Universidade Federal Rural de Pernambuco
Programa de Pós-Graduação em Informática Aplicada

Waldeck Antônio de Melo Lindoso Júnior

VERIFYING ROBOTIC DESIGNS USING A DIAGRAMMATIC LANGUAGE FOR PROPERTIES

M.Sc. Dissertation

Recife
Aug 2022

Universidade Federal Rural de Pernambuco
Programa de Pós-Graduação em Informática Aplicada

Waldeck Antônio de Melo Lindoso Júnior

VERIFYING ROBOTIC DESIGNS USING A DIAGRAMMATIC LANGUAGE FOR PROPERTIES

A M.Sc. Dissertation presented to the Programa de Pós-Graduação em Informática Aplicada of Universidade Federal Rural de Pernambuco in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Advisor: *Sidney de Carvalho Nogueira*
Co-Advisor: *Lucas Albertins de Lima*

Recife
Aug 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)

L747v Lindoso Júnior, Waldeck Antônio de Melo
Verifying robotic designs using a diagrammatic language for properties / Waldeck Antônio de Melo Lindoso Júnior. - 2022.
56 f. : il.

Orientador: Sidney de Carvalho Nogueira.
Coorientador: Lucas Albertins de Lima.
Inclui referências e apêndice(s).

Dissertação (Mestrado) - Universidade Federal Rural de Pernambuco, Programa de Pós-Graduação em Informática Aplicada, Recife, 2022.

1. RoboChart properties. 2. Activity diagram. 3. Sequence diagram. 4. CSP. 5. FDR. I. Nogueira, Sidney de Carvalho, orient. II. Lima, Lucas Albertins de, coorient. III. Título

CDD 004

*I dedicate this dissertation to my family for their love
throughout my life, my friends for the patience and support,
and professors for the knowledge and inspiration.*

Acknowledgements

I would like to thank my parents and family for their encouragement and support.

My thanks to my supervisors Sidney de Carvalho Nogueira and Lucas Albertins de Lima. Thank you for sharing your knowledge, enthusiasm and for the time we shared together. I learned and grew a lot.

Thanks to my professors at PPGIA (Postgraduate Program in Applied Informatics) and to all my dear colleagues who shared this arduous and rewarding journey with me.

Special thanks to Ana Cavalcanti, Augusto Sampaio and Madiel C. Filho for their suggestions, contributions, shared time and feedback regarding this work. Thank you to everyone who contributed to my academic and personal growth. You will always be remembered.

Finally, my deepest gratitude to God for making all this possible, giving me strength and courage to carry out this work.

“(...) Life is made of moments, moments that we have to go through, whether good or not, for our learning. Nothing is by chance. We need to do our part, play our part on the stage of life, remembering that life doesn’t always follow our will, but it is perfect in what it has to be."

—CHICO XAVIER

Resumo

RoboChart é uma notação diagramática no estilo de UML para projetar sistemas robóticos. Possui uma semântica bem definida na notação de CSP o que permite a verificação automática das propriedades funcionais utilizando a ferramenta FDR. Apesar de RoboChart ser baseado em UML, propriedades específicas de modelos RoboChart, que são específicas de aplicações, precisam ser especificadas usando a notação de CSP; contra-exemplos gerados pela ferramenta FDR são especificados na sintaxe de CSP. Consequentemente, o projetista deve estar familiarizado com CSP para definir propriedades e entender os contra-exemplos.

Este trabalho propõe uma abordagem para a verificação automática de propriedades específicas de aplicação para RoboChart. A abordagem introduz uma notação diagramática, cuja semântica é definida na notação de CSP, para expressar propriedades de um modelo RoboChart. A notação mistura elementos sintáticos da notação de Diagrama de Atividades de UML com elementos de RoboChart. A notação proposta facilita a especificação dos padrões de abstração usados na definição de propriedades para modelos RoboChart.

Como ferramenta de suporte, desenvolvemos um plug-in para a ferramenta de modelagem Astah que traduz uma propriedade diagramática descrita na notação proposta para a respectiva semântica CSP e invoca o verificador de refinamentos FDR para verificar se a semântica do modelo RoboChart refina a propriedade especificada com a notação introduzida. Contra-exemplos são analisados e apresentados na notação de diagramas de sequência. A abordagem apresentada permite ao projetista especificar, verificar propriedades e interpretar contra-exemplos no nível diagramático. Como parte da validação, apresentamos um estudo de caso que mostra a aplicação da abordagem para modelar e verificar propriedades de dois sistemas robóticos projetados com RoboChart.

Palavras-chave: Propriedades de modelos RoboChart, Diagrama de Atividades, Diagrama de Sequência, Astah, CSP, FDR

Abstract

RoboChart is a diagrammatic notation in the style of UML for the design of robotic systems. It has a well-defined semantics in the notation of CSP that enables the automatic verification of functional properties using the FDR tool. Nonetheless, RoboChart is based on UML, application-specific properties of RoboChart models need to be specified using the CSP notation; counterexamples yielded by the FDR tool are specified in the CSP semantics. Consequently, the designer must be familiar with CSP to define properties and understand counterexamples.

This work proposes an approach for the automatic verification of application-specific properties for RoboChart. The approach introduces a diagrammatic notation, which has CSP as the underlying semantics, to express behaviour that characterises properties for a RoboChart model. The notation mixes standard elements of the activity diagram notation with elements of RoboChart. The notation eases the specification of abstraction patterns used in defining properties for RoboChart models.

As tool support, we develop a plug-in for the Astah modelling tool that translates the proposed diagrammatic property to the respective CSP semantics and calls the FDR refinement checker to verify whether the semantics of the RoboChart model refines the property specified with the introduced notation. Counterexamples are parsed and presented in the notation of sequence diagrams. The presented approach allows the designer to specify, verify properties and interpret counterexamples at the diagrammatic level. This work presents a case study that shows the application of the approach to model and verify properties of two robotic systems designed with RoboChart.

Keywords: RoboChart properties, Activity diagram, Sequence diagram, Astah, CSP, FDR

List of Figures

List of Tables

List of Acronyms

UML	Unified Modeling Language	12
CSP	Communicating Sequential Processes	12
FDR	Failures-Divergences Refinement	12
DSL	Domain-Specific Language	12
JVM	Java Virtual Machine	33
API	Application Programming Interface	33
UI	User Interface	34
LTS	Labelled Transition System	35

Contents

1 Introduction

In the context of critical systems, where failures can result in serious problems, such as the loss of human lives or financial loss, it is crucial to ensure that robot software satisfies the expected properties.

Due to the high complexity of robotic controller software, several Domain-Specific Language (DSL) for modelling and simulation [2, 27, 7] for robots have been proposed to support the validation and verification of robotic controllers. Test and simulation are often used to verify robots. However, they cannot ensure that the system has the expected properties. Thus, formal verification techniques become fundamental to ensure that such properties hold.

RoboChart [23] is a DSL that uses a diagrammatic notation based on state machines notation from Unified Modeling Language (UML) [26] to describe robot behaviour. The properties of RoboChart models are verified through formal verification; this contrasts with DSLs for the design of robotic simulation. RoboTool[1] provides a graphic editor for RoboChart and automatically generates the Communicating Sequential Processes (CSP) [29] specification for RoboChart models. This tool is integrated with the Failures-Divergences Refinement (FDR) refinement checker [8] that checks classical properties (for instance, deadlock freedom), as well as application-specific properties that are stated as process refinement assertions of CSP specifications. A property is defined as a set of traces that must be refined by the traces from the RoboChart model.

A current limitation of RoboTool is that the specification of domain-specific properties is defined in the notation of CSP. Figure 1.1 shows the flow the property designer follows to specify and verify an application-specific property for RoboChart. In the current flow, the properties are defined as CSP processes whose alphabet follows the encoding for RoboChart. Additionally, the counterexample yielded by the FDR tool is described in terms of the CSP encoding for the RoboChart model. The property designer must be familiar with CSP operators and the CSP alphabet representing the RoboChart elements relevant to the property. He uses the CSP notation to define the property and to interpret the FDR counterexample trace if the property does not hold. As reported in [24], unfamiliarity with formal notations can result in challenges for RoboTool

[1]https://robostar.cs.york.ac.uk/robotool/

users during the verification process. Since RoboChart is a diagrammatic notation, it would be more convenient to express properties and read the counterexample using some diagrammatic notation that hides details of the CSP semantics and is similar to some well-accepted modelling language.

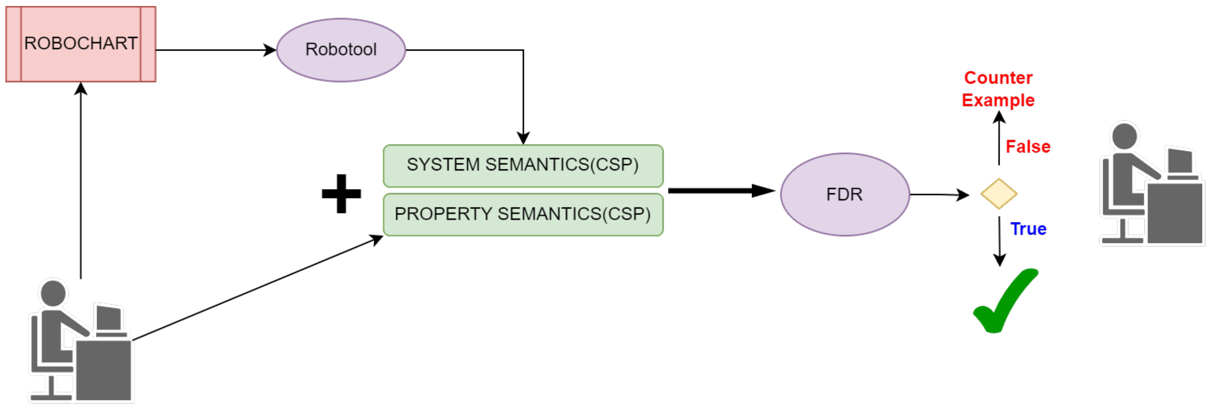


Figure 1.1: Checking properties in RoboChart

A classical property for a RoboChart model is an abstract behaviour defined as a CSP process that communicates visible events of a RoboChart model. A property holds if it is refined by the CSP process that captures the semantics for a given RoboChart component. A RoboChart property focuses on the behaviour of a single component and abstracts the structure of the component to be verified. In a similar way, activity diagrams describe the behaviour of a (sub) system while abstracting its structure.

Activity diagrams have been used for a variety of purposes. Business and system analysts use them to specify business processes, use cases and document the implementation of system processes. Moreover, they can be used to model system behaviour given the expressivity of their constructors that allows the modelling of condition, loop and concurrent behaviours. In previous work [14], we had defined the CSP semantics for an activity diagram. This allows activity diagrams to be used as a diagrammatic notation to express CSP processes. Nonetheless, the CSP semantics defined in [14] cannot express RoboChart properties because it does use the same encoding for the events that is used in the semantics for a RoboChart component.

This work defines a language based on the notation of activity diagrams of UML, which is used to specify application-specific properties for RoboChart. The language abstracts the internal structure of the components and aims at expressing the expected order for inputs, outputs and operations calls of RoboChart components. The proposed notation uses and extends the nodes of an activity diagram to make them compatible for expressing RoboChart events and operations. In addition, the language introduces stereotypes that express two common patterns of behaviour that ease the specification of properties. The language has well-defined semantics in the notation of CSP that has the same encoding of the RoboChart semantics; this enables the automatic verification of properties using FDR. A property holds if, and only if, the CSP process that represents the property is refined by the CSP process that represents the target RoboChart element, which can be a state machine, a controller or a module. Properties can be authored using the Astah UML tool [2]. Moreover, a plug-in for Astah has been developed to translate the property diagram to CSP and call FDR to verify the property of the target RoboChart element. The complete work has been submitted to a journal and is under revision.Additionally, if the property does not hold, the plugin parses the counterexample generated by FDR and presents it as a sequence diagram. In order to validate the approach, we present a case study that shows the application of the approach to model and verify properties of two robotic systems designed with RoboChart. Part of this work has already been published in [15].

Figure 1.2 shows how property definition and verification can be facilitated if the property designer uses the proposed plugin. The plugin supports the definition of the property in a notation similar to activity diagrams that is translated to CSP, then it calls FDR to verify whether the property holds. If a counter example is found, the plugin parses the FDR counterexample and displays the traces as a sequence diagram. The plugin completely hides the semantics of the CSP language.

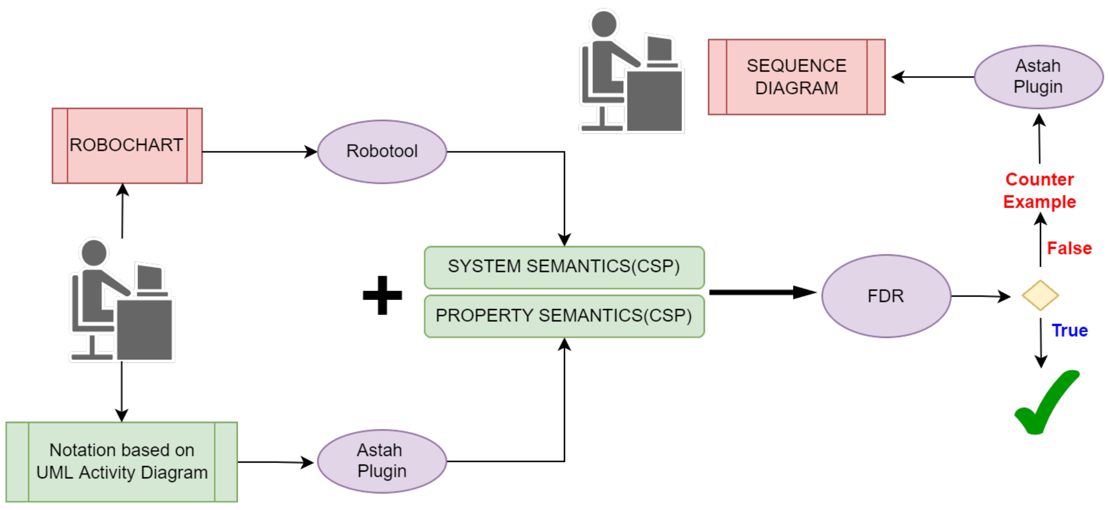


Figure 1.2: Checking properties in RoboChart - using the plugin

The remainder of this document is organised as follows. The next chapter presents the background for this work. Chapter 3 presents syntax and semantics for the language for specifying properties. Chapter 4 details the tool support developed to automatise the translation of the property to CSP, the verification of the property and the counterexample visualisation. Chapter 5 describes a case study related to the robotic domain. Chapter 6 discusses related work. Finally, Chapter 7 concludes and presents future works.

