMS for the framework API it was possible to transform the framework into an open source environment.

1.1 Objectives

For this work we aim to continue the work started in ([DOMINGUES, 2021](#)), in addition to increasing the expressiveness of the framework through the addition of new CSP semantics for composite datatypes, modifications to the parser so that it is possible to support such new semantics, along with with the creation of an adapter package that allows the communication of our API with that of the MMS and, finally, create a module to generate a counterexample to allow designers to remain at the diagrammatic level.

General objective:

Expand the open source environment that performs the verification of properties of UML/SysML activity diagrams automatically using formal reasoning technologies to allow traceability of results to the diagrammatic level.

Specific objectives:

1. Define semantics in CSP for structured types not currently supported by the approach, more specifically, enumerations and datatypes of SysML.
2. Implement the semantics for new structured types previously defined in order to increase the expressiveness of the framework.
3. Study possible mechanisms for generating counterexamples in order to allow the traceability of property verification in activity diagrams of the OpenMBEE environment.
4. Develop a counterexample generation module to allow traceability of property checks in activity diagrams of the environment provided by the OpenMBEE group (MMS).
5. Evaluate the activity diagram checker built through a real industry case study related to the OpenMBEE project.

1.2 Methodology

To achieve these specific objectives, we follow this methodology:

1. Study the technologies: In this step, we study the technical aspects related to our work: the OpenMBEE architecture and related technologies, the current state of the framework developed in previous works, and, finally, the CSP related tools.
2. Develop communication module: This step involves the development of the module responsible for communication with the OpenMBEE REST API.
3. Develop translation of composite types: This step provides, the necessary translation for the new semantics of complex data types.
4. Develop traceability module: This step focuses on the construction of the module responsible for receiving the counterexample trace from FDR4 and generating the counterexample diagram in the notation of OpenMBEE.
5. Evaluate results: In this step, we evaluate our proposal by performing a real case study from the industry, which is available in the OpenMBEE repository.

1.3 Work organization

This work is divided into 5 chapters. The remainder of this work is organized in the following chapters:

- Chapter 2 describes the background of our approach, showing in more detail the material that was used as a basis for the research.
- Chapter 3 presents our framework to verify properties of activity diagrams in the context of an open modeling environment. It details how the activity diagram checker works for the OpenMBEE environment.
- Chapter 4 evaluates the results found by our framework in our case study.
- Finally, Chapter 5 brings the conclusions and discuss future directions.

2 Background

In this chapter, the theoretical topics related to our work are presented. In Section 2.1 we present what the CSP language is, how a CSP specification works, how our work uses them and finally we discuss about the FDR tool. In Section 2.2, we detail SysML activity diagrams.

2.1 CSP

This section is divided in two parts, Section 2.1.1 makes a brief explanation about sequential processes, while Section 2.1.2 describes the semantics of the CSP language.

2.1.1 Sequential Processes

Before we talk about more complex issues such as the CSP language or models, it is interesting to present a more basic and equally important concept, which is that of sequential processes. To better understand, first let's abstract the computers and focus on the things around us and how they interact with each other. Let's imagine an arcade and some of its machines, a simpler machine like a slot machine, how could we describe how it works?

- Insert the coin
- Pull the lever
- Take the reward

Now for a more complex machine like an arcade machine where you can play different games like Street Fighter or King of Fighter, it would have a greater variety of events like:

- Insert the coin
- Choose the game
- Select game mode
- Pick your fighters
- Pass the stages
- Take your reward

Each of these events can be organized in different orders that represent the possible scenarios of execution of the system. When we perform permutations one by one in the order of these possible scenarios, it is possible to perceive the expansion capacity of this model and, consequently, the difficulty in analyzing them. To make it even more difficult, models like the one described above that are written in natural language allow multiple interpretations. For example, in the event *Pass the stages*, the conditions for this to happen were not described, so designer 1 might think the condition would be x while designer 2 would say y . This interpretive duality tends to increase especially when more complex elements such as decisions, parallelism, and aligned invocations are added to models. When we add these complex elements together with the increase in the size of the models due to permutations of the order of events, it is necessary to use the tooling power to solve this issue. In this work we use the formal language CSP to allow the use of such tools.

2.1.2 CSP language

According to (SCHNEIDER, 1999), the CSP language was designed for describing systems of interacting components, and it is supported by an underlying theory for reasoning on them. A CSP process is the basic unit of behavior description. It is defined in terms of events or other processes. Events are linked to channels, which may or may not communicate some type of data. The mathematical form of CSP cannot be read by a computer and for that, another version of CSP was developed, (ROSCOE, 1998) named it CSP_M , a machine-readable version of CSP that can be understood by computers.

Let's use the arcade machine example from Section 2.1.1 to demonstrate how a CSP_M process is written. Each of those events can represent a functionality of that machine, and to facilitate this representation, we can name each of them. We could name each of these events in a way that takes up less space but preserves the meaning, for example:

- *inCoin*
- *selGame*
- *selGMode*
- *selFighters*
- *actPlay*
- *outRwd*

In this example the prefix *in* would represent an input, *sel* would be for a selection, *act* for an action and *out* for an output. Furthermore, we can call the set of all these events as the process alphabet, in the mathematical CSP it is symbolized by the Greek letter α (Alpha), and for practical purposes we will use the acronym AM to abbreviate arcade machine.

$$\text{Alpha (AM)} = \{ \text{inCoin}, \text{selGame}, \text{SelGMode}, \text{Selfighters}, \text{actPlay}, \text{outRwd} \}$$

The process name starts with capital letters. In CSP language, an event can be followed by a process or an event. To be followed by an event, we use the model $a \rightarrow b$, where a is an event and b can be the same or another event, as seen in the example: *inCoin* \rightarrow *selGame*, however, one event followed by another does not constitute a valid CSP process. For an event to be followed by a process, we use the following model $a \rightarrow P$, where P is any process, *outRwd* \rightarrow *SKIP* is an example of this. To show that the process has ended successfully, the *SKIP* process is added to the end. So if we want to describe the arcade machine process it would look like this:

$$\text{AM} = \text{inCoin} \rightarrow \text{selGame} \rightarrow \text{selGMode} \rightarrow \text{Selfighters} \rightarrow \text{actPlay} \rightarrow \text{outRwd} \rightarrow \text{SKIP}$$

A process does not need to represent the entire system and, therefore, a process can invoke another, including itself, during its execution or termination. Another characteristic of processes is that the execution order does not always have to be a single, fixed path. For this there are two choice operators, which are represented by the symbols $[]$ and $|\sim|$. The first one represents external choice, where two options are offered. The second one is for internal choice where the system internally decides which path to take. Although an inner-choice operator always leads to non-determinism, other operators such as outer-choice can also cause it. Then, if we modify the arcade machine, process to include these properties we can specify the following process:

$$\text{AM} = \text{inCoin} \rightarrow \text{selGame} \rightarrow \text{selGMode} \rightarrow \text{Selfighters} \rightarrow \text{actPlay} \rightarrow (\text{outRwd} \rightarrow \text{SKIP} [] \text{outRwd} \rightarrow \text{AM})$$

With this modification, at the end of the execution, it is possible to choose between ending the process or starting it over so that the user can play again.

Events can communicate data through three operators: $'.'$, $'?'$ and $'!'$. For example, the *inCoin* can generate an input data using the $'?'$ operator: *inCoin?* x . The *outRwd* can communicate an output data with the $'!'$ operator: *outRwd!* y . The $'.'$ operator is used to explicitly communicate a value. However, to be able to carry out these communications it is first necessary that these events are declared as channels. To do this we specify the listing below:

```
channel inCoin : int
```

```
channel outRwd: int
```

When turning these events into channels that communicate integers (type `int`), we also need to modify our *AM* process so that it is correct again:

```
AM = inCoin?x -> selGame -> selGMode -> SelFighters -> actPlay ->
(outRwd!1 -> SKIP [] outRwd!1 -> AM)
```

As the specification is getting more complex, we can add more details on the system like, what if the price to play the game is 2 coins but the user only put 1? We can make an adjustment to cover it by adding guards on some events.

```
AM = inCoin?x -> ( x >= 2 & (selGame -> selGMode ->
selFighters -> actPlay -> (outRwd -> SKIP | outRwd -> AM)))
```

In this specification, if the guard condition is not accepted, it will behaves as the process STOP. The process STOP is used to represent an unsuccessful end for the process, this is a deadlocked process.

Regarding concurrency, CSP has some operators, among them we have the Interface parallel. In this operator, two processes are put in parallel and synchronized by an alphabet of events, this operator can be represented by $A \llbracket \text{alphabet} \rrbracket B$. These two processes act independently until one of the events present in the alphabet is encountered, at that point the process can only continue its execution when the other is also ready to perform that same shared event.

CSP_m has many more operators that are not covered on this section like: *hiding*, *interruption* and *interleave*. You can read more about it on ([ROSCOE, 2010](#)).

As previously mentioned, in this work we use the FDR tool to verify properties such as deadlock and non-determinism. The FDR tool takes as input a specification in CSP_m and translates it into a Labelled Transition System (LTS) ([ROSCOE, 1998](#)), which is a directed graph where the transitions between states represent events of the CSP specification. It has notation similar to a state machine. This state machine is then exhaustively checked for the desired property. When it does not hold, the tool generates a counterexample trace that is returned to the user. Due to the need for an exhaustive search, some optimization functions are also offered by FDR, such as the `normal()` function. This function performs a normalization on the LTS, which can reduce the state space, thus, reducing the analysis complexity. This normalization preserves the original semantics of the specification.

2.2 Activity Diagram

The activity diagram makes a lot of features available in a simple and intuitive way for designers, that is one of the reasons why it is one of the most used diagram types on industry right now (REGGIO et al., 2013). These diagrams can be represented by directed edges that control both the execution flow and the data flow between nodes through the use of control and object tokens. There are three categories of nodes: Action node, Control node and Object node.