[2]https://astah.net/products/astah-uml/

2 Background

In this chapter, we present the basics of the RoboChart language, the UML diagrams used in this work and the notation of CSP.

2.1 RoboChart

RoboChart models specify the behaviour of a software that controls and interacts with a robotic platform (hardware). A module is the RoboChart component that specifies the flow of events between software controllers and a robotic platform. Controller behaviour is specified by state machines. Input and outputs events, as well as operation calls, are possible observations for a module, controller and state machine. Naturally, properties for a RoboChart model consider the expected order of such observations. In this chapter, we focus on the notation for a state machine used by this work. For further details on the notation of RoboChart, please refer to [23].

The robot model is composed of a module that must contain exactly one robotic platform and at least one controller. Figure 2.1 shows the CFootBot module that contains the FootBot robotic platform and ref Movement, a reference to the Movement controller. This specification is a simplification of the model that can be found in the RoboCalc website [1]. This module specified a simple mobile robot that changes directions when an obstacle is detected. In a module, all variables and operations required by the controllers must be provided by the robotic platform. The platform FootBot has a interface for movement (provided), represented by the letter 'P' and another for obstacle (defined), represented by the letter 'i'. The interface MovementI defines the operation move that receives two parameters: lv and av. They represent linear and angular movements of the robot. The call of such operation makes the robot to move. The interface ObstacleI specifies the event obstacle. Events represent atomic communications. In our example, the event obstacle is an input event coming from the platform; it represents the robot encountering a obstacle. Still in Figure 2.1, we can see a connection (arrow) connecting the obstacle event leaving the platform (provided) and arriving at the obstacle event of the controller

[1] `https://www.cs.york.ac.uk/circus/RoboCalc/other_examples/SimFW/index.html`

(required).

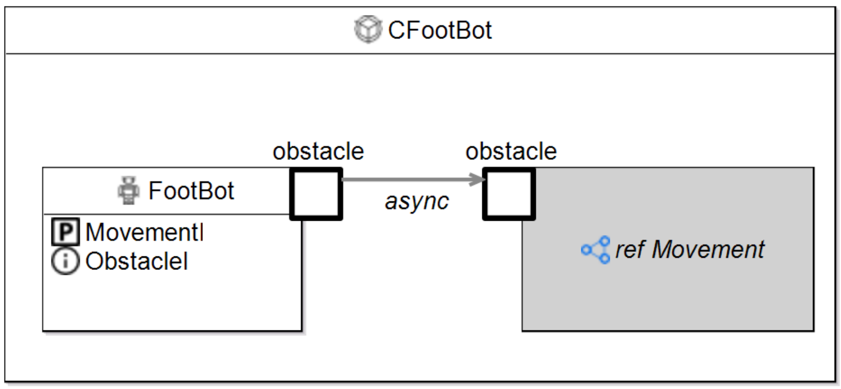


Figure 2.1: CFootBot module

In a controller it is necessary to have at least one state machine. In Figure 2.2 we can see the model for the Movement controller that contains a reference to the SMovement state machine. Such a controller has two interfaces: one for movement (required), represented by the letter 'R' and another for obstacle (defined), represented by the letter 'i'. In the same figure we can observe the connection (arrow) that links the obstacle event of the controller to the event of the state machine.

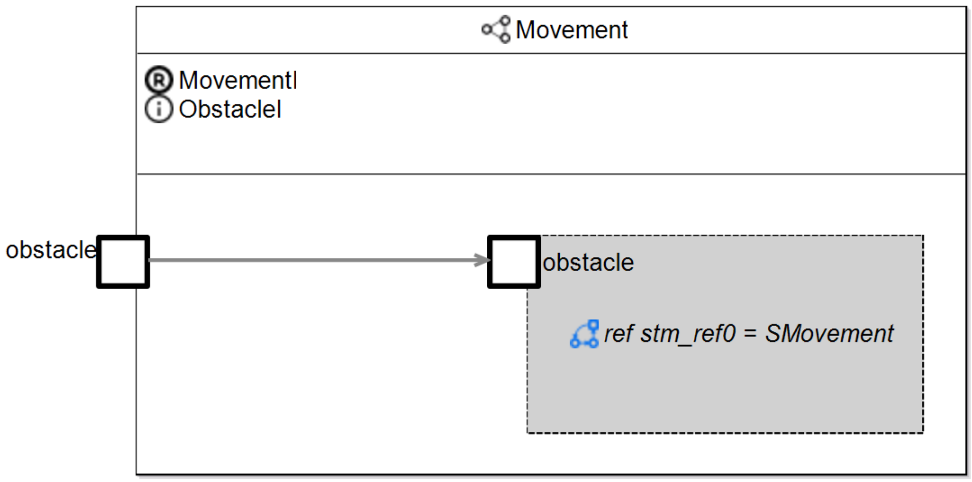


Figure 2.2: SimFW robot - controller

Figure 2.3 presents the specification for the state machine SMovement. Such a state machine requires the interface MovimentI and uses the interface ObstacleI. Moreover, it defines the constants lv and av.

Each state machine is composed of states, pseudostates and transitions. States can have actions: entry, during and exit. In the SMovement state machine, the initial node leads to the state Moving, which has an entry action that calls the operation move(lv,0) that makes the robot to perform a linear movement (av equals zero). When an obstacle is detected, the event obstacle triggers a transition that leads to the Turning state. The Turning state has an entry action that

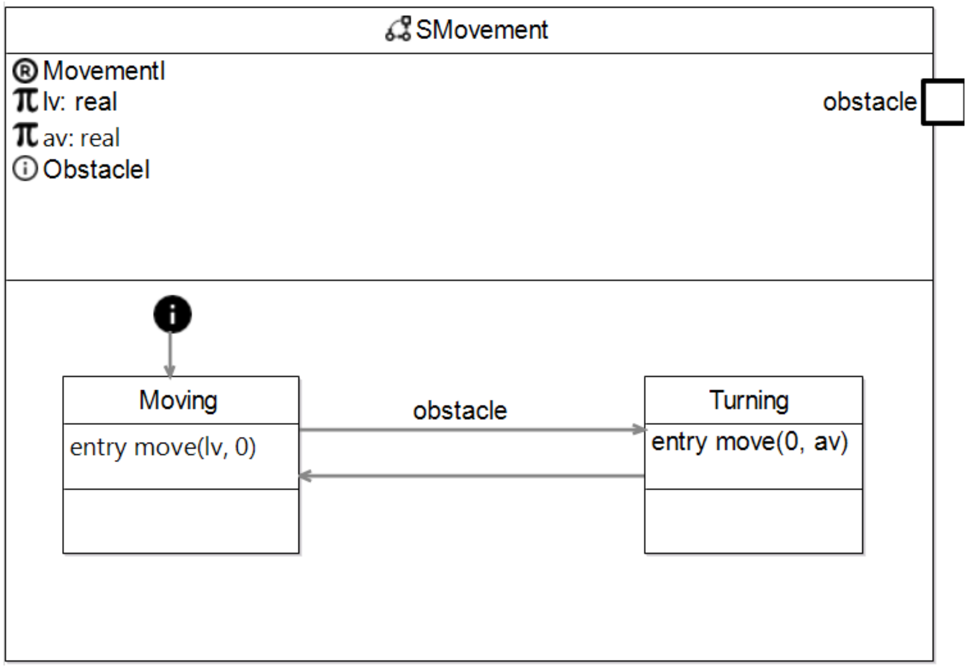


Figure 2.3: SimFW robot - state machine

executes an angular movement move(0,av), causing the robot to rotate, dodging the obstacle. In sequence, a transition to the Moving state is triggered.

2.2 UML Diagrams

A UML activity diagram is a graph of activity nodes interconnected by activity edges [26]. An activity node can be either an action node, an object node or a control node. Activity edges are directed connections between two activity nodes. They can be either a control flow used to sequence the execution of activity nodes explicitly, or an object flow, which can communicate data between two nodes. Action nodes execute the desired behaviour when ready, including sending or receiving signals or invoking another activity. Object nodes explicitly hold objects that arrive in their incoming edges and offers them to the outgoing edges. Control nodes organise the order flows are traversed. They act as “traffic switches" across the activity edges. Nodes and edges can be grouped in swimlanes (or partitions), used to organise parts of flows. They can be vertical or horizontal, and their primary purpose is to delimit boundaries of responsibility for each group of behaviour. Figure 2.4 shows all types of control nodes, some of the main types of action nodes and object nodes grouped by swimlanes. The descriptive semantics for each constructor can be seen in the UML specification [26].

Besides the semantics of each node, the execution semantics of an activity diagram is described in terms of tokens flowing through the edges and nodes. Activity edges are directed with tokens flowing from the source activity node to the target activity node. However, the token must only flow if the target is ready to accept it. Some nodes may generate tokens. For instance,

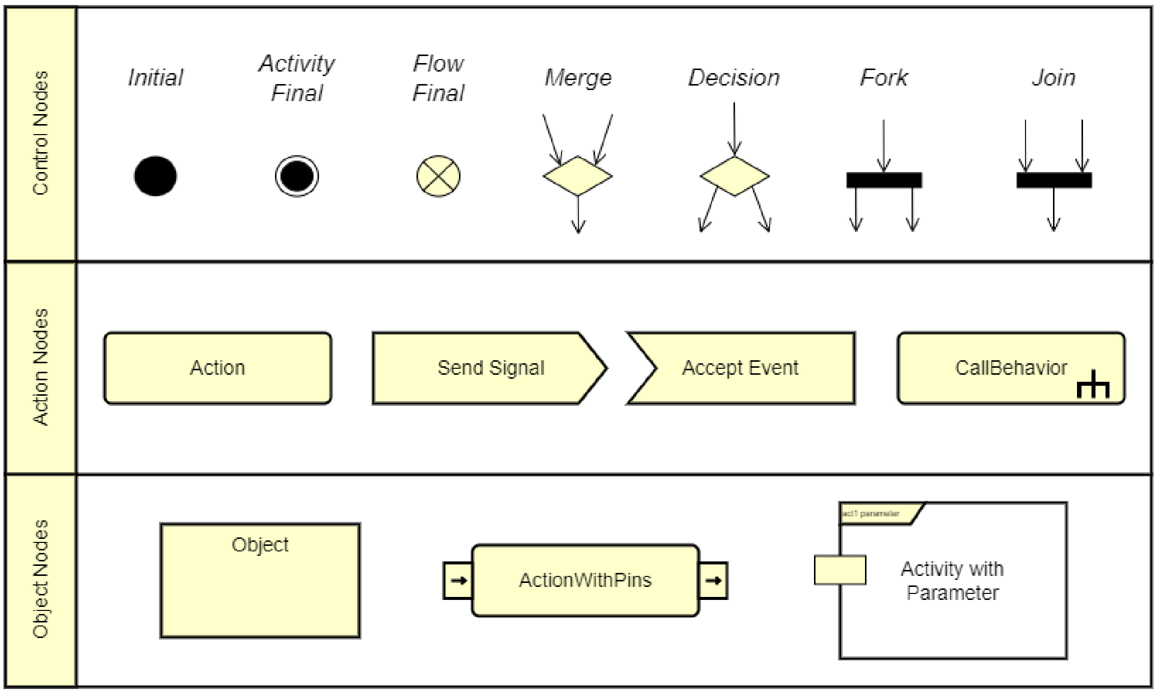


Figure 2.4: Activity diagram nodes.

an initial node creates tokens on its outgoing edges when the activity starts. Other nodes only consume tokens, like flow final and activity final nodes. An action node can only be executed once all incoming edges are offering tokens, and when it terminates, it must offer tokens in its outgoing edges.

Finally, an activity diagram can only terminate in two scenarios: if no active tokens are flowing through the activity after it has been started, or if an activity final node has consumed a token. In the latter case, all current flows are halted.

Sequence diagrams are another UML model element widely used to describe dynamics focusing on interactions. One of the reasons for the acceptability of such a notation is its readability, which facilitates communication between stakeholders. They have been used to represent use cases, test purposes, business scenarios, etc. Since it presents an easy-to-read way to show traces (in terms of message events), we have chosen it as the representation for counterexamples.

Sequence diagram focus on the message interchange between several lifelines, the lifespans of objects depicted in the diagram. A message flows between two lifelines and is composed of a sending event and a receiving event. In addition, it can be synchronous (filled arrowhead) or asynchronous (open arrowhead). Figure 2.5 illustrates a simple interaction between a User, its phone and the message server. It has three lifelines, User (depicted as an actor), userPhone and MsgServer. User sends an asynchronous message submitText() to userPhone and MsgServer also sends an asynchronous message to userPhone. According to the UML semantics for sequence diagram, in this example, the sending events of these messages can happen in any order, however, the lifeline of userPhone restricts that the submitText() receiving event happens before the reception the sendMsgId() event.

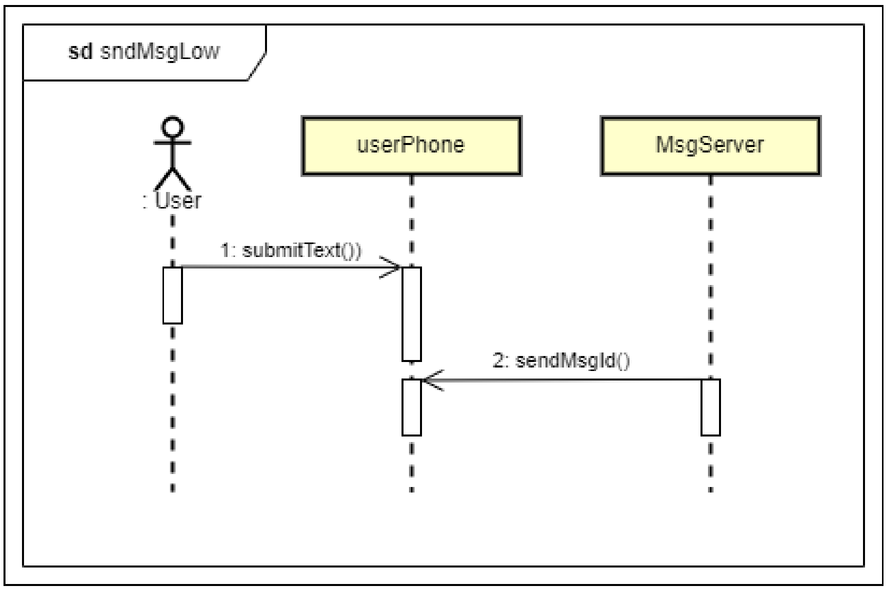


Figure 2.5: Example of a sequence diagram.