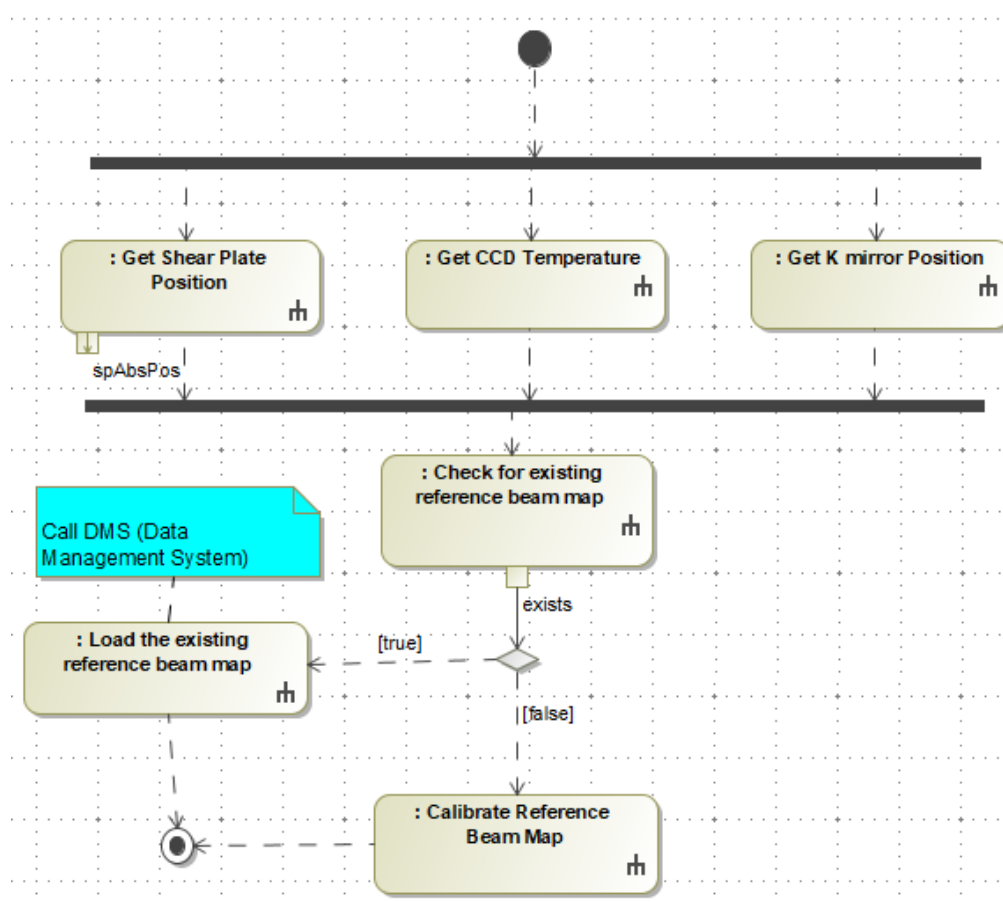


Figure 2 – Example of an SysML activity diagram
Source: Diagram Retrieve Reference Beam Map available on the MMS

Action nodes can be of several types, some of the most used are: *simple action node* that executes some behavior at that point, *send signal node* that creates a signal instance and transmits it to the target object, *accept event node* that waits for a signal reception to proceed, and *call behavior action node*, which represents the invocation of another activity. In Figure 2, there are six call behavior actions, *Get Shear Plate Position*, *Get CCD Temperature*, *Get K mirror Position*, *Check for existing reference beam map*, *Calibrate Reference Beam Map* and *Load the existing reference beam map* nodes.

There are seven types of control nodes. *Decision node* selects through guards which route will be taken. *Merge node* that brings together multiple alternate flows, it

is not used to synchronize concurrent flows but to relay any token received by one of its incoming edges to its unique outgoing edge. *Fork Node* splits a flow into multiple concurrent flows and *join node* synchronizes multiple incoming concurrent flows into one outgoing flow. *Initial node* offers a single control token when the activity starts, while a *final flow* consumes a token received in its incoming flow. The *final node* terminates the execution of its owning activity once it is reached. Some of these nodes are represented in Figure 2, the *fork node* is the horizontal black bar on the top, while the *join node* is the one on the bottom. Both, the *decision* and the *merge node* are represent as the white diamond, the *initial node* is the black dot on the top and the *final node* is the white circle with a black dot inside on the bottom left of the figure.

Object nodes are used to represent data inside the diagram. Some of the most used object nodes are: *activity parameter node* that is used for accepting values from the input parameters or providing values to the output parameters of an activity and *pin nodes* that provides input values to an action or accepts output values from an action. Once again in Figure 2, we have two output pins that can be seen on the *Get shear Plate Position node* and *Check for existing reference beam map node*.

Regarding edges, we have control edges and object edges. The former are traversed by control tokens, these tokens are used to control the order of execution of nodes. Meanwhile, object edges are traversed by object tokens that may hold values. In Figure 2, Control edges are represented by dashed lines, while object edges are represented by full lines. The representations are being based on the SysML model, the language used to create the *Retrieve Reference Beam Map* diagram from the Figure 2.

As already mentioned in Section 2.1.1, when we add complex elements to the model, its complexity for the analysis grows exorbitantly. Like the parallelization of three diagram invocations, as in the case of *Get Shear Plate Position*, *Get CCD Temperature*, *Get K mirror Position* nodes, followed by a decision of what other diagram will be invoked. To demonstrate how much a single invocation of another diagram can grow, Figure 3 represents the diagram invoked by the *Calibrate Reference Beam Map* node of Figure 2. This invoked diagram contains eleven action nodes and seven of them also invoke other diagrams.

Analyzing such a diagram without using an appropriate tool can take days or even weeks due to the need to verify each possible scenario. When we grow even further in the scale of models, analysis without tools becomes unfeasible both in terms of time and effort, in addition to increasing the chance of failures going unnoticed. Despite this, nowadays there are not so many approaches that deal with this kind of problem, so in this work we use the translation approach to a formal language that has the tooling support to try to solve it.

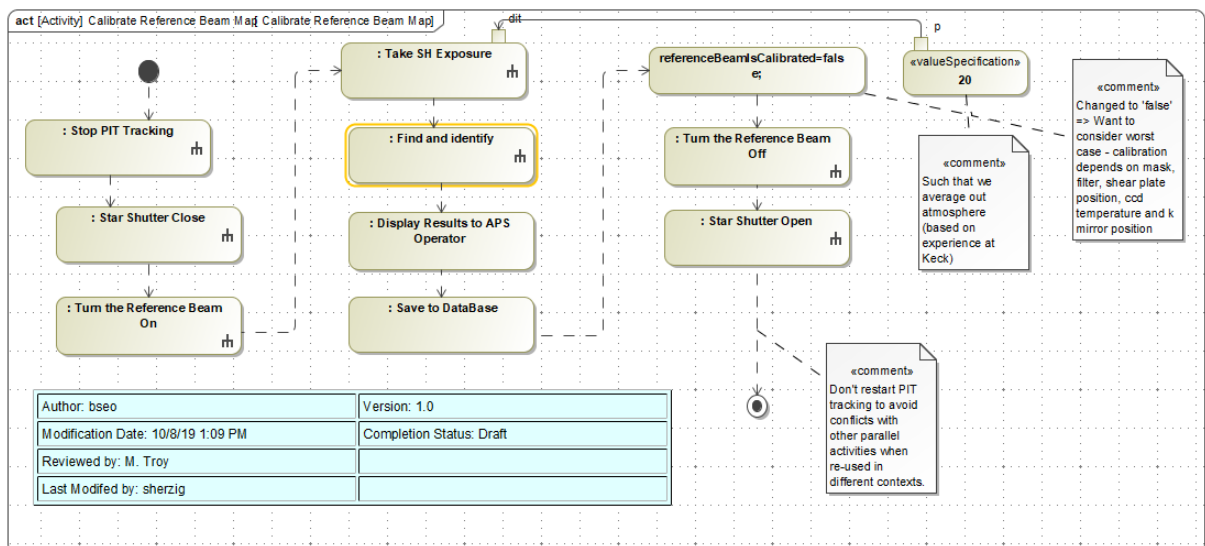


Figure 3 – Diagram invoked by the Calibrate Reference Beam Map node
 Source: Diagram Calibrate Reference Beam Map available on the MMS

3 Activity Diagram Checker for OpenMBEE environment

In this section, the entire property checker process is presented. In Section 3.1 we discuss the architecture of the framework, Section 3.2 details the OpenMBEE adapter sub module, Section 3.3 we show the new CSP semantics and finally, Section 3.4 explains the verification and traceability process.

3.1 Framework architecture

In this work we used the new architecture created in (DOMINGUES, 2021) and added a new package called OpenMBEE in the adapters part. Thus, the architecture used in this work can be seen in Figure 4.

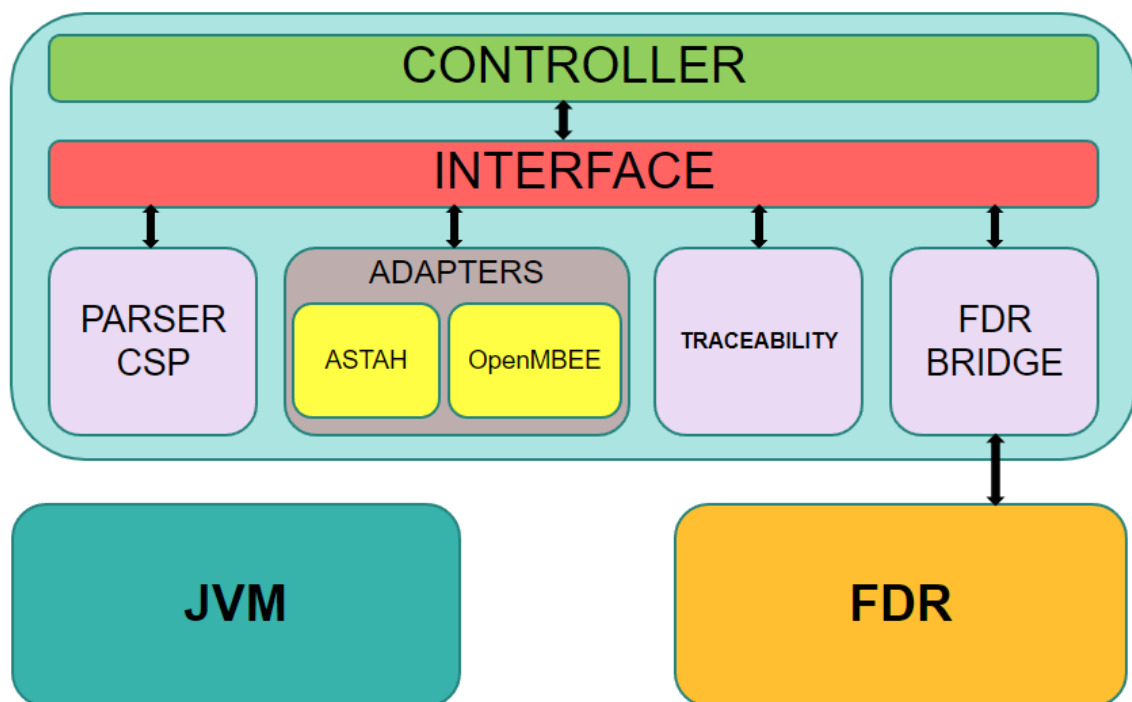


Figure 4 – Framework architecture

In the upper part of Figure 4, we have the internal part of the framework architecture, represented by the light blue box, while in the lower part we have the dependencies. The Java Virtual Machine (JVM) is a virtual machine that allows a computer to run programs in the Java programming language (GOSLING et al., 2000), the language in which the framework has been developed. The FDR model checker tool is necessary

so that it is possible to verify deadlock and non-determinism properties in the CSP specification derived from the activities. In the internal part of the framework code, we have the presence of six modules: controller, interface, parser, adapters, traceability and FDR bridge.

The controller module is responsible for receiving client requests, performing internal communication between the framework modules and returning the counterexample when it exists. The interface module, on the other hand, contains an API with interfaces that describe functions and elements of how we represent an activity. When implemented by the adapter module packages, these objects can be passed to the parser module to generate CSP specifications. The Parser CSP module is a tool-independent module, which receives the encapsulated object that represents an activity, according to the Interface module, and generates a CSP code equivalent to that activity diagram, following the defined CSP semantics, which are discussed in Section 3.3. The FDR bridge module is responsible for carrying out all the communication with the FDR tool. Through the use of dynamic programming technique, Java Reflection allows us to load the FDR library and use its attributes and methods at runtime. From the event trace that the FDR returns to the FDR bridge module, the traceability module creates the counter example, when necessary, and forward it to the controller module. Finally, the adapter module performs all the communication between the modeling environment and our framework, and also encapsulates the activity data according to definitions in the Interface module. Once implemented, the data is forwarded to the controller module that sends it to the parser module to generate the equivalent CSP code. This module has two adapter packages, the first for the Astah modeling tool, more details about its latest version can be found in (DOMINGUES, 2021). The second that was created in this work is for the OpenMBEE environment, which will be further detailed in the next section.

3.2 OpenMBEE sub module

As described in (OPENMBEE, 2021), "The Model Management System provides services for managing models and is a version control system for structured data. It exposes model information through RESTful web services that can be used for CRUD operations, branching, and tagging of the model repository". This exposure of information is performed through the use of files in the JSON format, which can be consulted through a swagger interface, that will be better described on Chapter 4.

To give a better understanding of the framework verification process, Figure 5 illustrates it in a graphical and simplified way. In it we can see that the client makes the request for our framework, which is represented by the square with rounded edges of light

blue color in the center of the figure. From there, all the necessary verification requests are made to the MMS API. This API returns a series of JSON files corresponding to the diagram elements to be verified. Figure 6 shows a shortened version of one of these returned JSON files. In it we can extract relevant information such as the type of that item being represented, its name, the ids of the owned elements. Once all the JSONs are collected, the framework performs all the necessary internal operations to generate the equivalent CSP_m specification. This specification is passed to the FDR tool, which then performs the verification of the desired property. If a deadlock or non-determinism is found, a trace of CSP events is returned to the framework. Finally, the framework receives this trace, converts it into a JSON file containing the id of all the elements of that diagram that generated the path to the property found and returns it to the client. To delve in more detail, let's talk now about the OpenMBEE sub module.

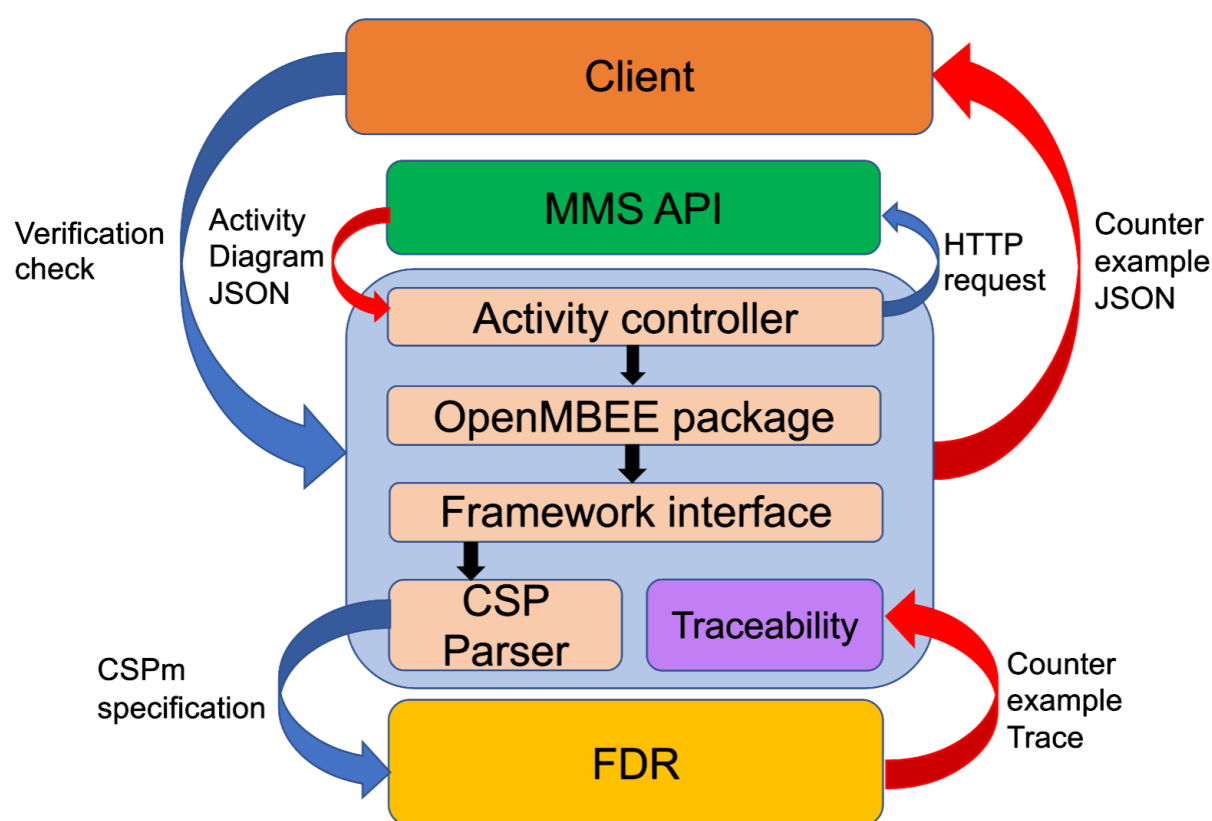


Figure 5 – Framework process

The OpenMBEE sub module is the main contribution of this work. In Figure 7 we can see how its architecture is structured. The communication layer is responsible for carrying out all requests to the REST API of the OpenMBEE MMS environment, in addition to transmitting the data received to the other packages, joining the structures created by them so that they can be passed to the controller module. In order to make requests to the MMS, it is necessary to pass the login and password to the package to access the environment, along with the id of the diagram to which the property

```
"elements": [
{
  "type": "Diagram",
  "ownerId": "_17_0_2_3_41e01aa_1382473227299_833303_50967",
  "_inRefIds": [
    "master"
  ],
  "_artifactIds": [
    "_17_0_2_3_41e01aa_1382473227313_422459_50968_svg",
    "_17_0_2_3_41e01aa_1382473227313_422459_50968_png",
    "_17_0_2_3_41e01aa_1382473227313_422459_50968_png"
  ],
  "id": "_17_0_2_3_41e01aa_1382473227313_422459_50968",
  "_refId": "master",
  "_modified": "2022-04-07T21:02:07.171+0000",
  "supplierDependencyIds": [
    "_18_0_2_baa02e2_1423591778981_842434_136008",
    "_18_0_2_b4c02e1_1436279234541_575502_145508"
  ],
  "_appliedStereotypeIds": [
    "_17_0_2_3_b4c02e1_1376582877372_304977_35082",
    "_9_0_be00301_1108044380615_150487_0",
    "_17_0_1_232f03dc_1325612611695_581988_21583",
    "_12_1_8740266_1173775032859_804702_281"
  ],
  "visibility": "public",
  "_docId": "7074dc23-0e8d-454b-a1ef-359f67d14f53",
  "_commitId": "5b6c606f-fbfa-427c-89df-b0a9c8ab9e90",
  "_diagramType": "SysML Activity Diagram",
  "_creator": "test",
  "_created": "2022-04-07T21:02:07.171+0000",
  "name": "Retrieve Reference Beam Map",
  "_projectId": "PROJECT-d94630c2-576c-4edd-a8cd-ae3ecd25d16c"
}
]
```

Figure 6 – Shortened version of a returned JSON file

verification is going to be checked and which type of verification is desired, deadlock or non-determinism.

The utility class contains a series of public and static functions that are used during communication and creation of objects that implement the interface module API. In this class it is possible to find functions for:

- Adaptation of JSON data types received from MMS
- Node and edge nomenclature checker and corrector
- Primitive type generator
- Definition of data types used in the diagram

The ActivityElementBuilder interface is implemented by all builders in this package. It is used to create a basic unit of an activity diagram object. The builder package consists of seven classes that build a specific type of element from an activity diagram. They are:

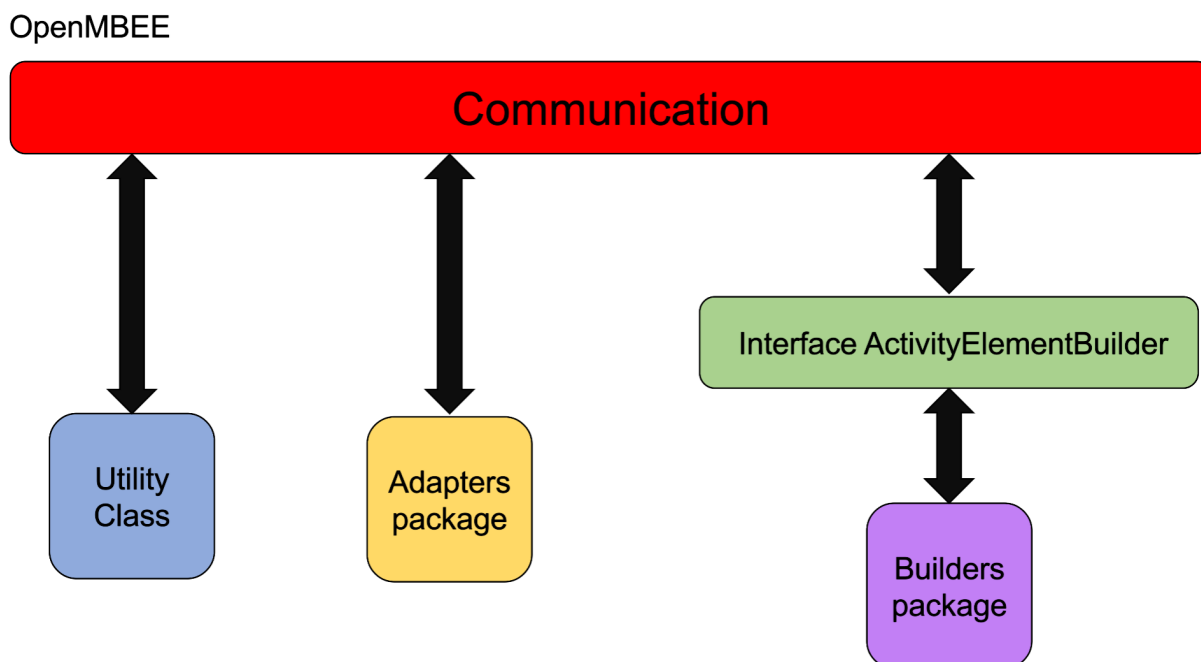


Figure 7 – OpenMBEE sub module architecture

- ActionBuilder creates action node types, and currently, we support the types shown on Table 1.
- ControlBuilder creates control type nodes, it also supports seven node types as shown on Table 2.
- DiagramBuilder generates the structure of an activity diagram.
- FlowBuilder is responsible for creating the two types of edges shown on Table 3.
- ObjectBuilder creates object type nodes, the three types are shown on Table 4.
- PartitionBuilder creates an activity diagram partition.
- TypeBuilder generates the data types used by the object nodes.