There are additional constructors that allow the specification of complex scenarios, however, this subset is enough to contextualise our strategy. More details can be seen in [26].

2.3 CSP notation

The CSP process algebra is very expressive to specify systems composed of interacting components. In CSP, a process is the basic unit for describing behaviour. It is defined in terms of events and other processes. The function $\alpha(P)$ yields the alphabet of a process P, that is, the events that the process P may communicate. The primitive process $SKIP$ represents successful termination. The process $a \rightarrow P$ offers the event a to the environment and then behaves as the process P. CSP channels abstract a set of events with a common prefix. The syntax $c?x : A$ represents the channel c inputs a value x, such that $x \in A$, which is the type for the channel c. The value for x is chosen by the environment. The syntax $c.e$ ($c!e$) represents an expression $e \in A$ is communicated through the channel c. The difference between . and ! happens when the channel communicates more than one value at once. For instance, consider c is a channel that communicates simultaneously four different values, and has type $A.B.C.D$. The communication $c?x.y!z.t$ is equivalent to $c?x?y!z!t$, because a dot following a ? (!) is taken to be part of a pattern that is matched by the input. However, for this to be true, we need the variables z e t to be in the context. The sequential composition $P1;\ P2$ behaves like the process $P1$ and, provided it terminates successfully, $P2$ takes over. The CSP notation has no explicit operator for recursion, but it allows one to use the name of the process in its definition. For instance, the process $P = a \rightarrow P$ communicates the event a and then behaves as P.

The external choice $P1 \square P2$ initially offers events of both processes $P1$ and $P2$. The communication of the first event resolves the choice in favour of the process that performs it. The parallel composition $P1 \, |[cs]| \, P2$ synchronise $P1$ and $P2$ on the events in the set cs; events not in cs occur independently. Processes composed in interleaving $P1 \; ||| \; P2$ progress in parallel without synchronisation. The event hiding operator $P \setminus cs$ internalises the events that belong to the set cs, which become no longer visible to the environment. The interruption operator ($\triangle$) allows a process to be interrupted by another. The process $P \triangle Q$ behaves as P until Q communicates an event. When this happens, we say that P has been interrupted by Q.

The traces of a process P, say $traces(P)$, is the set of possible sequences of events performed by P. We can compare the traces of two processes using a refinement assertion denoted by $P \sqsubseteq_T Q$. Such an assertion holds if, and only if, $traces(Q) \subseteq traces(P)$. The FDR tool [8] verifies process refinement as well as classical properties like deadlock and nondeterminism.

3 Diagrammatic Language for properties

The visual language to specify properties of RoboChart models allows the definition of the expected order for inputs, outputs and operation calls of RoboChart components, which can be robotic controllers, modules and state machines. Properties defined with this language are used to define the possible behaviours a component can perform. A property is valid if the the behaviour of the component is a subset of the behaviour of the property. Consider *P_Prop* is the CSP process that represents the semantics of a visual property and P_CName is the CSP process of a component named CName. The property Prop holds in the RoboChart component CName if, and only if, the traces for a component is a subset of the traces of the property. Formally.

$$P\_Prop \sqsubseteq_T P\_CName$$

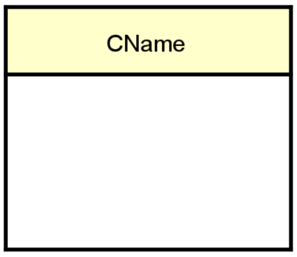
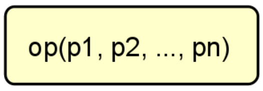
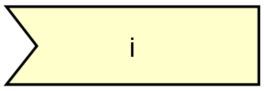
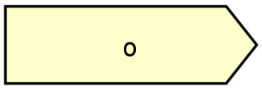
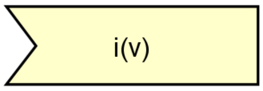
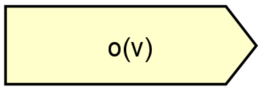
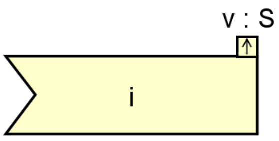
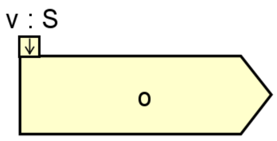
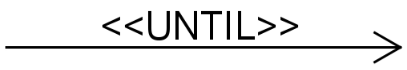
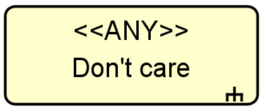
The remaining of this chapter presents the syntax and the semantics of the proposed language.

3.1 Language syntax

The language for properties is based on the notation of UML activity diagrams. Standard activity diagram nodes (Figure 2.4) can be mixed with specialised nodes and stereotypes to specify the expected flow of observations for a RoboChart component. Table 3.1 shows the notation of the proposed language.

First row in Table 3.1 shows the diagram that represents a RoboChart property has a unique swimlane with the same name of the RoboChart component, for illustrative purposes, we use the nomenclature CName which means Component Name (state machine, controller or module). Inside the swimlane are the nodes that specify the property. A call to a RoboChart operation *op* with parameters *p1,...,pn* is represented as an action in the property language (Row 2). Furthermore, a RoboChart input event *i* that does not communicate values is represented in the property language as an accept event action, as seen in Row 3. On the other hand, when it is an output event *o*, then we use a send signal action (Row 4). If the input or output communicates a value *v*, such a value is put between parenthesis — Rows 5 and 6. The property language

Table 3.1: Property notation

N°	Notation	Description
1		Swimlane with the nodes of the property
2		Operation op with parameters p1,...,pn
3		Simple input i
4		Simple output o
5		Input i with value v
6		Output o with value v
7		Input i with value v within S
8		Output o with value v within S
9		Edge with <<UNTIL>> stereotype
10		Call behaviour with <<ANY>> stereotype

allows the specification of the range of values for inputs and outputs using pins. Consider that S is a set of values that is a subset of the values for the event (input i or output o) defined in the RoboChart model. Rows 7 and 8 of Table 3.1 show how to specify a range of values for inputs and outputs through the usage of pins.

The stereotype <<UNTIL>> labels edges that target a send signal, an accept event or an operation — Row 9. This stereotype specifies that the diagram performs any possible sequence of observations in the RoboChart component between the source node and the target node except the target node event/operation. If the diagram executes the target node event/operation, the diagram follows the outgoing edge of the target node. The stereotype <<ANY>> labels call behaviour actions — Row 10. When the diagram executes a call behaviour with this stereotype, any sequence of observation in the context of the component can be performed by the diagram.

There is an important difference in the meaning of a signal sending and receipt in the property language and the original meaning in the activity diagram. The original meaning is that a send signal action may transmit a signal that could be received by an accept event action. In the proposed property language these nodes do not exchange signal/events. This is more explicit in the next section that presents the semantics for the language.

Considering the combination of the standard nodes for activity diagram and the introduced abstraction patterns, it is possible to define a range of properties using the proposed notation. We illustrate the usage of the notation with examples.

Figure 3.1 illustrates a property, say Angular, for the SMovement state machine in Figure 2.3. This property specifies that whenever the robot receives an obstacle event, it turns to avoid a collision with a constant angular speed. We explain how this property is expressed using the proposed notation. The diagram starts with a merge node with a control edge with the stereotype <<UNTIL>>. This edge targets an accept event node that represents the reception of the obstacle event. As explained, this stereotype specifies that any sequence of observations can happen in the RoboChart model before the obstacle event. After the obstacle, the move operation must be called with the first parameter equals 0 (linear speed) and the second parameter equals the constant av (angular speed).

Chapter 5 presents a more comprehensive use of the notation for modelling properties of two different systems designed with RoboChart.

3.2 Language semantics

This section presents the CSP semantics for the property language for RoboChart models.

Figure 3.2 shows the overall structure of the CSP process that specifies a property. In this figure, boxes represent CSP processes, arrows events and vertical bars the parallel composition of processes. Synchronisation alphabet appears inside brackets. This semantics is an adaptation to represent RoboChart properties of the compositional semantics in [14] for activities. The process *PROP* represents a property. It is the parallel composition between the activity process,

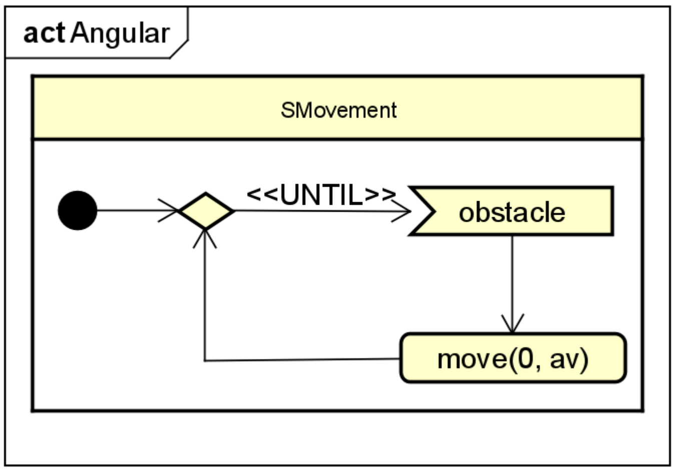


Figure 3.1: Property for SimFW.

say *ACTIVITY*, and the process *AUXILIARY*. The semantics for the activity is the parallel composition among the process that specify each of the activity nodes, say *Nodes*, with the process that manages the number of active tokens in the activity, say *TokenManager*. The semantics for *AUXILIARY* is the composition of auxiliary processes used in the specification of the stereotypes <<UNTIL>> and <<ANY>> introduced in Table 3.1. The visible events of the process *PROP* represent events and operations for a RoboChart model, other events are internal to the specification and used to control the composition between the processes.

We formalise the CSP semantics for a property. Let α_C the set of control events of the process *ACTIVITY*. Such a process is interrupted by the process $endDiagram \rightarrow SKIP$. The event *endDiagram* is communicated by the process *ACTIVITY* to indicate the conclusion of the activity. The composition synchronises on control events that belong to the channels *begin*, *end*, *chaos* and *endDiagram*. The first three channels are used to specify the behaviour of the stereotypes, the last channel belongs to α_C. The channels *begin* and *end* are in the synchronisation set if the stereotype <<UNTIL>> is used in the definition of the property. The channel *chaos* is in the synchronisation set if the stereotype <<ANY>> is used. This is the CSP expression for the process *PROP*.

$$\begin{array}{lll} PROP = & (& ACTIVITY \\ & & \quad [| \mid begin, end, chaos, endDiagram \mid |] \\ & & (||| \; P : AUXILIARY \bullet P \triangle endDiagram \rightarrow SKIP) \\ &) & \setminus \alpha_C \cup \{| \; begin, end, chaos \; |\} \end{array}$$

The process *ACTIVITY* represents the behaviour of an activity adapted from [14] to

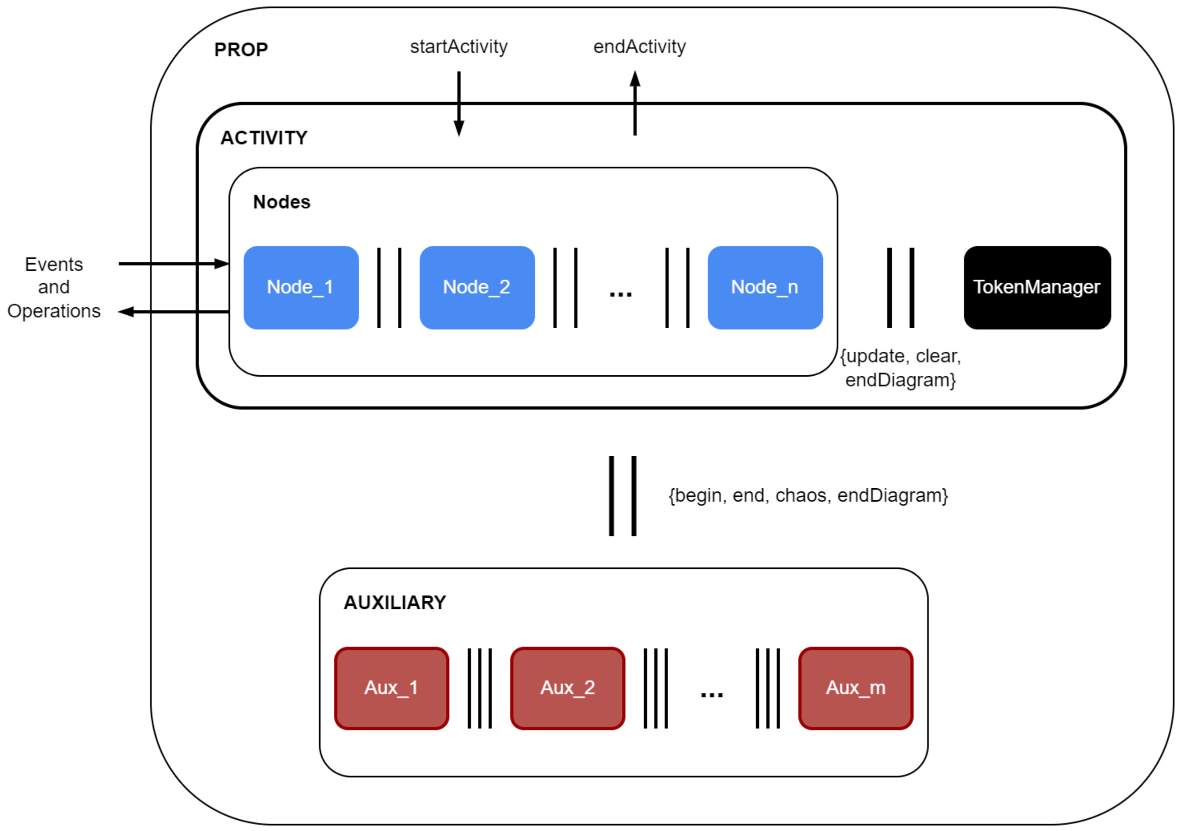


Figure 3.2: Property semantics in CSP.

represent RoboChart observations as inputs, outputs and operation calls.

$$\begin{aligned} \mathit{ACTIVITY} = \quad & (\mathit{startActivity} \to \mathit{SKIP};\ \mathit{Nodes};\ \mathit{endActivity} \to \mathit{SKIP}) \\ & [|\ \mathit{update}, \mathit{clear}, \mathit{endDiagram}\ |] \\ & \mathit{TokenManager} \end{aligned}$$

The behaviour of *ACTIVITY* is the parallel composition of the nodes and the *TokenManager* process. An activity starts when the *startActivity* event is communicated, then it behaves as the process *Nodes*. The *startActivity* event may receive input data required by the activity parameter nodes. As soon as an activity terminates, it communicates the *endActivity* event with the data available in its output parameter nodes. The underlying semantics of activity diagrams is described by the flow of tokens among nodes. An activity terminates once there is no active token, or a token reaches an activity final node. In our semantics, the *TokenManager* process records the number of active tokens of an activity and controls the termination of the diagram. This process is shown next.

$$\begin{aligned} \mathit{TokenManager}(n, \mathit{init}) = \quad & \mathit{update}?x \to \mathit{TokenManager}(n + x, \mathit{true}) \\ & \Box\ \ \mathit{clear} \to \mathit{endDiagram} \to \mathit{SKIP} \\ & \Box\ \ (n == 0\ \ \land\ \ \mathit{init})\ \ \&\ \ \mathit{endDiagram} \to \mathit{SKIP} \end{aligned}$$

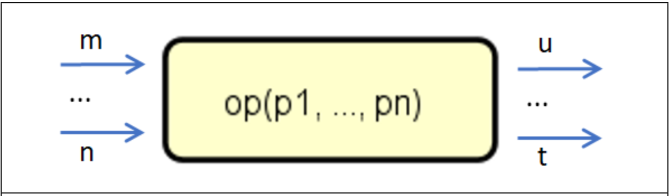
The initial values of the parameters *n* and *init* are 0 (zero) and false, respectively. The *TokenManager* process can receive communications on the channel *update* with an integer value x to update the current number of tokens to n + x (x can be either positive or negative). It is positive when the node creates new flows and negative if the node join flows. After the first update event, the token manager becomes active, so the value of *init* is set to true. An activity final node terminates the activity once a token has reached it. In our semantics it is the only node that communicates the event *clear*, which synchronizes with the *TokenManager* process. Once it has been synchronized, the flows of the diagram terminate (event *endDiagram*). Another possibility of termination is when the number of tokens reaches 0 (zero) and the diagram has already started (init is true). In this situation, the flow of the diagram terminates as well. After the event *endDiagram* is communicated, all CSP processes of the nodes are interrupted and the *endActivity* event is performed.

According to the UML semantics, an activity is described in terms of nodes and edges between the nodes. In our CSP semantics, these elements are represented by processes and events, respectively. The process *Nodes* is the parallel composition of the processes that represent each node. Node processes synchronise on the events that represent edges. The outgoing edge of a node is the incoming edge of another node. Control edges are represented by events in the form $ce.id$ and object edges by events in the form $oe.id$, such that id is the edge identifier. For instance, if there is a control edge $ce.1$ between two nodes $N1$ and $N2$, the event $ce.1$ is part of the alphabet of the processes of both nodes. Events in the form $ce.id$ and $oe.id$ belong to α_C, and are not visible because events in α_C are hidden in the process *PROP*.

In what follows, we present the semantics for the nodes whose semantics is particular to the properties language. Nodes that are not presented here keep the semantics presented in [14].

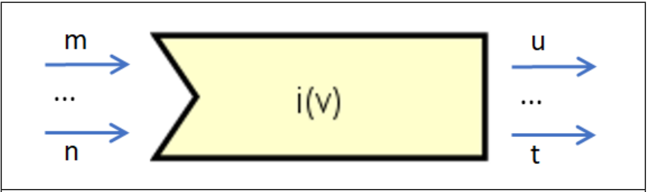
We start showing the semantics for nodes that represent an operation call, a input and an output (Rows 2—6 in Table 3.1). Tables 3.2, 3.3 and 3.4 represent the semantics for such nodes. Let $\{m..n\}$ be the range of indices for the incoming edges of the node, and $\{u..t\}$ the range of indices for the outgoing edges. The processes *P_op*, *P_i* and *P_o* represent the formal semantics for such nodes. The semantics of these nodes is to wait for the communication of the incoming control edges, communicate an event, wait for the communication of the outgoing control edges and to recurse. The wait for the incoming (outgoing) edges is specified as the interleaving of the communication of the events for the edges followed by a successful termination. In CSP, the parallel composition only terminates if all the processes in the composition do terminate. The event communicated between incoming and outgoing edges has a particular format for each kind of node. In Table 3.2, the event $opCall.p1...pn$ represents an operation op with parameters p1,...,pn. In Table 3.3, the CSP event $i.in.v$ represents a RoboChart input event named i with value v. Table 3.4 presents the opposite scenario when the model communicates an output event. In this case, the CSP event $o.out.v$ relates to a RoboChart output event o with value v. The CSP specification for a RoboChart model is organised in CSP modules. CSP events prefixed with *CName* :: are those defined in a CSP module with the name *CName* ::. For instance, *CName* :: *opCall*.*p*1...*pn* denotes *opCall*.*p*1...*pn* is defined in the module *CName*.

Table 3.2: Operation call - CSP Semantics



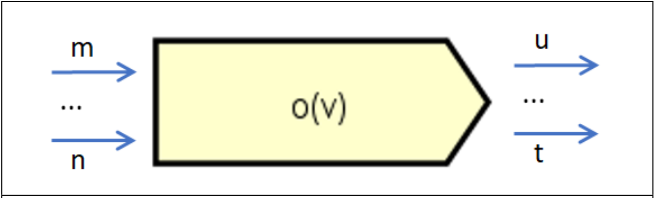
$$P\_op = \begin{array}{l}(ce.m \rightarrow SKIP \ ||| \ ... \ ||| \ ce.n \rightarrow SKIP);\\ CName :: opCall.p1...pn \rightarrow SKIP;\\ (ce.u \rightarrow SKIP \ ||| \ ... \ ||| \ ce.t \rightarrow SKIP);\\ P\_op\end{array}$$

Table 3.3: Input with value - CSP Semantics



$$P\_i = \begin{array}{l}(ce.m \rightarrow SKIP \ ||| \ ... \ ||| \ ce.n \rightarrow SKIP);\\ CName :: i.in.v \rightarrow SKIP;\\ (ce.u \rightarrow SKIP \ ||| \ ... \ ||| \ ce.t \rightarrow SKIP);\\ P\_i\end{array}$$

Table 3.4: Output with value - CSP Semantics

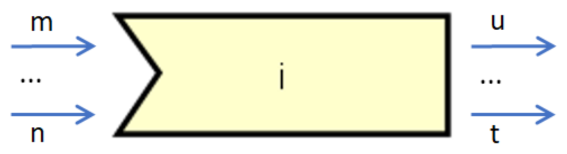


$$P\_o = \begin{array}{l}(ce.m \rightarrow SKIP \ ||| \ ... \ ||| \ ce.n \rightarrow SKIP);\\ CName :: o.out.v \rightarrow SKIP;\\ (ce.u \rightarrow SKIP \ ||| \ ... \ ||| \ ce.t \rightarrow SKIP);\\ P\_o\end{array}$$

The semantics for a simple input, see Row 3 in Table 3.1, is presented in Table 3.5. Such a semantics is very similar to that for an input with value. The difference is the event that follows after the component identification (*CName* ::). Consider a simple input (output) named i, the CSP event that represents a simple input is *i*.*in*. Similarly, a simple output (Row 4 in Table 3.1) is depicted in Table 3.6. The simple output event named (o) is translated to a CSP event *o*.*out*.

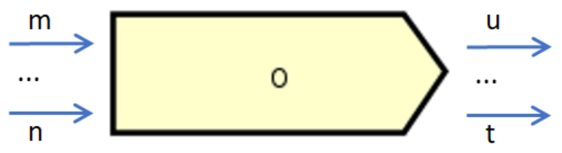
The semantics for nodes that specify an input and an output with a range of values (Table 3.1 lines 7 and 8) are specified by the process *P_ir* and *P_or* presented in Table 3.7 and Table 3.8, respectively. The semantics for *P_ir* is to wait for the communication of the

Table 3.5: Simple input - CSP Semantics



$P\_si =$	$(ce.m \rightarrow SKIP \; \vert\vert\vert \; ... \; \vert\vert\vert \; ce.n \rightarrow SKIP);$
	$CName :: i.in \rightarrow SKIP;$
	$(ce.u \rightarrow SKIP \; \vert\vert\vert \; ... \; \vert\vert\vert \; ce.t \rightarrow SKIP);$
	P_si

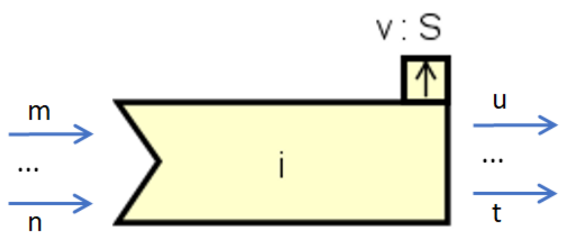
Table 3.6: Simple output - CSP Semantics



$P\_so =$	$(ce.m \rightarrow SKIP \; \vert\vert\vert \; ... \; \vert\vert\vert \; ce.n \rightarrow SKIP);$
	$CName :: o.out \rightarrow SKIP;$
	$(ce.u \rightarrow SKIP \; \vert\vert\vert \; ... \; \vert\vert\vert \; ce.t \rightarrow SKIP);$
	P_so

incoming object edges, communicate the event $CName :: i.in?v$, wait for the communication of the outgoing object edges and to recurse. The semantics for P_or has similar steps but a different event $CName :: o.out?v$.

Table 3.7: Input with range - CSP Semantics

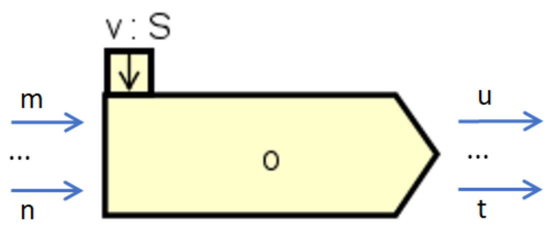


$P\_ir =$	$(ce.m \rightarrow SKIP \; \vert\vert\vert \; ... \; \vert\vert\vert \; ce.n \rightarrow SKIP);$
	$CName :: i.in?v : S \rightarrow SKIP;$
	$(ce.u \rightarrow SKIP \; \vert\vert\vert \; ... \; \vert\vert\vert \; ce.t \rightarrow SKIP);$
	P_ir

Next, we discuss the semantics for nodes reached by an edge with the stereotype <<UNTIL>> and for a call behaviour node with the stereotype <<ANY>>. These two kinds of nodes synchronise with the process *AUXILIARY* (recall the definition of *PROP*) in the events of the channels *begin*, *end* and *chaos*.

Nodes reached by an edge with the stereotype <<UNTIL>> communicate events *begin.id* and *end.id*, and call behaviour nodes with the stereotype <<ANY>> communicate

Table 3.8: Output with range - CSP Semantics


$P\_or = (ce.m \to SKIP \mid\mid\mid ... \mid\mid\mid ce.n \to SKIP);$ $CName :: o.out?v : S \to SKIP;$ $(ce.u \to SKIP \mid\mid\mid ... \mid\mid\mid ce.t \to SKIP);$ P_or

chaos.id events. The value for *id* is unique for each usage of <<UNTIL>> and <<ANY>>. Let *A_PROCESSES* be the set of auxiliary processes. Moreover, let the expression $\big|\big|\big| P : S \bullet P$ be equivalent to the process $P_1 \mid\mid\mid \ldots \mid\mid\mid P_n$, for $S = \{P_1, \ldots, P_n\}$. The semantics for *AUXILIARY* is the interleaving of the processes in *A_PROCESSES* interrupted by the event *endDiagram*.

$$AUXILIARY = (\big|\big|\big| A : A\_PROCESSES \bullet A) \triangle endDiagram \to SKIP$$

The definition for an auxiliary process depends on the stereotype. We show the semantics for auxiliary processes in what follows.

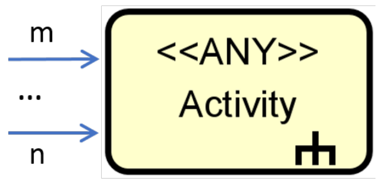
The process *CallBehaviour* in Table 3.9 formalises the semantics for a call behaviour node with the <<ANY>> stereotype (Row 10 in Table 3.1). Such a process waits for the communication of the incoming edges, then communicates *chaos.id* and behave as *CallBehaviour*.

Consider $\Box\, ev : S \bullet ev \to P$ is equivalent to the choice $ev_1 \to P \,\Box\, \ldots \,\Box\, ev_k \to P$, for $S = \{ev_1, \ldots, ev_k\}$. The process $RUN(S) = \Box\, ev : S \bullet ev \to RUN(S)$ is the CSP process that produces the traces in S^*. This process is used in the definition of the auxiliary process. Moreover, let *P_CName* be the CSP process that formalises the untimed semantics of RoboChart component CName. The process $AUX\_ANY = chaos.id \to RUN(\alpha(P\_CName))$ belongs to *A_PROCESSES* and represents the auxiliary process for a node with the <<ANY>> stereotype. Remember in the expression for the process *PROP*, *AUX_ANY* synchronises with the process *CallBehaviour* in the *chaos.id* event and behaves as $RUN(\alpha(P\_CName))$. This last process contains all traces that can be produced by *P_CName*. This semantics means that any sequence of events after the call behaviour stereotyped by <<ANY>> are valid behaviour.

The semantics for a node that is reached by an edge with the <<UNTIL>> stereotype (Row 9 in Table 3.1) is formalised by the process *Until*[1] presented in Table 3.10. Only three kinds of nodes can be reached by an edge with <<UNTIL>>: send signal (output), an accept event (input) or an action (operation). The semantics for such a node is to wait for the communication

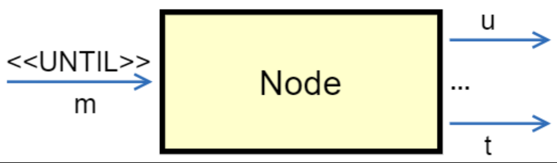
[1]The semantics for the <<UNTIL>> stereotype is not equivalent to the until operator used in temporal logics. The expressiveness of CSP refinement and its relation to temporal logics is reported in [30, 16].