Finally, the adapter package is responsible for the existing classes in the OpenMBEE sub module. It contains all the types already mentioned in the constructor package in addition to the abstract types they inherit from such as:

- ActionNode: all classes generated by the ActionBuilder package inherit from this class
- ControlNode: all classes generated by the ControlBuilder package inherit from this class
- ObjectNode: all classes generated by the ObjectBuilder package inherit from this class


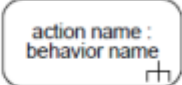




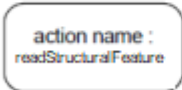
Accept event action 	Call Behavior action 
Send signal action 	Opaque action 
Value specification action 	Read self action 
Read structural feature action 	

Table 1 – Action nodes representations

- Activity: while a diagram represents the view of an activity, this represents the activity attached to it. It is mainly used for invocations from the Call Behavior Action node.
- ActivityNode: represents the most basic unit of an activity node

3.3 Translation into CSP semantics

In this section, we will discuss how to translate an activity diagram taken from MMS to a specification in CSP_m. In Figure 8, we can see in a generic way how a CSP specification is structured.

The external part named *MainProcess* represents the CSP process that encapsulates the activity. It composes in a synchronized parallelism two other process, *Nodes* and *TokenManager*. Upon receiving a signal from the *startActivity* channel, the two processes are started and when the *endActivity* channel communicates some data, it




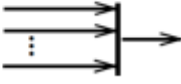



Decision 	Merge 
Fork 	Join 
Activity Final 	Final Flow 
Initial 	

Table 2 – Control nodes representations



Control flow 	Object flow 
---	---

Table 3 – Edges representations

means the termination of the activity. The synchronization is represented by the || symbol and contains a synchronization alphabet, which in this case is specified as $\{update, clear, endDiagram\}$. The *TokenManager* process is responsible for managing the active tokens in that activity diagram. The process *Nodes* invokes i other sub processes, where i is the number of nodes of that diagram. Each process synchronizes with each other through its own synchronization alphabet represented in this figure by $\alpha(n_n)$, where each of these parameters represents a specific process of a node. In addition, node

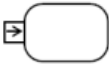


Input pin	Output pin	Activity parameter
		

Table 4 – Object nodes representations

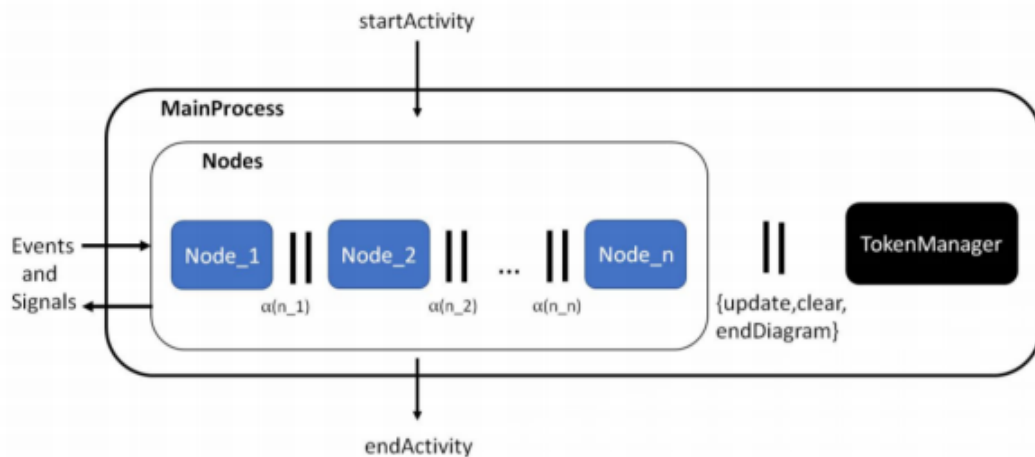


Figure 8 – CSP specification structure
 Source: (LIMA; TAVARES; NOGUEIRA, 2020)

processes can also synchronize through send signal and accept event node events, these events can be used to synchronize one or more activities that may be running in parallel.

In this work we use the semantics already defined in (LIMA; DIDIER; CORNÉLIO, 2013; LIMA, 2016; LIMA; TAVARES; NOGUEIRA, 2020). However, throughout the work we realized the need to define semantics to support more types of data, in this case, composite data types. That's why we defined two new mapping rules for generating CSP semantics to support these new data types, these semantics are better defined in the following subsections.

3.3.1 Datatype

In order to facilitate understanding, we show the graphical representations of the elements instead of their respective JSON.

In the *Find and identify* diagram of Figure 9 it is possible to notice that the parameters are of non-primitive types. Let's take the input parameter *shImage* for example. If we open the specification of the data type *SH_Image*, shown in Figure 10, it is possible to verify the existence of two attributes: *pix* and *header*, both of them are from the primitive type *Integer*.

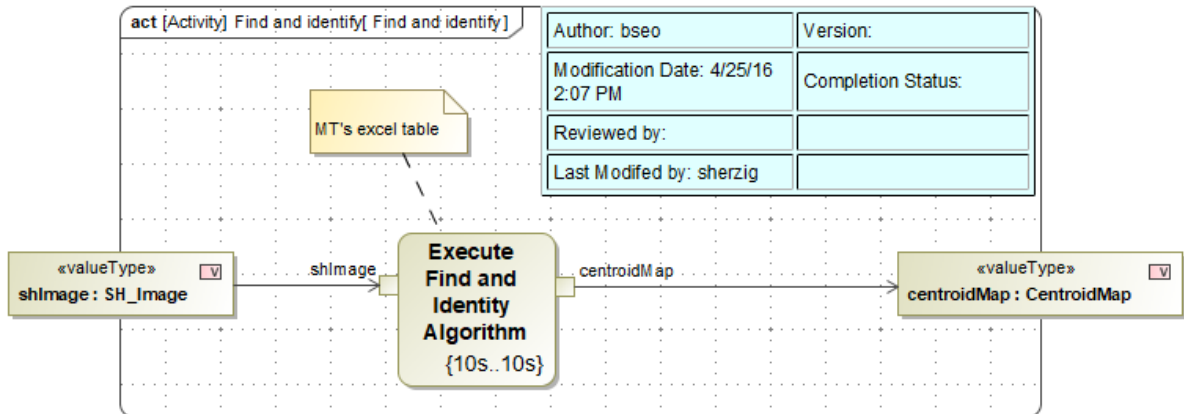


Figure 9 – Find and Identify diagram available on the MMS

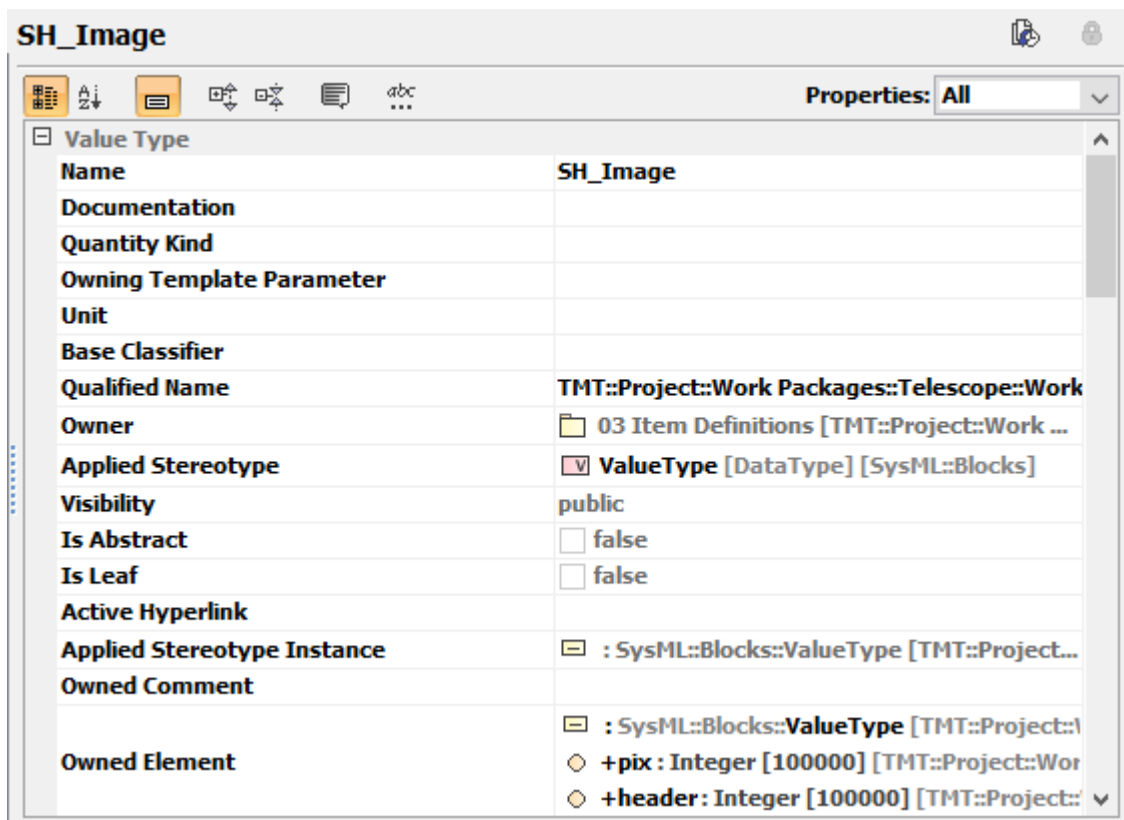


Figure 10 – Specification of the SH_Image datatype

The semantics starts with the creation of each attribute's type. The left part is the $\{attributeType+ _ +DiagramName\}$ and the right part is the range of value of this specific type. As FDR uses a method close to brute force, it is necessary that a range of values be stipulated for each type so that there is no memory overflow of possible combinations. If we were to write generically we would have:

```
attribute1Type_DiagramName = {TypeMinimumValue .. TypeMaximumValue}
...
attributeNType_DiagramName = {TypeMinimumValue .. TypeMaximumValue}
```

After that, we define each attribute. For it, the semantic is defined as follows: $\{attributeName+ _ +DiagramName\} = \{attributeType+ _ +DiagramName\}$. In a generic way:

```
attribute1_DiagramName = attribute1Type_DiagramName
...
attributeN_DiagramName = attributeNType_DiagramName
```

For the last part we generate the specification of the datatype, in CSP there is the reserved word *datatype*. By using it together with the prefix *type*, the type name and the suffix *DiagramName*, we write the left part. For the right part we have the type name, the same suffix and for each attribute that the datatype has, we add $\{attributeName + _ + suffix\}$. The main part result is the following:

```
datatype type_NameType_DiagramName =
NameType_DiagramName . attribute1_DiagramName . . . attributeN_DiagramName
```

Using these generic structures, we can develop a concrete example for *SH_Image*, which can be seen in the following code snippet:

```
Int_Findandidentify = {0..1}
pix_Findandidentify = Int_Findandidentify
header_Findandidentify = Int_Findandidentify
datatype type_SH_Image_Findandidentify =
SH_Image_Findandidentify . pix_Findandidentify . header_Findandidentify
```

Originally the *Integer* type of *pix* and *header* goes up to 100000, however to avoid a memory overflow we reduced it's range, in this case the chosen interval was 0 to 1.

3.3.2 Enumeration

Similar to the last semantic, for this one, in the left part we use the same reserved word *datatype*, the prefix *type* and the type name. On the right part we write the name of each contained element separated by the character `|`. The generic way of writing this is:

```
datatype type_NameType = Element1 | Element2 | ... | ElementN
```

We can observe the diagram *Select Filter SH* available in Figure 11. This time the input parameter is of type *SH Filter*, if we look at its specification, available in Figure 12, we can see that it is an enumeration that contains seven possible values: *NPH-1/BB30*, *BB-10(Backup)*, *BB-3*, *BB-1*, *NPH-2/SH*, *BB-100* and *Open*, the exact meaning of each name is irrelevant to the specification. So for *SH Filter*, using the generic formula we generate the following specification:

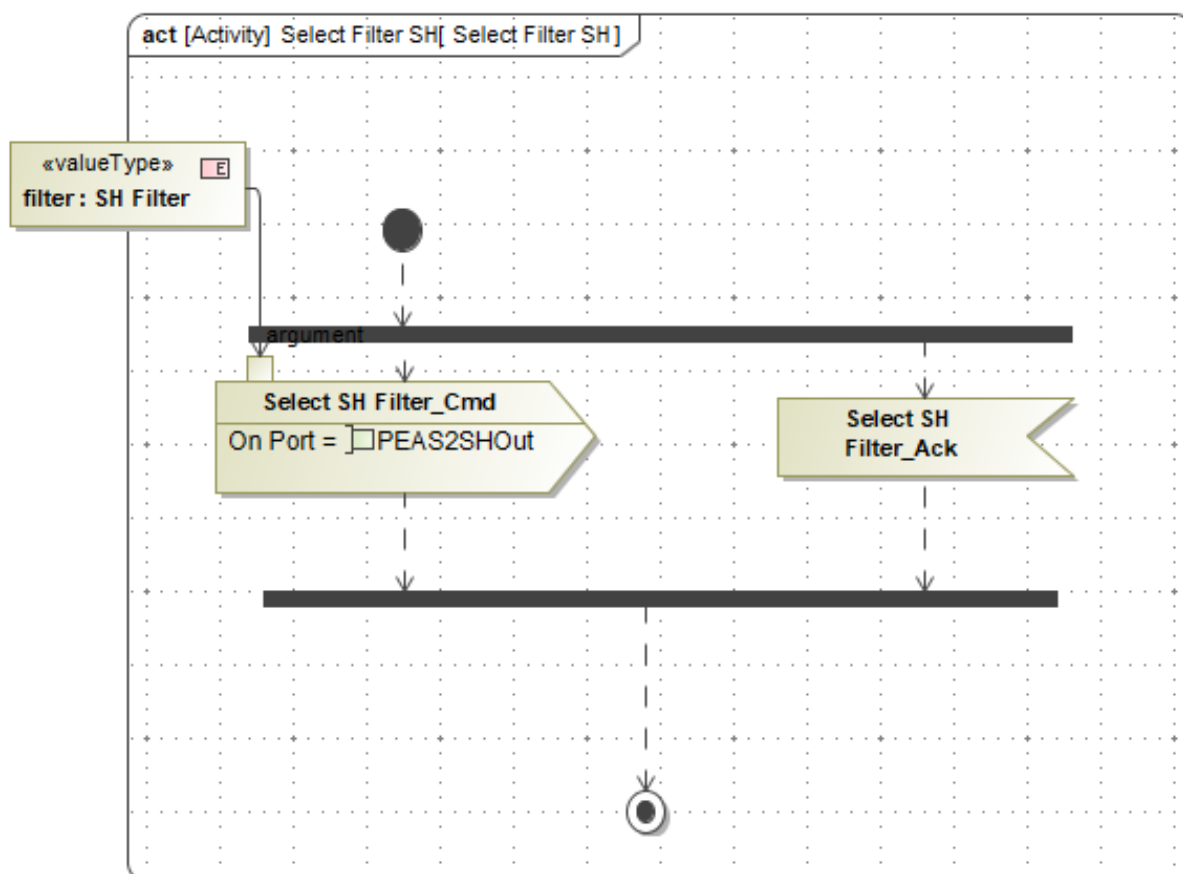


Figure 11 – Select Filter SH diagram available on the MMS

```
datatype type_SHFilter = NPH_1_BB_30 | BB_10_Backup_ | BB_3 | BB_1 |
NPH_2_SH | BB_100 | Open
```

3.4 Verification and traceability

To perform the checks on the FDR via code, it is necessary to add the commands in the code, for this we add the following two lines of code, where MAIN is the CSP process that represents the semantics of an activity.

```
assert MAIN :[ deadlock free ]
assert MAIN :[ deterministic ]
```

The first line is used to verify the existence of deadlocks, while the second line is used to verify if the CSP process is deterministic. When reading these commands, FDR internally uses the global analysis technique, exhaustively checking the model. That is why it is fundamental to propose an well-defined and less complex semantics together with optimizing function to reduce the state space when possible. Once finished, if the property does not hold, the tool creates a CSP event trace that represents the entire path taken by the specification to that violation. To represent an event trace, let's assume

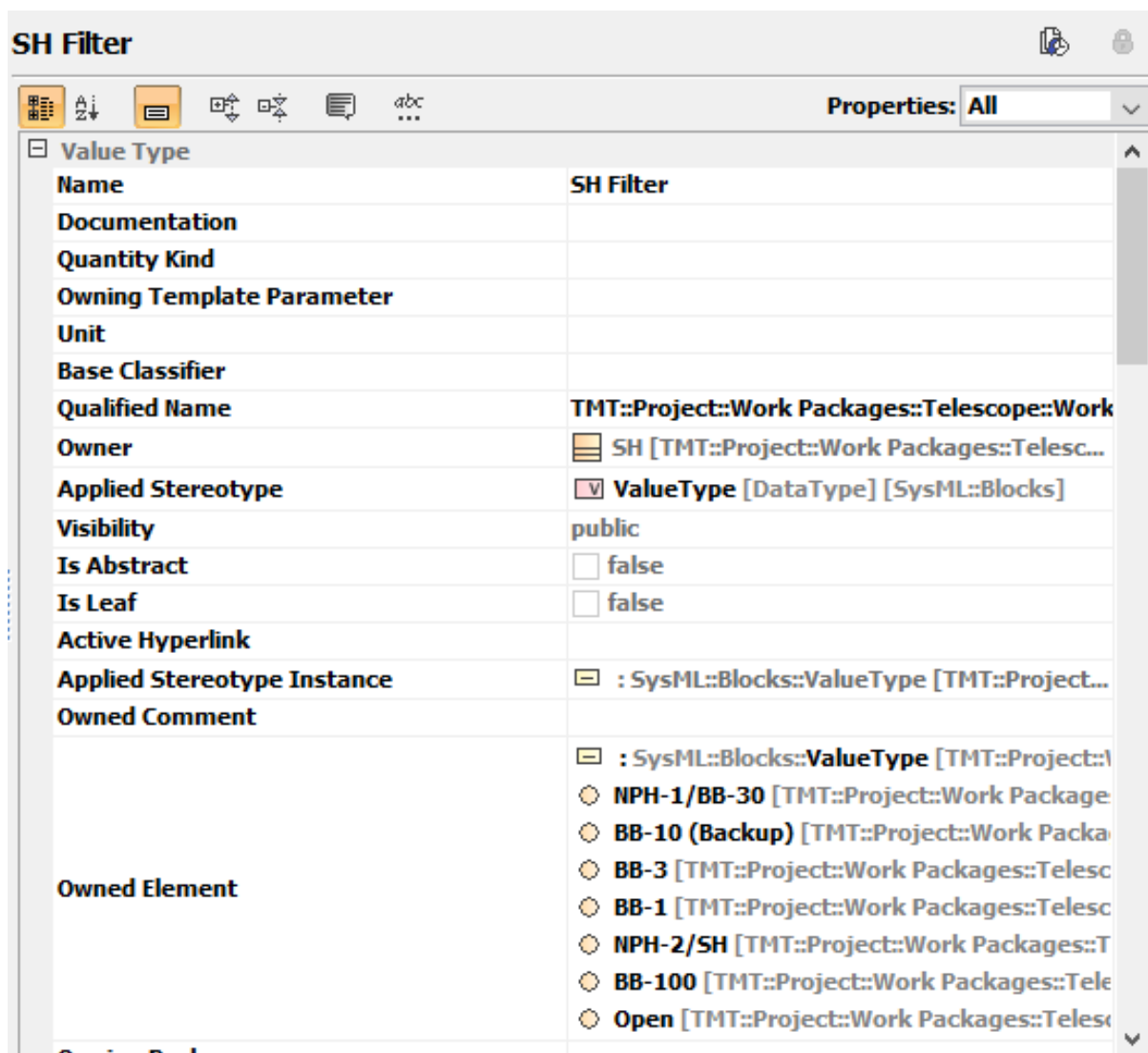


Figure 12 – Specification of the enumeration SHFilter

that Figure 9 had the outgoing edge of the opaque action removed, thus generating a deadlock. The generated trace would be this:

```
<startActivity_Findandidentify.1.SH_Image_Findandidentify.1.0 ,
update_Findandidentify.1.2.1 ,
oe_1_Findandidentify.1.SH_Image_Findandidentify.1.0 ,
event_ExecuteFindandIdentityAlgorithm_Findandidentify.1>
```

In this, the first represents the start of the activity, the second a token update, the third the data traffic over the object edge and the fourth the event performed by the Execute Find and Identity Algorithm action. This event trace is then returned to the framework. To perform traceability, we take the alphabet of each node and identify which nodes and edges were traversed during its execution. For this example we would get the following list of ids:

```
[_18_0_5_c0402fd_1461618020872_818345_179201 ,
```

```
_18_0_5_c0402fd_1461618443836_316998_180228 ,  
_18_0_5_c0402fd_1461618422523_981297_180201 ,  
_18_0_2_baa02e2_1415755464319_82416_80418]
```

These ids represent, respectively: the input parameter *shImage*, its outgoing edge, the input pin of the action *Execute Find and Identity Algorithm* and the action itself. Once we have the list of nodes and edges in hand, we create a list of JSONObjects where each element contains the diagram name, the ids of each node and edge traversed by it and the id of its corresponding invocation. At the end this list is added to a single JSONObject which is then returned as the counterexample. Figure 13 shows what the generated counterexample JSON for that list of ids would look like.

```
{ "CounterExampleIds": [  
  {  
    "Find and identify": [  
      "_18_0_5_c0402fd_1461618020872_818345_179201",  
      "_18_0_5_c0402fd_1461618443836_316998_180228",  
      "_18_0_5_c0402fd_1461618422523_981297_180201",  
      "_18_0_2_baa02e2_1415755464319_82416_80418"  
    ],  
    "Id": 1  
  }  
]
```

Figure 13 – Counterexample JSON

4 Evaluation

This chapter has two sections, Section 4.1 shows the necessary preparations for creating the environment, while Section 4.2 talks about the study case.