Table 3.9: Call behaviour with the <<ANY>> stereotype — CSP Semantics

	
$CallBehaviour =$	$(ce.m \rightarrow SKIP \mathbin{\vert\vert\vert} \ldots \mathbin{\vert\vert\vert} ce.n \rightarrow SKIP);$ $chaos.id \rightarrow SKIP;$ $CallBehaviour$

of the event for the incoming edge (whose stereotype is <<UNTIL>>), to communicate *begin.id* and *end.id* events, wait for the communication of the outgoing edges, and to behave as *Until*.

Table 3.10: Node reached by an edge with <<UNTIL>> — CSP Semantics

	
$Until =$	$(ce.m \rightarrow SKIP);$ $begin.id \rightarrow end.id \rightarrow SKIP;$ $(ce.u \rightarrow SKIP \mathbin{\vert\vert\vert} \ldots \mathbin{\vert\vert\vert} ce.t \rightarrow SKIP);\ Until$

The process *WAIT* is used in the definition of the auxiliary process for *Until*. Consider the CSP process $Recurse(S,P) = \Box\, evt : S \bullet evt \rightarrow P$ that offers a choice of all events in the set S; after the process communicates the event in the choice it behaves as P. The process *WAIT* is parametrised by the event *ev*. The behaviour of this process is to communicate the events of the alphabet of *P_CName* except *ev*, and to behave as $WAIT(ev)$. If the event *ev* is communicated, the process terminates.

$$
\begin{aligned}
WAIT(ev) = \;& Recurse(\alpha(P\_CName) \setminus \{ev\}, WAIT(ev)) \\
& \Box \\
& ev \rightarrow SKIP
\end{aligned}
$$

The process *AUX_UNTIL* belongs to *A_PROCESSES* and formalises the auxiliary process for *Until*. The events *being.id* and *end.id* of this process synchronise with the events in the *Until* process depicted in Table 3.10. In between these events, it behaves as $WAIT(CName :: event)$. The content for *event* follows the semantics for an output, an input or an operation node. The behaviour of the parallel composition of *AUX_UNTIL* with *Until* is to accept any event in $\alpha(P\_CName)$ except $CName :: event$. After $CName :: event$ is communicated, the composition follows one of the outgoing edges.

$$\begin{aligned} AUX\_UNTIL = \quad & begin.id \rightarrow WAIT(CName :: event); \\ & end.id \rightarrow AUX\_UNTIL \end{aligned}$$

The semantics for a node of any kind has an implicit interruption with the *endDiagram*. The effect of this interruption is that the behaviour of all nodes are interrupted by such an event, whenever the *TokenManager* process communicates this event. This is kept implicit to simplify the presentation of the semantics.

Consider the function *compose* that follows. This function is used to specify the composition of a sequence of nodes. The first function parameter is a sequence of process identifiers, and the second is the set of events that have already been used in the synchronisation of the parallel composition. Let α_{id} be the alphabet of control events of a node identified by *id*. Such a function uses generalised parallel composition to compose the nodes of a property. Synchronisation set of the parallel composition contains the control alphabet of the node to be composed, and the *endDiagram* event that allows the nodes to terminate together. The synchronisation set excludes the events already used in the composition of previous nodes of the network of processes formed by the already composed nodes.

$$\begin{aligned} compose(\langle id \rangle, \_) \quad &= \quad Node_{id} \\ compose(\langle id \rangle \frown tail, past) \quad &= \quad Node_{id} \\ & \qquad |[(\alpha_{id} \setminus past) \cup \{endDiagram\})]| \\ & \quad\;\; compose(tail, \alpha_{id} \cup past) \end{aligned}$$

The formal definition for the process *Nodes* is

$$Nodes = compose(seq(Nodes\_IDs), \{\})$$

In such a definition *Nodes_IDs* represents the set of identifiers for processes that formalise nodes and *seq* is a function that converts a set into a sequence.

The order of nodes returned by the function *seq* is arbitrary; however, the order does not change the semantics of the composition. This holds since the generalised parallel composition operator is associative with different synchronisation alphabets if the synchronisation set between the processes contain the intersection of the alphabets of the processes (refer to [29] for the laws of parallel composition operators).

To illustrate our semantics, we show the process *P_Angular* that captures the semantics of the property Angular in Figure 3.1. Appendix A shows the complete CSP semantics for the property Angular. For conciseness, instead of presenting the syntax of the process obtained by the compositional semantics, we present an equivalent process that has a shorter representation. Let *P_SMovement* be the CSP process that formalises the behaviour of the *SMovement* state machine.

The behaviour of *P_Angular* is to recurse if an event different from obstacle is communicated. When an obstacle event happens, then it communicates $SMovement :: moveCall.0.av$. In the sequence it behaves as *P_Angular*.

$$\begin{array}{l} P\_Angular = \\ \quad Recurse(\alpha(P\_SMovement) - \{SMovement :: obstacle.in\},\ P\_Angular) \\ \quad \Box \\ \quad SMovement :: obstacle.in \rightarrow SMovement :: moveCall.0.av \rightarrow\ P\_Angular \end{array}$$

The property Angular in SMovement holds if, and only if, the following refinement holds.

$$P\_Angular \sqsubseteq_T P\_SMovement$$

This refinement holds since the property holds in the SMovement state machine model.

To illustrate a model where the property does not hold, suppose the state Turning of the machine SMovement (Figure 3.1) performs a call to the operation move(av,0) — this change simulates a specification error in the RoboChart model. Checking such a refinement using the FDR tool, we have that the refinement does not hold and FDR yields the counterexample trace.

$$\langle SMovement :: move(lv,0), SMovement :: obstacle.in, SMovement :: move(av,0)\rangle$$

This counterexample reveals that after communicating an obstacle input event, the SMovement state machine does perform a call to the move operation with an unexpected value for the parameters, as we switched the order of the parameters of this call in the SMovement state machine model.

Also, we used FDR to verify whether the semantics of properties using the proposed language equals the semantics of the original property specified as CSP processes. For instance, we were able to verify the corresponding CSP specification generated from the visual specification of the Angular property presented in Figure 3.1 is equivalent to the simplified property written in CSP.

4 Tool Support

This chapter focuses on describing how the tool support is designed, its dependencies, and features.

4.1 Architecture

Our tool has been implemented as a plug-in for the Astah UML modelling environment, which can be extended by the integration of plug-ins to add new features. It has been built based on another plug-in that verifies properties on activity diagrams [14]. Figure 4.1 shows the architecture of our plug-in. Astah runs on top of the Java Virtual Machine (JVM) to enable the tool to be platform independent. We use the Astah Java Application Programming Interface (API) to programmatically read activity diagrams as properties, translate them to CSP and verify properties on RoboChart models using the integration built with FDR. Also, we provide a traceability mechanism when a property does not hold, generating a counterexample as a sequence diagram.

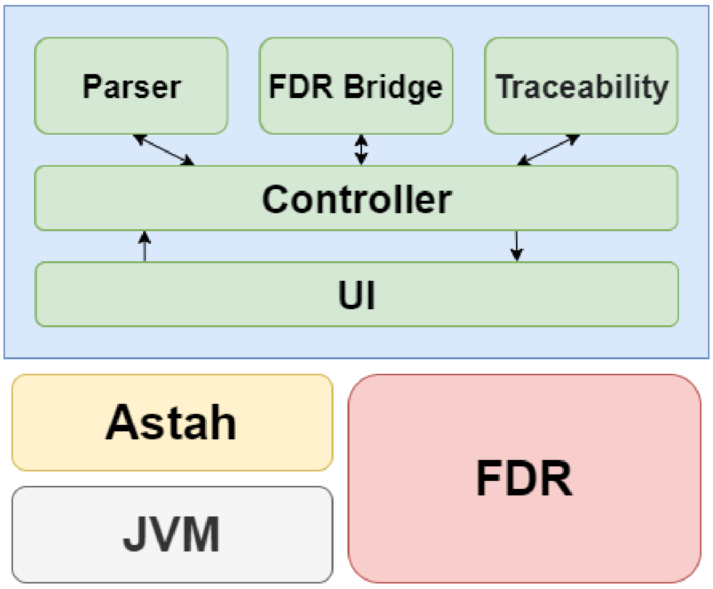


Figure 4.1: Plug-in Architecture.

The developed plug-in is divided into five modules, which are:User Interface (UI), Controller, Parser, FDR Bridge and Traceability. The UI module is responsible for making the connection between the user and the controller through the plug-in menu. For instance, to check the property of a given RoboChart model, the user must provide the path to the RoboChart file. This can be set by accessing the plug-in UI menu available at `Tools -> Properties Plug-in Configuration -> Robochart Location`, as can be seen in Figure 4.2.

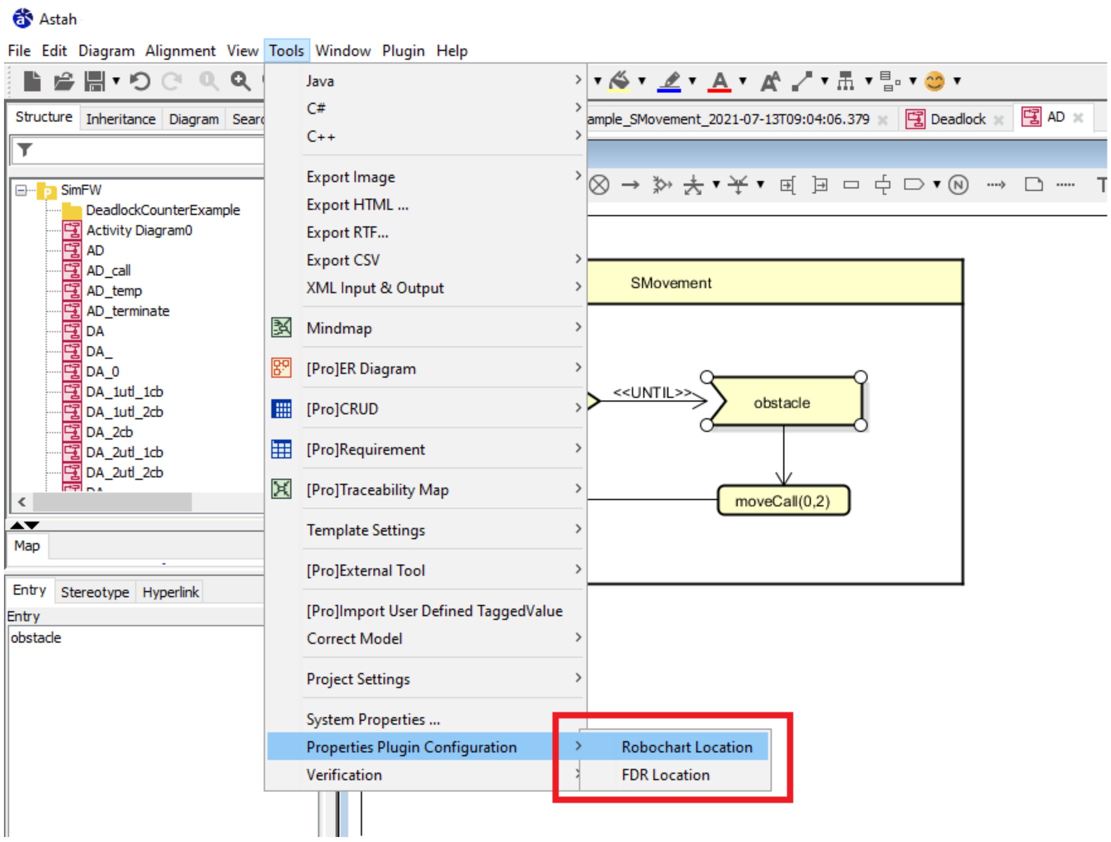


Figure 4.2: Menu for setting the path to the RoboChart file.

The Controller module is responsible for receiving information (commands and diagrams) from the UI module, managing the entire plug-in operation, and returning a response (messages and/or diagrams) to the UI module. The Parser module is responsible for receiving a diagram from the Controller module, translating it according to the semantics described in Section 3.2, and returning a CSP file to the Controller module.

The FDR Bridge module is responsible for communicating with FDR. It invokes the assertions specified in the CSP file generated by the parser. When the assertion does not hold, it is also responsible for collecting the counterexample returned by FDR (list of events) and returning it to the Controller. The Traceability module is responsible for receiving an event list (trace) of the Controller module and providing a diagrammatic view of it in terms of a sequence diagram.

4.2 Parser module

The parser module implements the transformation from the visual property to the CSP semantics for the property showed in Figure 1.2. The process of translating the diagram is done automatically by the plug-in. The translation rules are encoded directly in the Java program we have built. The coding task followed a Test-Driven Development approach [3] where first we define the test cases describing how the translation should be, and, next, we implement the parser code.

The architecture of the parser follows the same structure as defined in [14], however, we had to extend the translation functions to cover the new elements proposed for defining RoboChart properties: proposed stereotypes, swimlanes and the RoboChart data. Regarding the latter, in order to analyse a RoboChart model, we link the CSP specification we generate for the property to the the RoboChart file provided by the user in the corresponding menu. In Figure 1.2, the plus symbol illustrates the link of the property specification in CSP with the CSP semantics for the RoboChart model. The union of the two specification is the input for the automatic verification performed by FDR.

4.3 FDR Bridge module

FDR is a powerful model checker for CSP specifications. It generates a Labelled Transition System (LTS) [29] from the CSP specification and traverses all possible states to establish if a refinement expression holds.

This module is responsible for the communication with FDR. It invokes the assertions specified in the CSP file generated by the parser. When the assertion does not hold, it is also responsible for collecting the counterexample returned by FDR (list of events) and returning it to the Controller. In order to make this integration possible, the user has to inform the path to the FDR installation folder. This can be performed by accessing the plug-in UI menu `Tools -> Properties Plugin Configuration -> FDR Location`, as can be seen in Figure 4.2.

After informing the path, whenever a RoboChart property is verified, the plug-in (FDR Bridge module) loads the FDR library given the provided path to the FDR installation folder. This is possible because we use the Java Reflection technique, which allows us to load the FDR API dynamically. Therefore, we do not need to include FDR as part of the plug-in.

4.4 Robochart property analysis

After modelling the activity diagram as a property and providing the path to the RoboChart file, the user can start the verification process in the menu we have created in Astah as can be seen in Figure 4.3. This action triggers the following tasks: the tool generates the corresponding

CSP specification (Parser module), loads it in FDR and invokes the assertion that checks if the property is valid in the RoboChart model (FDR Bridge module). If the property does not hold, FDR returns a counterexample trace displaying the sequence of events that led to the violation, which is, then, translated to a sequence diagram. This way the user does not need to read the counterexample trace in the CSP notation.