4.1 Environment preparation

During the first stages of this work, we used the public version of MMS. In this version, our privileges only allowed the reading of existing models in the MMS. This ended up limiting the development of the framework. Moreover after a period of time this version was deactivated, thus making it impossible for the work to continue in that way. In order to continue the work, we looked for other public versions of MMS, however, the option we found was an outdated version not compatible with our work. So, we decided that it would be necessary to set up our own instance of an MMS that was compatible with the latest release we had already developed. Another advantage of setting up our own instance of the MMS is that we would be able to make changes to the files, thus, being able to obtain more significant results.

To be able to create our own version of MMS, first, we have identified the technologies required to set a new server. Searching the existing public documents of OpenMBEE, we found tutorials on one of their GitHub pages on how to use Docker (MERKEL, 2014) for this. From these tutorials, we used the following technologies to develop our version:

- Docker
- Java SE Development Kit 11
- Postgresql
- Elasticsearch
- MinIO
- Gradle
- Spring

Version 4.0.7 of the MMS Reference Implementation or MMSRI, which is used in this project, was developed in the Java language, so it was necessary to use the *Java SE Development Kit 11* (JDK 11). For the storage part, *Postgresql* and *minIO* were used,

Elasticsearch for the search part, *Gradle* and *Spring* for the construction part. The usage of these technologies was facilitated through *Docker*, which allowed us to use the Docker compose technique that loads the images of the versions we want of these technologies, through a single file. The images used in Docker compose were: `postgres:11-alpine` for *Postgresql*, `docker.elastic.co/elasticsearch/elasticsearch:7.8.1` for *elasticsearch* and `minio/minio:latest` for *minIO*, in this case the version of the *minIO* is the latest released. In addition, a Linux virtual machine was created to run the Docker compose and activate the local server. Figure 14 shows the Swagger interface generated by our version. Even though we generate all these tabs, in this work we only use the following tabs:

- *Auth*: Responsible for carrying out requests related to authentication.
- *Elements*: Main tab used, it is responsible for requests to all individual model elements, such as a node or an edge.
- *Projects*: Tab responsible for making requests about the project.
- *Refs*: Tab responsible for the project version.

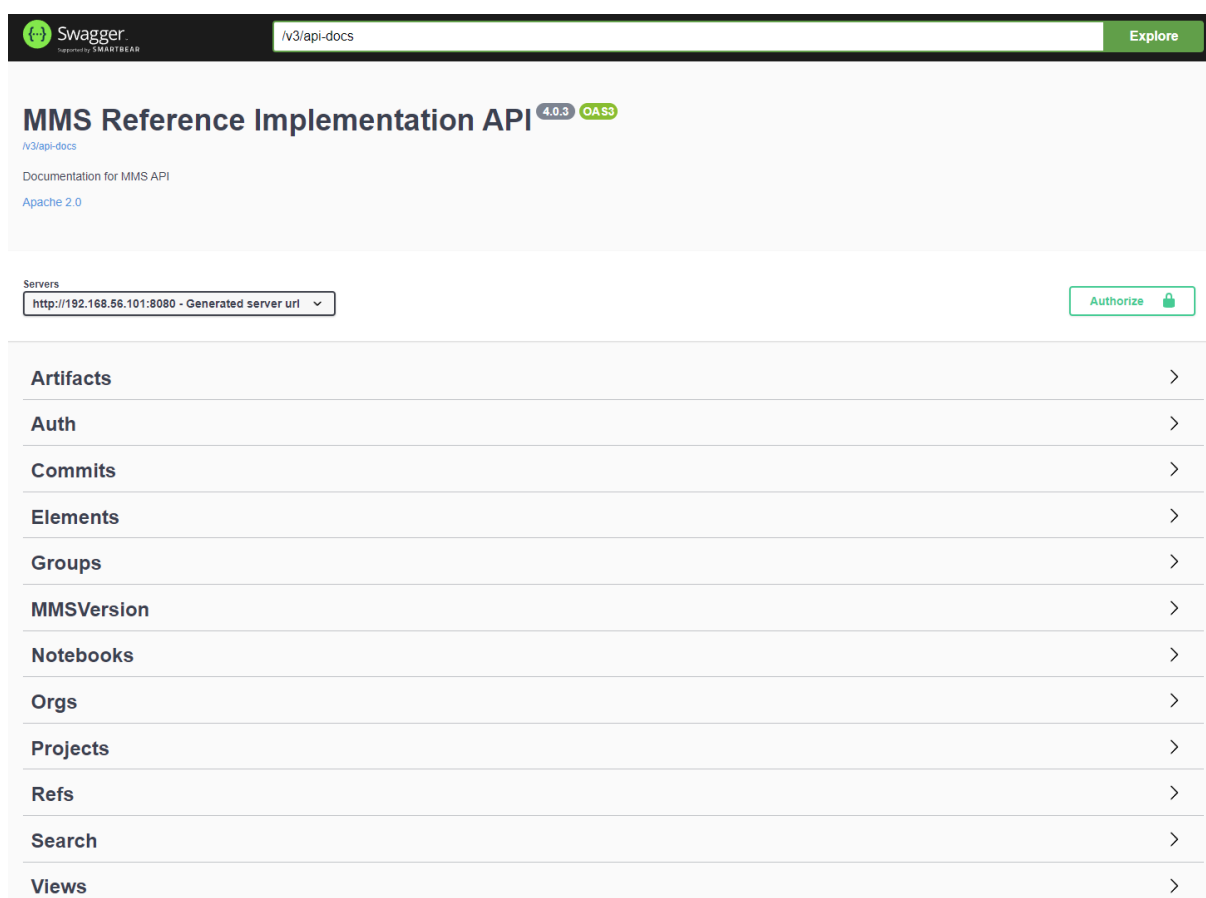


Figure 14 – Swagger interface

4.2 Case study

Now that we have the MMS running, let's talk about our case study. For this work we used the diagrams from the TMT project, whose name comes from the abbreviation for Thirty-Meter Telescope. According to the project's official website: "The Thirty Meter Telescope is a new class of extremely large telescopes that will allow us to see deeper into space and observe cosmic objects with unprecedented sensitivity. With its 30m diameter prime-mirror, TMT will be three times as wide, with nine times more area, than the largest currently existing visible-light telescope in the world. This will provide an unparalleled resolution, with TMT images more than 12 times sharper than those from the Hubble Space Telescope." ([OBSERVATORY, 2021](#)). This project is about a telescope located on the island of Hawaii, which started its project in 2003. This project has the support of several countries such as the USA, Japan and India. To start working on our case study, we copied the project available in the public MMS to our private version of MMS. With that, now it is possible to make changes to the TMT project model.

4.2.1 Structural verification

TMT is a very ambitious project. According to his document Detailed Science Case ([OBSERVATORY, 2021](#)), he aims to answer some of the great questions of astronomy such as:

- What is the nature and composition of the Universe?
- When did the first galaxies form and how did they evolve?
- What is the relationship between black holes and galaxies?
- How do stars and planets form?
- What is the nature of extrasolar planets?
- Is there life elsewhere in the Universe?

For this, he plans to cooperate with some of the biggest telescopes of this decade like the James Webb Space Telescope, Large Synoptic Survey Telescope and Atacama Large Millimeter Array. Its SysML model is considerable large, comprising several elements like, requirements, blocks, use cases, state machines, and activities. Of course, we are interested in the latter, and, although there are several functionalities described in terms of activities, we decided to focus on the *Setup APS, Acquire and Start Guiding* activity, which is presented, in Figure 15. This part of the project is responsible for configuring the Alignment and Phasing System (APS) in preparation for a specific alignment activity, acquires and guides on a new object and starts the Pupil Image

Tracking loop (PIT) depending on the input parameters to the activity. In this figure we can see seven Call Behavior Action nodes: *Ask OP to select a Star*, *Configure APS for SH Test*, *Stop PIT Tracking*, *Send Star Coords to ExecSW for Acquisition and Orientation*, *Save M1CS Configuration*, *Start PIT Tracking* and the already known from Figure 2, *Retrieve Reference Beam Map*. To give an idea of the total size of the *Setup APS, Acquire and Start Guiding* diagram, it contains more than 150 edges and almost 200 nodes along it and all the diagrams invoked by it. In addition, within these nodes there are also several forks that generate concurrent flows, thus increasing the complexity even more.

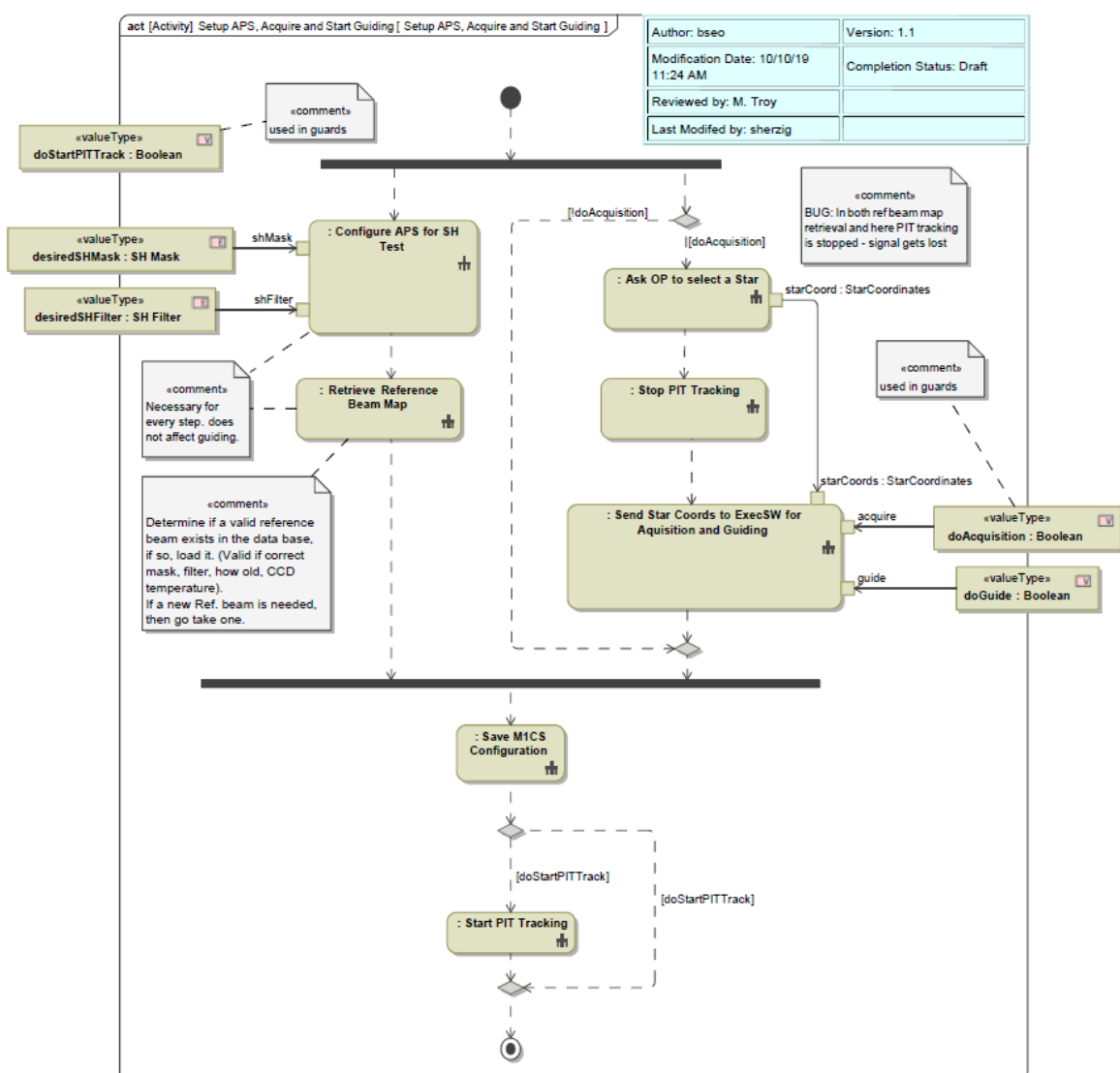


Figure 15 – Diagram Setup APS, Acquire and Start Guiding

The first step carried out was to use our framework to find well-formedness flaws in the model. We recall that the model must be consistently structured to translate the activity to CSP. Some of these structural checks are captured by our framework. Using it, we could identify the following syntactic problems:

- In diagram *Ask OP to select a Star*:
 1. One pin without base type
 2. Datatype *starcoord* with three attributes without types
- In diagram *Retrieve Reference Beam Map*:
 1. Decision node with ingoing object flow but outgoing control flow
 2. One useless pin
- In diagram *Check for existing reference beam map*:
 1. Three pins without base type
- In diagram *Find and Identify*:
 1. Datatype *SHImage* with one attribute without type
 2. Call Behavior without *behaviorId*(does not invoke any diagram)
 3. Output parameter using a type without any attributes
 4. One pin using a type without any attributes
- In diagram *Calibrate Reference Beam Map*:
 1. Call Behavior that invokes an empty diagram

To solve these problems, we changed these diagrams as follows: Object nodes and attributes without a type were given a type, the useless pin was removed, the Call Behaviors invoking empty diagrams were replaced by Opaque actions and the edges of the decision node were swapped to make them all of type object flow. Once these fixes were performed, we were able to translate these diagrams into a CSP specification.

4.2.2 Deadlock and non-determinism verification

Once the CSP specifications have been generated, the framework send them to the FDR tool to perform verification on the diagrams. By checking the diagram in Figure 2, we were able to identify a deadlock caused by an Accept Event node. This node is located in the diagram invoked by the Call Behavior node *Get CCD Temperature*, shown on Figure 16. This node waits indefinitely for a signal sent by a send signal action that matches this accept event action. However, this corresponding node is not present in any verified diagram, so the signal never arrives, causing the execution of this invocation not to proceed and consequently the junction node of the main diagram is never reached. To visually represent the path taken during execution, it is necessary to use some tool that supports this function, however, for demonstration purposes only, Figure 17 shows the

path taken in red. In addition to this case, it was possible to find some other deadlocks along the diagrams. However, the others were caused due to the models consider the relationship between activities and state machines. For instance, a signal can be sent by a state machine and received in a activity to trigger some behavior. Nevertheless, our framework currently does not support state machines and this relationship. Therefore, the generated specifications would wait indefinitely for communications with events from state machines that would never happen. We plan to consider these relationship in future works.

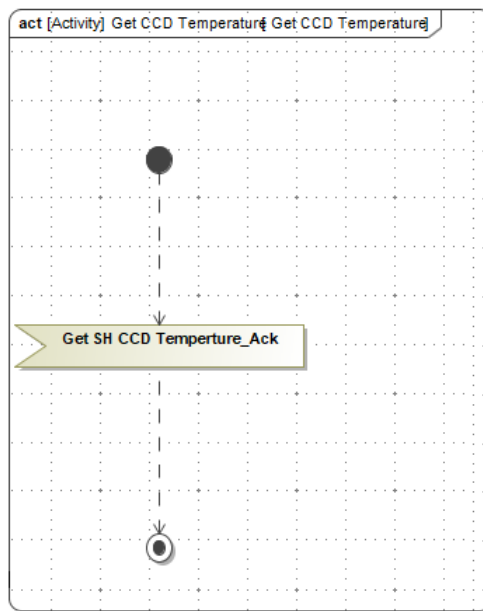


Figure 16 – Get CCD Temperature diagram

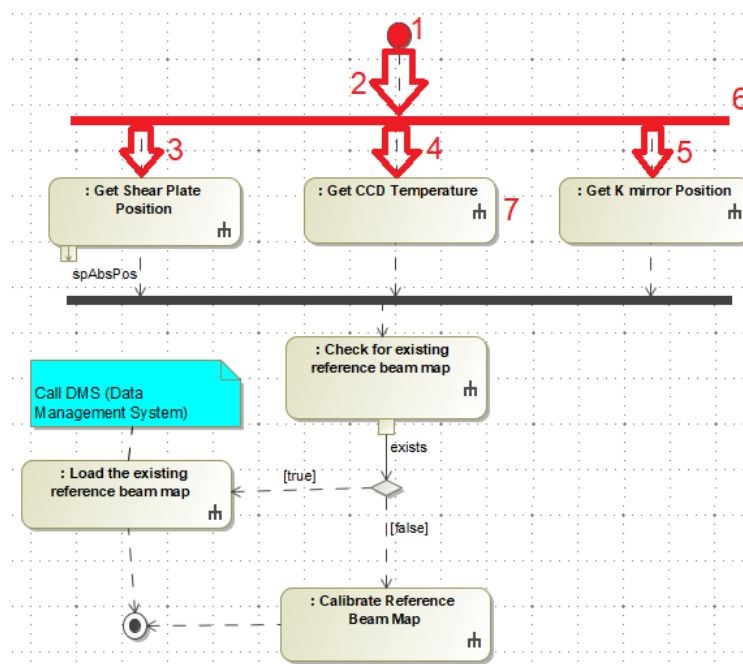


Figure 17 – Visual representation of the path

Figure 18 shows a modified version of the JSON counterexample of the deadlock found in the case of Figure 17. In this version, an arrow was included in the right part of each id to identify which type each one represents. In addition, we use ellipses to represent some of the elements returned in the counterexample for the activities *Get Shear Plate Position* and *Get K mirror Position* to improve readability. The id `_17_0_2_3_41e01aa_1382473227299_833303_50967` represents the id of the activity as a whole, not of an activity-type node. The green numbers on the right in Figure 18 show the correspondence between the id and the element in Figure 17.

```

{
  "CounterExampleIds": [
    {
      "Retrieve Reference Beam Map": [
        "_17_0_2_3_41e01aa_1382473227342_211556_51010", <- CBA 7
        "_17_0_2_3_41e01aa_1382473227337_507823_50989", <- Fork node 6
        "_17_0_2_3_41e01aa_1382473227342_572877_51012", <- Control flow 5
        "_17_0_2_3_41e01aa_1382473227345_555237_51032", <- Control flow 4
        "_17_0_2_3_41e01aa_1382473227345_349639_51034", <- Control flow 3
        "_17_0_2_3_41e01aa_1382473227345_853712_51037", <- Control flow 2
        "_17_0_2_3_41e01aa_1382473227338_832991_50990", <- Initial node 1
        "_17_0_2_3_41e01aa_1382473227299_833303_50967"], <- Activity
      "Id": 1
    },
    {
      "Get K mirror Position": [...]
    },
    {
      "Get Shear Plate Position": [...]
    },
    {
      "Get CCD Temperature": [
        "_17_0_2_3_41e01aa_1380816101132_236268_46770", <- Control flow
        "_18_0_5_baa02e2_1457571730513_308097_151125"], <- Initial node
      "Id": 1
    }
  ]
}

```

Figure 18 – Modified version of the counterexample JSON

Regarding non-determinism, a curious case was found in one of the diagrams in which both guards of a decision node were identical. In this case, it is not possible to control which of the two will be used by the system, thus, causing non-determinism. Due to this non-determinism, it is possible that the call behavior action node *Start PIT Tracking*, from Figure 19, is not invoked whenever it should. Just like when a deadlock is encountered, in this case a JSON file is also generated describing the IDs of the elements returned by the counterexample. Figure 20, shows another modified JSON file, this time representing the non-determinism path found. As in the other modified JSON, the green numbers on the right represent the id equivalent to the element from Figure 19.