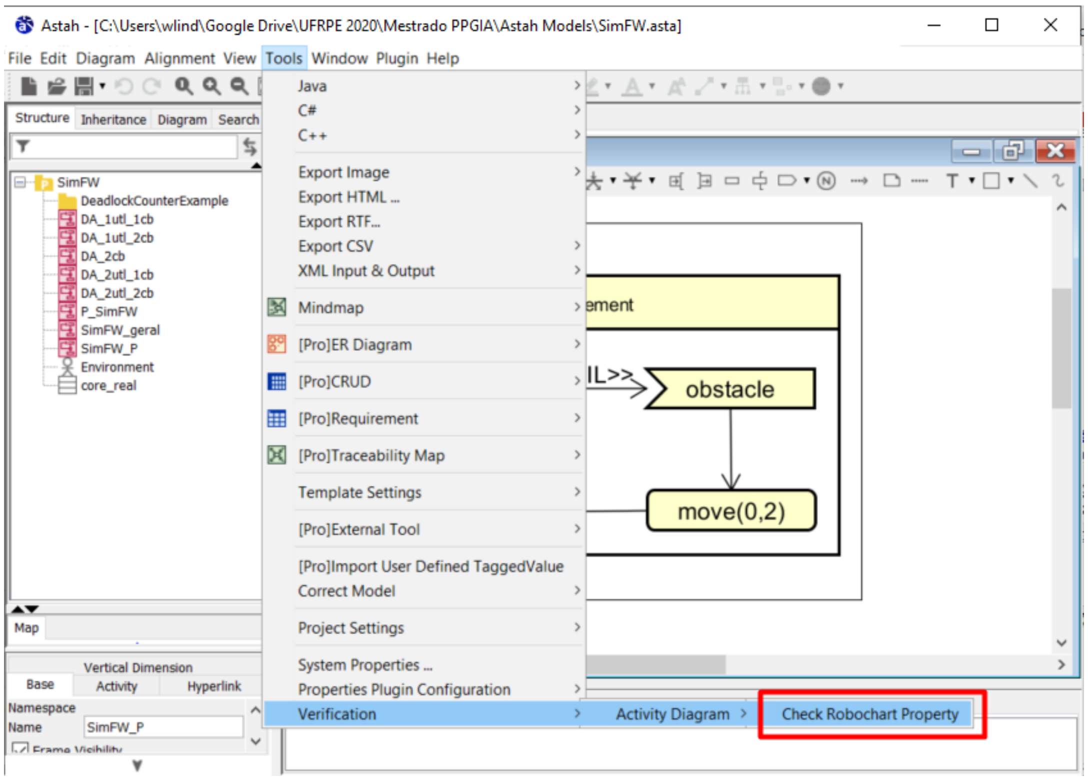


Figure 4.3: Checking a RoboChart property in Astah.

4.5 Traceability

The trace returned by FDR when the modelled property does not hold, as previously mentioned, is an ordered list of events that shows us the path to point that violates the property. This ordered list is crucial to identify where the problem occurred and to create a diagrammatic counterexample that shows the issue in a view friendly to the user. For example, consider the trace

$$\langle SMovement :: move(lv,0), SMovement :: obstacle.in, SMovement :: move(av,0) \rangle$$

, already introduced in Section 3.2, that is yielded by FDR as the counterexample for the Angular property. Figure 4.4 shows the traceability in terms of a sequence diagram that illustrates the events in the counterexample. In this figure, the constants lv and av are replaced by the values 1 and 2, respectively. Such values are the concrete values for the constants, which are defined in the SimFW model before the verification performed by FDR. The actor Environment represents the elements that interact with the SMovement state machine. The move operation is provided by an element outside of the state machine, then, the direction of the messages related to this call is from the SMovement to the Environment. On the other hand, obstacle is an input event for this state machine, so it represented as a message from the Environment to the SMovement lifeline. Finally, we highlight the last message with a different colour to emphasise that this event violates the property.

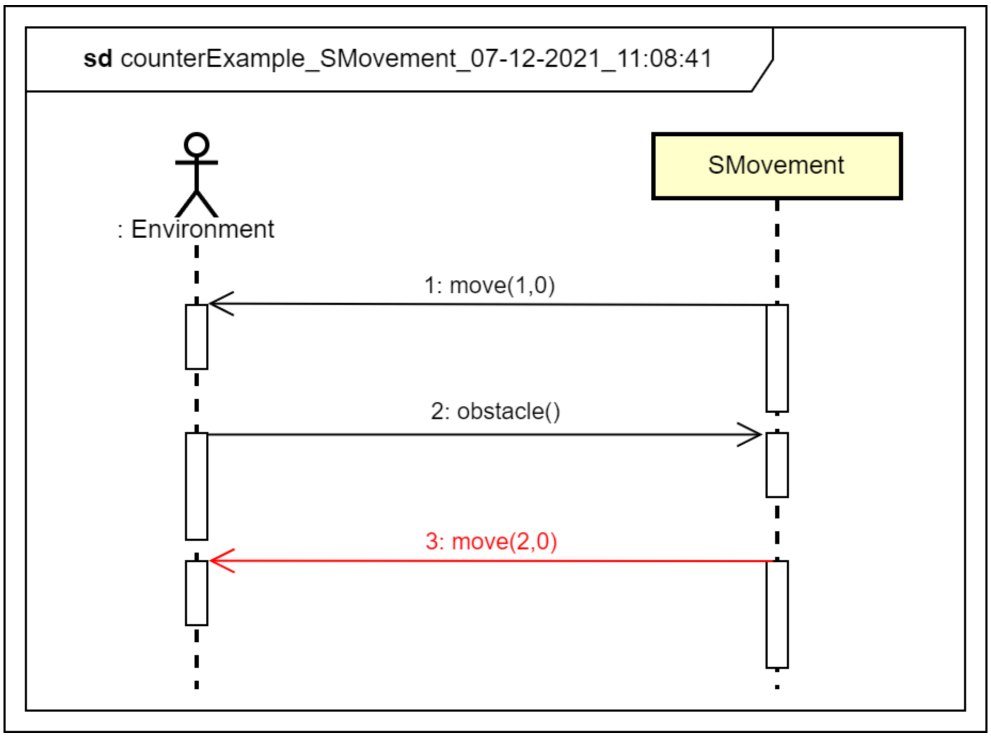


Figure 4.4: Counterexample as displayed in Astah.

5 Case Study

To validate the applicability of the approach, we use the proposed notation and tool support to model and verify non-trivial properties of two different robotic systems: Solar Panel Vacuum Cleaner and Chemical Detector. Such systems are designed using RoboChart and their domain-specific properties are expressed in CSP. The RoboChart design and the properties for the two systems can be found at RoboStar website [25]. This chapter shows how CSP properties for each of the above mentioned systems are expressed using the proposed diagrammatic notation proposed and verified using our Astah plugin.

5.1 Solar Panel Vacuum Cleaner

Solar Panel Vacuum Cleaner (SPVC) consists on a self driving robot that follows a predetermined route. The system is reported in [6]. The cleaning robot has two wheels and moves through the panels with a rolling brush to remove the panel dirt. While moving, the robot checks its battery status. If the battery level is below a predefined value, it returns to the docking station that is connected to one of the panels. Once the battery is full, the robot returns to the point where it paused for charging and continues the cleaning. The robot movement follows a predefined path that covers the entire surface of the solar panel. Figure 5.1 illustrates the robot path in the solar panel. The robot's initial position is the corner on the bottom left-hand side (near the docking station). The robot moves forward pointing to the top of the panel and uses a ultrasonic sensor to check if the edge has been reached. When the edge is reached, the robot rotates 90° to the right, walks forward a small preset distance (to avoid overlap with the already cleaned surface), turns 90° to the right and moves forward pointing to the bottom edge. When reaching the bottom edge, the robot turns 90° to the left. At this point, it repeats the movement pattern started at the initial position until it finishes the cleaning. Whenever the robot needs to return to the docking station for recharging, it rotates 180° and follows a straight line until reaching the station.

The RoboChart model for the SPVC is available at the website [25]. Such a model has a single controller with twelve instances of state machines that are used to specify the controller

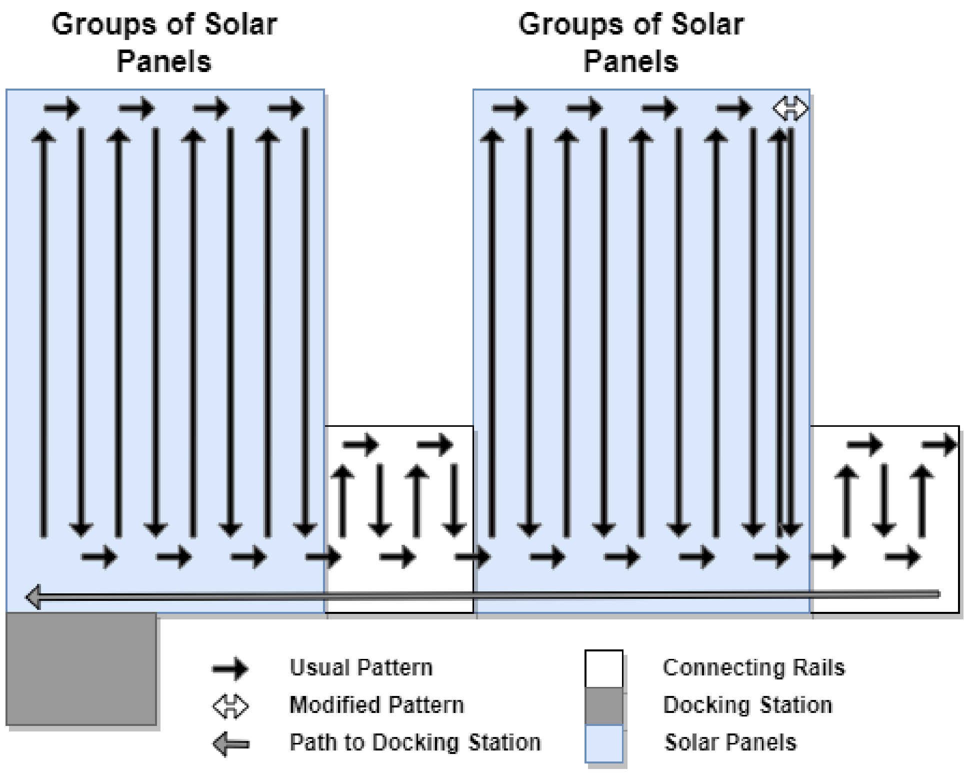


Figure 5.1: Robot movement

behaviour. The properties we model are focused on the behaviour of the PathPlanningSM state machine (reproduced in Figure 5.2), which controls the robot movement. Such state machine calls the operations move_forward, stop, and turn. The last operation receives the direction as a parameter. Moreover, the machine receives environment information through the events ultrasonic, displacement and battery_level; such events input integer values. There are also the charging and clean events that output a boolean value. The value indicates whether the charging (the cleaning) must be active or not.

In the sequence, we detail the properties for PathPlanningSM verified using our approach.

The ReturnToCharge property specifies the robot's behaviour to stop cleaning and returning to the dock station if the battery level is low; the robot continues the cleaning otherwise. Figure 5.3 represents the diagrammatic specification for such a property, say ReturnToCharge. The initial behaviour of the property is to wait for a battery_level event and to follow to a decision node that uses the current value for the battery, say b. If the value b associated with the battery level is less or equal to zero, the robot will turn left two times, move forward and disable the clean mode (set as false). At this point, the property behaviour is a loop that repeats while the value u from the input ultrasonic is less than one. The loop exits if the value is greater than or equal to one. The behaviour after the loop is to get the left direction, move forward, charge the battery and repeat the behaviour after the initial node. If the battery level is greater than zero, the robot turns left and behaves as the property NoFall.

The NoFall property exhibited in Figure 5.4, specifies the robot behaviour to move away from the panel edge to avoid falling off the panel surface. The intuition for this property is that it

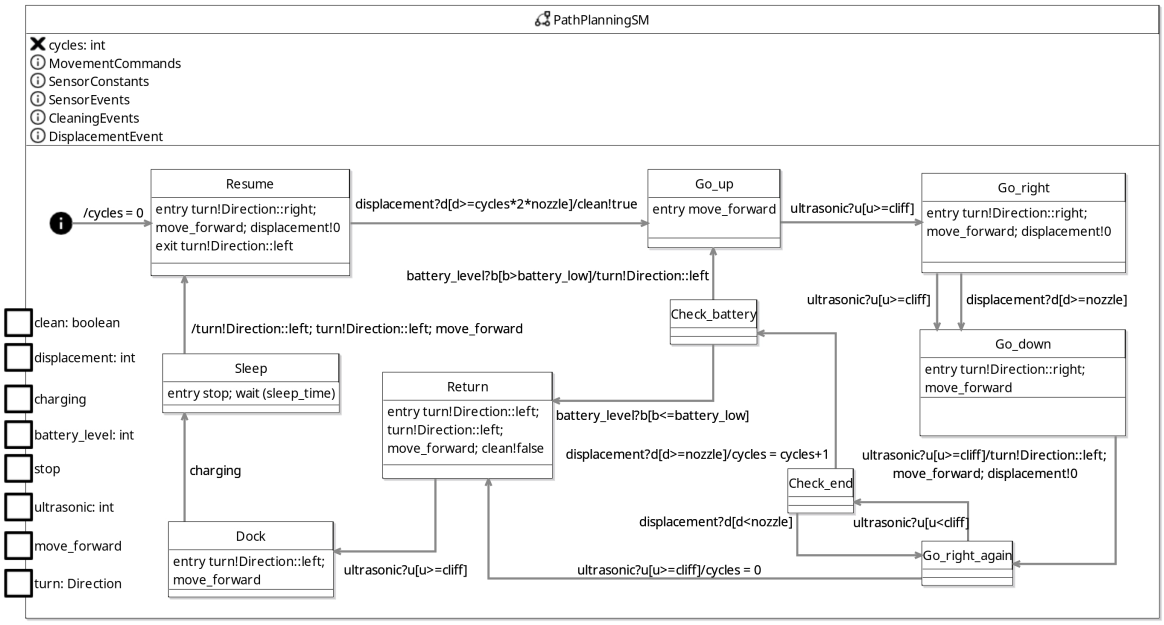

PathPlanningSM
cycles: int
MovementCommands
SensorConstants
SensorEvents
CleaningEvents
DisplacementEvent
/cycles = 0
Resume
entry turn!Direction::right; move_forward; displacement!0 exit turn!Direction::left
displacement?d[d>=cycles*2*nozzle]/clean!true
Go_up
entry move_forward
ultrasonic?u[u>=cliff]
Go_right
entry turn!Direction::right; move_forward; displacement!0
battery_level?b[b>battery_low]/turn!Direction::left
Check_battery
ultrasonic?u[u>=cliff]
displacement?d[d>=nozzle]
Go_down
entry turn!Direction::right; move_forward
clean: boolean
displacement: int
charging
battery_level: int
stop
ultrasonic: int
move_forward
turn: Direction
/turn!Direction::left; turn!Direction::left; move_forward
Sleep
entry stop; wait (sleep_time)
Return
entry turn!Direction::left; turn!Direction::left; move_forward; clean!false
battery_level?b[b<=battery_low]
displacement?d[d>=nozzle]/cycles = cycles+1
ultrasonic?u[u>=cliff]/turn!Direction::left; move_forward; displacement!0
charging
Check_end
displacement?d[d<nozzle]
ultrasonic?u[u<cliff]
Dock
entry turn!Direction::left; move_forward
ultrasonic?u[u>=cliff]
ultrasonic?u[u>=cliff]/cycles = 0
Go_right_again

Figure 5.2: State machine PathPlanningSM — RoboChart Model for SPVC

contains all the possible directions a robot can take without falling off from the panel. If the CSP process that represents the PathPlanningSM state machine refines this property, than this state machine does not cause the robot to fall. The initial behaviour of the property is to wait for the value for the ultrasonic sensor. If the value read by the sensor is below one, that represents a safe distance to the panel cliff, the property returns to the initial merge node to wait for a reading of the sensor that is closer to the cliff. If the distance is greater or equal to one, the behaviour is a combination of several directions that do not make the robot to fall. After any of the directions, the property returns to the initial merge node.

The MovementShape property shown in Figure 5.5 specifies the robot's movement patterns that are illustrated in Figure 5.1. These notes on the right-hand side document the meaning of a sequence of events. Each group of events presented (ResumeCycle, Cycle, Continue and GoBack), describes cycles that are essential to the movement of the robot. For instance, it may be used to save the robot's position after needing to stop to charge the battery. At the top of the figure, after the initial state and the merge node there is a sequence of two RoboChart events (turn(Direction_right) and move_forward) named ResumeCycling. This sequence represents the resume cycling sequence of the SPVC robot. Following this sequence of events, we have the Cycle sequence and the Continue sequence. The repetition of these two sequences illustrates the robot's movement throughout the panel. After the Continue sequence, the robot may proceed to the GoBack sequence instead of recursing to the Cycle sequence. The GoBack sequence describes the movements to return to the dock station. Following this last sequence, the robot restarts the movement behaviour by performing the ResumeCycling sequence.

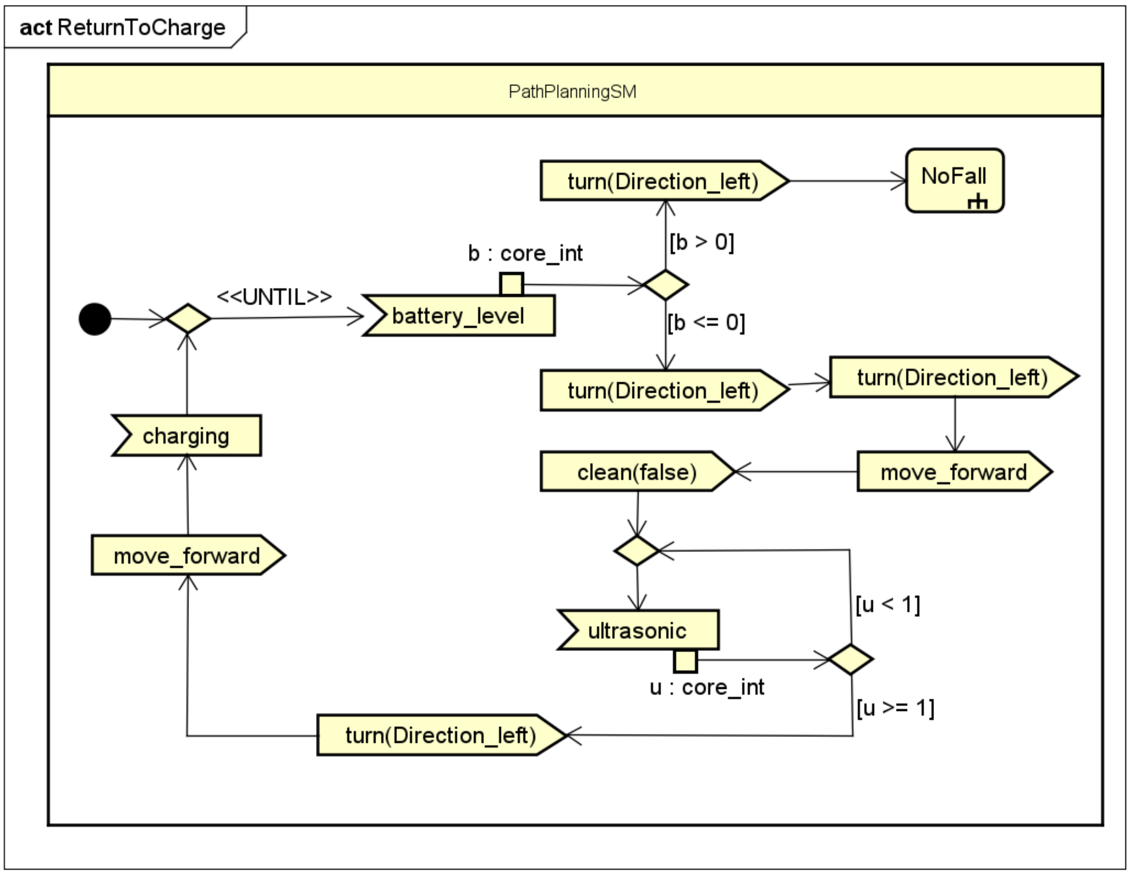


Figure 5.3: ReturnToCharge Property

5.2 Chemical Detector

We also modelled properties for the Chemical Detector robotic system designed in RoboChart by A. Miyazawa and A. L. C. Cavalcanti [21]. In such a system, the robot performs a random walk until it finds a potentially dangerous chemical product. The robot is equipped with a portable chemical monitor that creates chemical signatures and identifies dangerous substances. The chemical monitor is based on the work of James A. Hilder et al [10] that describes the application of a density receiver algorithm (an artificial immune system) used to detect chemicals. Upon finding a suspect substance, the robot uses its sensors to analyse the existence of dangerous gases and turn a light whose color (green, amber or red) indicates how dangerous is the substance. If the red colour lights on, the robot drops a flag to mark the place of the suspect substance.

The RoboChart design for the Chemical Detector is introduced in Figure 5.6. The RoboChart module (ChemicalDetectorSoftware) has two controllers connected to the platform (SensorVehicle). The MainComputer controller specifies the robot movement, while the Microcontroller controller specifies the actions performed after the detection of a substance: dropping a flag, sounding a siren and turning on a light.

The Spec1 property (see Figure 5.7) specifies the gas detection and commands should not interfere with each other. This behaviour expressed in the proposed language as the fork between

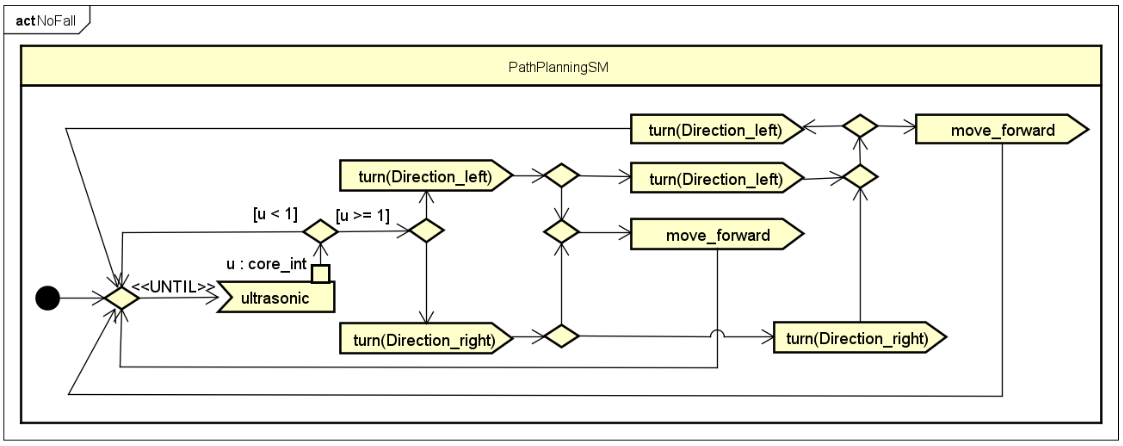


Figure 5.4: NoFall Property

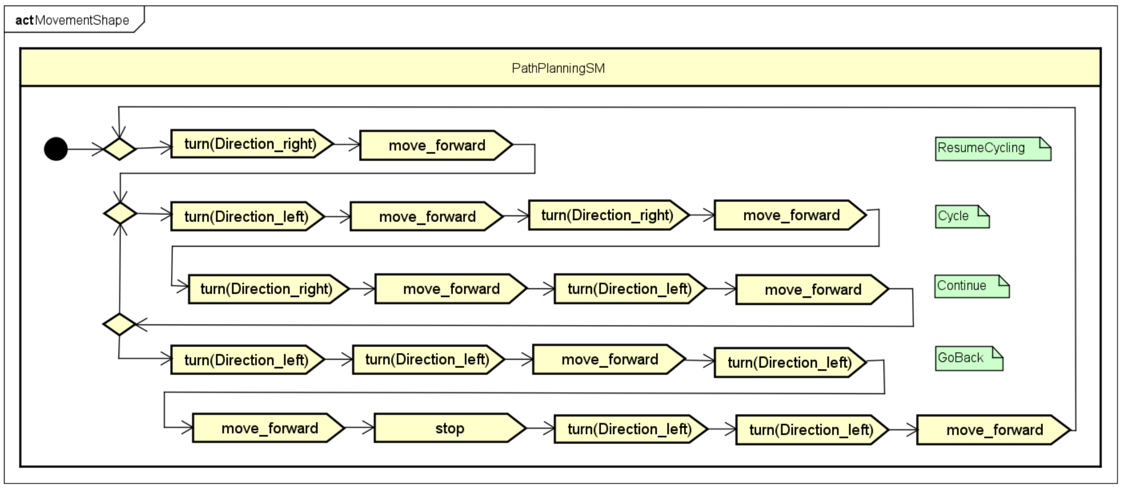


Figure 5.5: MovementShape Property

accepting commands (the diagram Commands) and detecting a gas (the diagram GasLight).

The Commands diagram, depicted in Figure 5.8, specifies that after a command is received, the robot has the choice to turn to any direction (turn), to increase the speed (incrCall) or slow down the speed (decrCall). The input pin for turning defines the angle the robot spins, it is a value that ranges between {-180..180} degrees.

The Gaslight diagram, presented in Figure 5.9, models that initially it behaves as the call behaviour to the Init diagram (Figure 5.10). This last diagram specifies the robot is prepared to detect a gas and flash the light green, in any order. After concluding Init, the Gaslight diagram specifies that, whenever a gas is detected, a light should flash. When the light flashes red, a siren must sound and a flag must be dropped. At most one gas detection can occur before the light must flash. This behaviour is recursive.

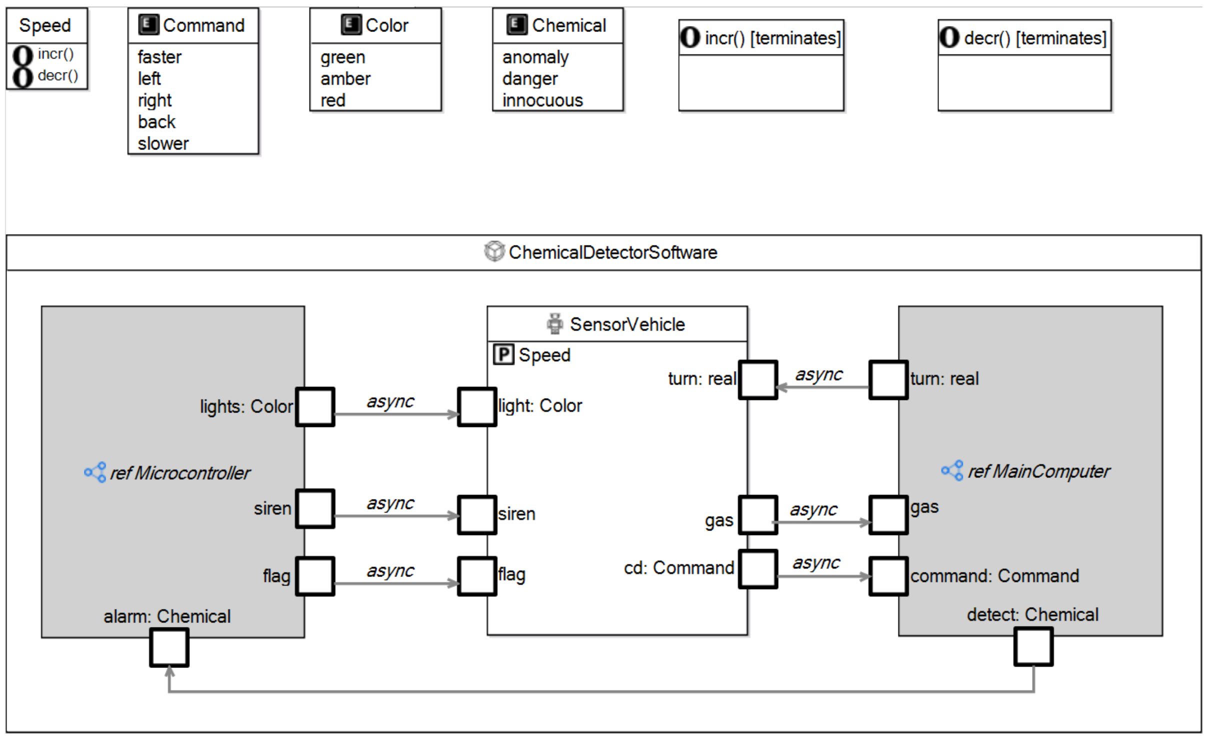


Figure 5.6: Chemical Detector - Robochart model

5.3 Verification of properties

Before verifying whether the modelled properties hold in their respective RoboChart designs, we checked the equivalence between the properties documented in the website and the properties crafted using the diagrammatic notation. We used the plugin to automatically generate a CSP process that express the diagrammatic property, and run FDR to check this process has the same traces of the process that express properties in the website [25].