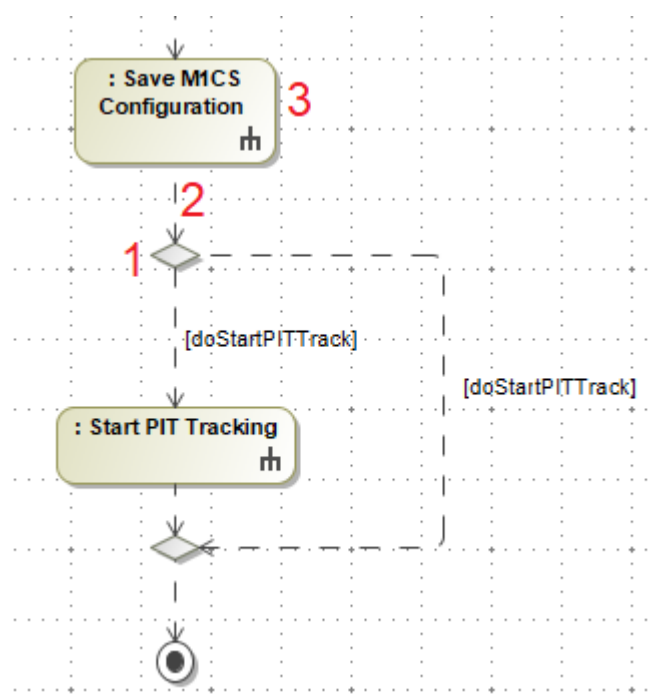


Figure 19 – Decision node with identical guards

```

{"CounterExampleIds":[
  {
    "Setup APS, Acquire and Start Guiding":[
      "_18_0_5_c0402fd_1467305249274_256775_210226", <- CBA 3
      "_17_0_2_3_41e01aa_1386574999938_547871_76850", <- Control flow 2
      "_17_0_2_3_41e01aa_1386575770058_85064_77118", <- Decision node 1
      ...
      "_17_0_2_3_41e01aa_1386574999937_219862_76836"], <- Initial node
      "Id":1
    ]
  }
  ... <- all others invoked diagrams
]
}

```

Figure 20 – Modified JSON from non-determinism counterexample

5 Conclusion

Throughout this work, we presented the most recent version of our framework. Now it is possible, up to a certain point, perform deadlock and non-determinism checking on SysML activity diagrams that are available in the MMS open environment of the OpenMBEE group, thanks to the new OpenMBEE sub module. For this version, we use the formal semantics defined in previous works, which is based on the CSP process algebra. In addition, the semantics for composite data types as datatypes and enumerations has been defined. With the addition of these semantics, we aimed to increase the expressiveness of our tool so that it could cover a greater amount of modeling scenarios and consequently more real industry cases.

Although we made several additions to the framework structure, we managed to keep all of them at a level hidden to the user. Thus, it can remain at the diagrammatic level without the need to carry out algebraic manipulations on the CSP semantics present in the verification. Considering that the formal CSP method is quite complex and uses very rigorous algebra, a lot of users would not feel attracted to use it. Therefore, we believe that the new traceability mechanism, based on JSON files, will increase the attractiveness of our framework for all those who wish to perform manipulations on the models available in the MMS.

5.1 Related works

The works described in this section use an approach similar to our strategy, that is, translating activity diagrams to an underlying notation, usually for verification purposes. These other languages, in turn, are able to generate a version equivalent to the original diagram, but with the ability to perform property verifications through verifiers.

In the work of ([ELMANSOURI; HAMROUCHE; CHAOUI, 2011](#)), they proposed to use a meta-model for UML activity diagrams and a graph grammar that performs the transformation of these diagrams, created in the AToM³ tool, into CSP semantics. Despite the defined semantics being able to perform checks, none were performed in the work.

In ([LÓPEZ-GRAO; MERSEGUER; CAMPOS, 2004](#)), a method is used to transform activity diagrams into a Labeled Generalized Stochastic Petri Net (LGSPN). Once these LGSPNs are composed, they are analyzed using the GreatSPN tool to obtain performance indices. In this work, deadlock or non-determinism checks can not be

performed.

(GUELF; MAMMAR, 2006) turns UML activity diagrams into XPDL specifications. However, the semantics defined by this work do not have any formal rules to support them, in addition their authors state that the completeness or correctness of these semantics are not guaranteed.

(OUCHANI; MOHAMED; DEBBABI, 2014) performs the verification of SysML activity diagrams through probabilistic systems. This check is done by transforming activity diagrams into equivalent PRISM models. After that, the PRISM model checker is used to make the checks. The work deals with checks such as deadlock, however, it does not provide any traceability mechanism.

(RAHIM; HAMMAD; BOUKALA-IOUALALEN, 2015) transforms activity diagrams into modular PETRI nets, this transformation is done through meta models implemented in the TOPCASED tool. After that, these meta models can be checked and detect properties such as deadlock. Like the work mentioned above, this one also does not present traceability mechanisms.

5.2 Limitations and future work

Currently, the framework is presented in a working way for some models. However, the tool does not yet support some SysML activity diagram elements such as *local precondition*, *local postcondition* nor *probability* or *rate*. Furthermore, when dealing with very large and complex models, due to the nature of the FDR tool verification, the amount of states that the LTS has to verify can grow exponentially. Therefore, currently, our tool limits the range of values allowed in primitive data types and consequently composite types as well, thus reducing the number of verified states.

Therefore, in future works we hope to be able to correct these problems. In addition, we also intend to continue increasing the expressiveness of the framework by adding new modeling environments and new types of diagrams. So that it is possible to verify models with more than one type of diagram, such as the case previously mentioned in Subsection 4.2.2. We also consider expanding the existing ones, such as implementing a plugin for modeling tools that use the OpenMBEE's MMS to generate a counterexample diagram instead of a counterexample JSON. Finally, carry out a more detailed scalability study, to be able to identify which are the bottlenecks in our approach and also what is the limit of it.

Bibliography

- ANSYS. *Ansys SCADE Suite - Ansys SCADE Suite Model-Based Development Environment for Critical Embedded Software*. 2022. Available in: <<https://www.ansys.com/products/embedded-software/ansys-scade-suite>>. Access date: March 23, 2022. Citado na página 13.
- DOMINGUES, R. Verificação de deadlock e não-determinismo em diagramas de atividades com suporte a sinais e invocações de comportamento. *XXX CIC UFRPE*, 2021. Citado 4 vezes nas páginas 13, 14, 24, and 25.
- ELMANSOURI, R.; HAMROUCHE, H.; CHAOUI, A. From uml activity diagrams to csp expressions: A graph transformation approach using atom 3 tool. In: . [S.l.: s.n.], 2011. Citado na página 45.
- GIBSON-ROBINSON, T. et al. Fdr3 — a modern refinement checker for csp. In: ÁBRAHÁM, E.; HAVELUND, K. (Ed.). *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2014, (Lecture Notes in Computer Science, v. 8413). p. 187–201. ISBN 978-3-642-54861-1. Disponível em: <http://dx.doi.org/10.1007/978-3-642-54862-8_13>. Citado na página 13.
- GOSLING, J. et al. *Java Language Specification, Second Edition: The Java Series*. 2nd. ed. USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN 0201310082. Citado na página 24.
- GUELFY, N.; MAMMAR, A. A formal framework to generate xpdL specifications from uml activity diagrams. In: *Proceedings of the 2006 ACM symposium on Applied computing*. [S.l.: s.n.], 2006. p. 1224–1231. Citado na página 46.
- HASKINS, B. et al. 8.4.2 error cost escalation through the project life cycle. *INCOSE International Symposium*, v. 14, n. 1, p. 1723–1737, 2004. Disponível em: <<https://onlinelibrary.wiley.com/doi/abs/10.1002/j.2334-5837.2004.tb00608.x>>. Citado 2 vezes nas páginas 7 and 11.
- HOARE, C. A. R. *Communicating sequential processes*. [S.l.]: Prentice-Hall, Inc., 1985. ISBN 0-13-153271-5. Citado na página 12.
- LIMA, L. *FORMALISATION OF SYSML DESIGN MODELS AND AN ANALYSIS STRATEGY USING REFINEMENT*. Tese (Doutorado) — Centro de Informática - UFPE, 3 2016. Citado 2 vezes nas páginas 13 and 31.
- LIMA, L.; DIDIER, A.; CORNÉLIO, M. A formal semantics for sysml activity diagrams. In: IYODA, J.; MOURA, L. (Ed.). *Formal Methods: Foundations and Applications*. Springer Berlin Heidelberg, 2013, (Lecture Notes in Computer Science, v. 8195). p. 179–194. ISBN 978-3-642-41070-3. Disponível em: <http://dx.doi.org/10.1007/978-3-642-41071-0_13>. Citado 2 vezes nas páginas 13 and 31.
- LIMA, L.; TAVARES, A.; NOGUEIRA, S. C. A framework for verifying deadlock and nondeterminism in uml activity diagrams based on csp. *Science of Computer*

Programming, v. 197, p. 102497, 2020. ISSN 0167-6423. Citado 2 vezes nas páginas 13 and 31.

LÓPEZ-GRAO, J. P.; MERSEGUER, J.; CAMPOS, J. From uml activity diagrams to stochastic petri nets: application to software performance engineering. In: *Proceedings of the 4th international workshop on Software and performance*. [S.l.: s.n.], 2004. p. 25–36. Citado na página 45.

MATHWORKS. *Simulink - Simulation and Model-Based Design*. 2022. Available in: <https://www.mathworks.com/products/simulink.html>. Access date: March 23, 2022. Citado na página 13.

MERKEL, D. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, v. 2014, n. 239, p. 2, 2014. Citado na página 37.

OBSERVATORY, T. I. *Thirty Meter Telescope, Astronomy's Next-Generation Observatory*. 2021. Available in: <https://www.tmt.org/>. Access date: March 23, 2022. Citado na página 39.

OMG. *OMG Unified Modeling Language (OMG UML), Version 2.5.1*. [S.l.], 2017. Disponível em: <https://www.omg.org/spec/UML/About-UML/>. Citado na página 11.

OPENMBEE. *OpenMBEE - Open Model Based Engineering Environment*. 2021. Available in: <https://www.openmbee.org>. Access Date: October 14, 2021. Citado 2 vezes nas páginas 14 and 25.

OUCHANI, S.; MOHAMED, O. A.; DEBBABI, M. A formal verification framework for sysml activity diagrams. *Expert Systems with Applications*, v. 41, p. 2713–2728, 05 2014. Citado na página 46.

RAHIM, M.; HAMMAD, A.; BOUKALA-IOUALALEN, M. Towards the formal verification of sysml specifications: translation of activity diagrams into modular petri nets. In: IEEE. *2015 3rd International Conference on Applied Computing and Information Technology/2nd International Conference on Computational Science and Intelligence*. [S.l.], 2015. p. 509–516. Citado na página 46.

REGGIO, G. et al. What are the used uml diagrams? a preliminary survey. In: *EESSMOD@MoDELS*. [S.l.: s.n.], 2013. Citado 2 vezes nas páginas 12 and 21.

ROSCOE, A. *Understanding Concurrent Systems*. [S.l.]: Springer-Verlag London, 2010. ISBN 978-1-84882-258-0. Citado na página 20.

ROSCOE, B. *The theory and practice of concurrency*. 1998. Citado 2 vezes nas páginas 18 and 20.

SCHNEIDER, S. *Concurrent and Real Time Systems: The CSP Approach*. 1st. ed. USA: John Wiley & Sons, Inc., 1999. ISBN 0471623733. Citado na página 18.

VISION, C. *Astah - Premier Diagramming, Modeling Software and Tools*. 2019. Available in: <http://astah.net>. Access Date: October 14, 2021. Citado na página 13.