Subsequently, we used the plugin to verify whether the properties described in this section hold automatically. The result of the verification showed that all properties hold, except ReturnToCharge. Figure 5.11 shows the counterexample generated by the plugin showing the behaviour of the PathPlanningSM state machine that is not expected by the property Return-ToCharge. Analysing the trace displayed in the sequence diagram, we noticed that after the ultrasonic(0) event, the robot performed the turn(Direction_left) event. However, if we look at the property, these events should only occur when the value received by ultrasonic is greater than or equal to one. After analysing the RoboChart model, we detected that the PathPlanningSM state machine (Figure 5.2) uses a constant named cliff, which corresponds to the threshold value to detect the borders of the panel. This constant had been wrongly set to zero. Once the correct value has been properly set to one, the property holds.

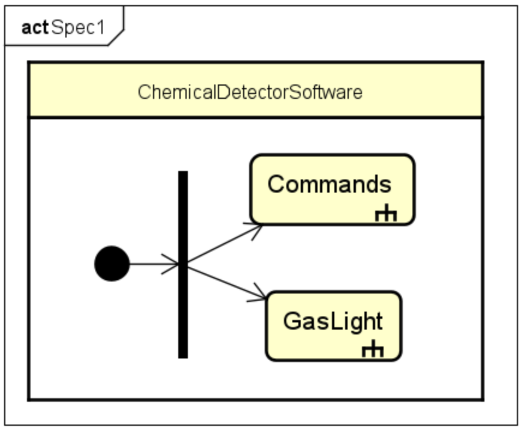


Figure 5.7: Spec1 Property

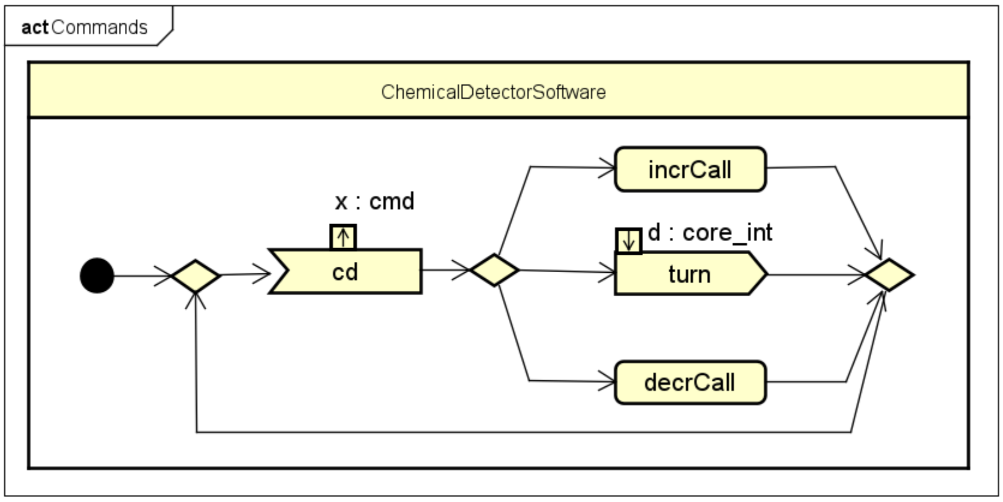


Figure 5.8: Commands Property

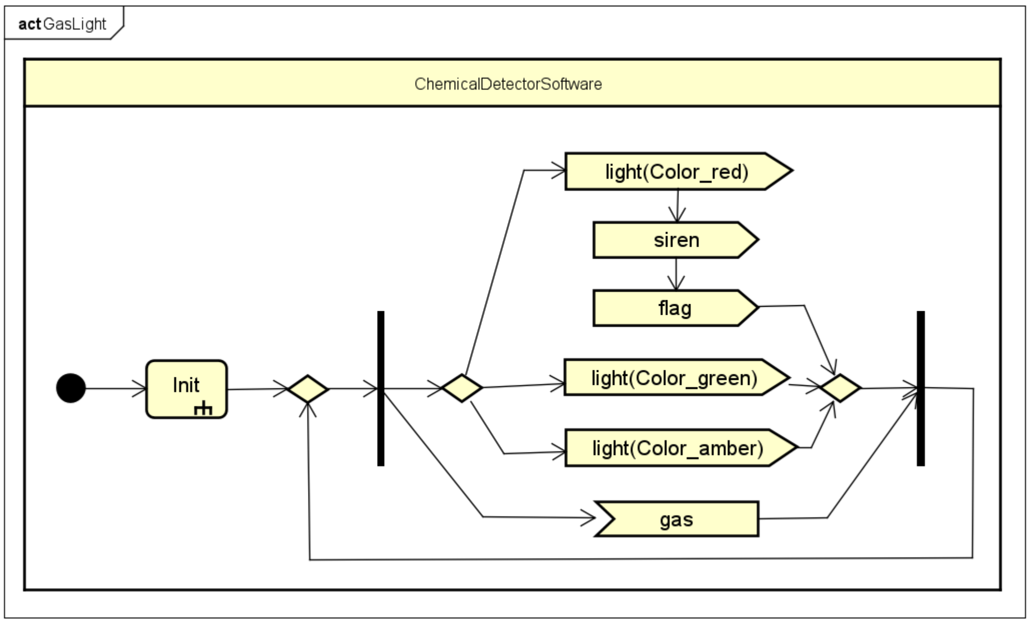


Figure 5.9: GasLight Property

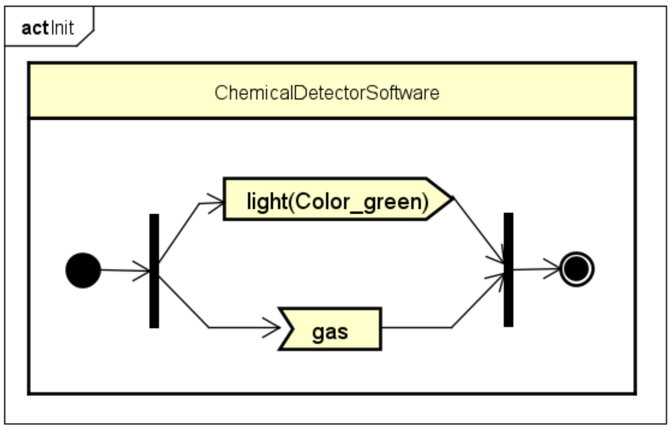


Figure 5.10: Init Property

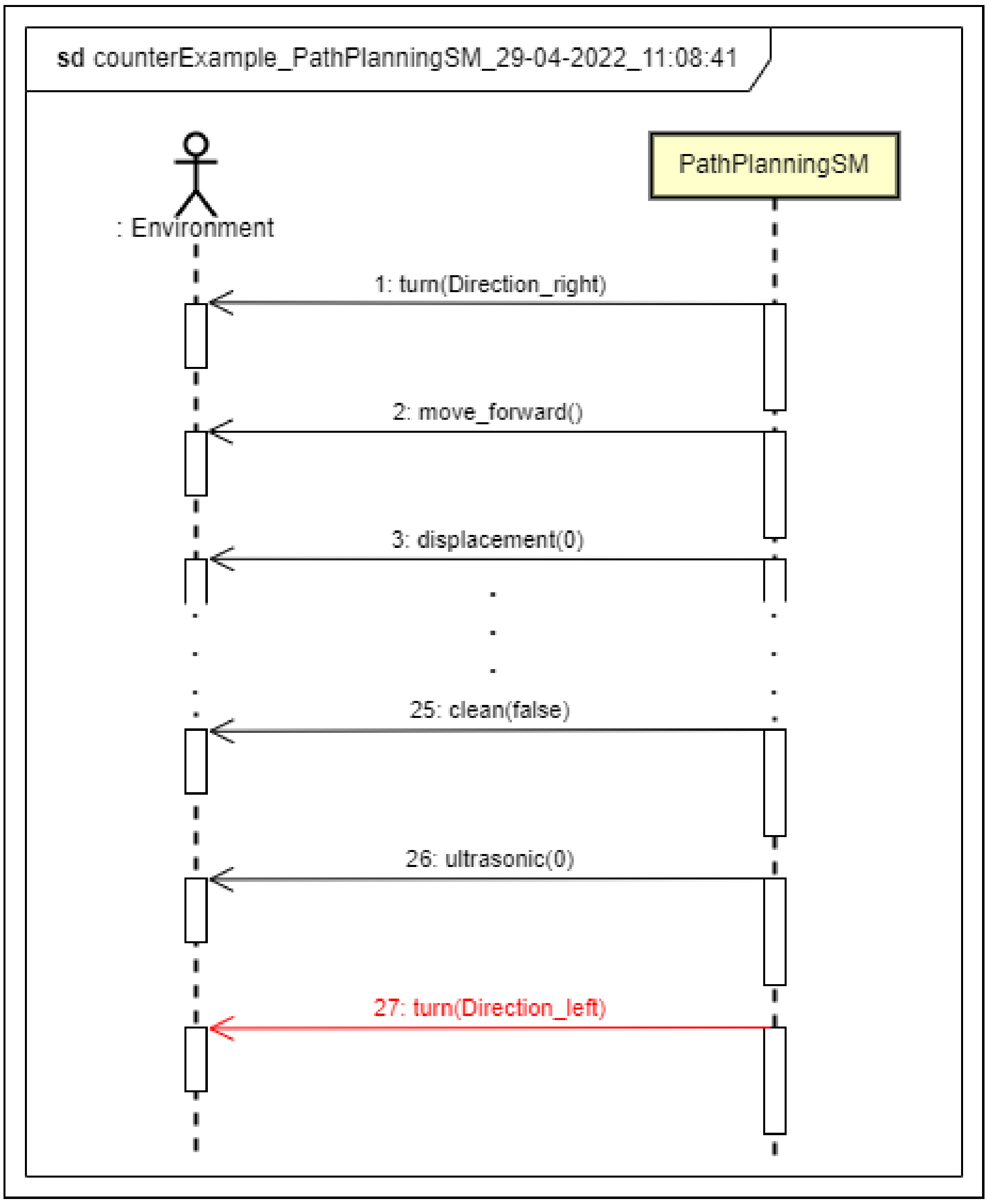


Figure 5.11: Counterexample ReturnToCharge property

6 Related Work

This work focuses on the verification of properties for RoboChart that are expressed in the form of process refinements. This is the first diagrammatic language for the specification of properties for RoboChart models.

Miyazawa et al. [22] propose a tool that checks both classical and domain-specific properties for RoboChart models that use time constraints. The proposed tool deals with properties with time constraints that are specified as CSP process refinement expressions.

The authors in [6] use RoboChart to model the behaviour of a robotic system and CSP processes to express application-specific properties. Informal requirements are formalised as CSP process refinements expressions that are verified using FDR. Processes that abstract behavioural patterns have been proposed to help the specification of properties. One of these patterns have directly influenced the <<UNTIL>> stereotype proposed by our work. We specified some properties defined in this work using our property language. In addition, using FDR, we were able to verify the equivalence between the CSP specifications generated using our property language and the ones written directly in CSP by this work. This provided evidences that our semantics supports the specification of the RoboChart properties.

The work [17] surveys notations and verification techniques based on formal methods in the context of autonomous robotic systems. Most of the works propose verification methods that input the specification as a logic formula (for instance, temporal logic [5]). As the semantic domain of RoboChart is CSP, we do not use temporal logic to specify properties but behaviours in terms of processes that the model must refine.

The authors in [19, 18] propose a new set of patterns focusing on robotic missions requirements. While in our work we used UML to abstract the CSP syntax, the authors used a structured English to describe the requirements. They implemented PsALM [18], a tool that supports the authoring of mission requirement through a structural English grammar using basic block patterns and operators (temporal logic syntax) in a compositional way. Moreover, the tool automatically translates the mission requirements into temporal logic properties in the format for the NuSMV model checker [4] and simulators [12].

The authors in [20] present a framework, called ProMoBox, for designing domain-specific modelling languages (DSMLs) that are used to specify models and their graphical Linear Temporal Logics (LTL) [28] properties. The languages are used to hide the underlying formal notation whose details are abstracted by the framework. In addition, the provided tooling support allows the verification of properties using the generated property language. The domain-specific model is translated to a *Promela* specification while the property is translated to an LTL formula. Both serve as input for the Spin model checker [11]. Like in our approach, the results of the verification are also lifted to the language of the DSML. Consequently, users do not need to manipulate formal notation or tools in any part of the verification process. Such framework is implemented using the AToMPM [31] modelling tool.

The authors in [13] solve the consistency checking problems of concurrent system designs modelled with Petri nets [1] for scenario-based specifications specified with Message Sequence Charts (MSC) [9]. Algorithms are used to check properties as if a given scenario specified using MSC must happen or is forbidden in the possible runs of a Petri net. In this work, the authors argue that using MSC is much more acceptable to the industry than using temporal logic.

Table 6.1 makes a comparative summary between our work and some others cited so far. Table columns present features considered in the related works. The column *Model Notation* shows the notation used to specify the system. The column *Property Notation* depicts the notation used for describing the properties to be verified, and column *Traceability* indicates whether the work traces the counterexample to the notation used for modelling. Looking Table 6.1, we see that only our approach and the one in [20] use non formal notations for the specification of the model, the property and to represent the counterexample traces. Nonetheless, our approach contrasts with the existent approaches for RoboChart ([6, 22]) that use CSP to specify properties and to represent the counterexamples.

Table 6.1: Related Work

Work	**Model Notation**	**Property Notation**	**Traceability**
[6]	RoboChart	CSP	X
[22]	RoboChart	CSP	X
[18, 19]	FSM	Structured English	X
[20]	DSML	Graphical DSML	✓
[13]	Petri Nets	MSC	X
Our work	RoboChart	Activity diagram	✓

7 Conclusion

We have introduced a diagrammatic language based on activity diagrams for reasoning on application-specific properties of RoboChart models. Several diagram elements like swimlanes, actions, pins have been adapted to specify RoboChart observations as events and calls to operations. Furthermore, we have extended activity diagrams with the <<*UNTIL*>> and <<*ANY*>> stereotypes to increase the expressiveness of the notation. The presented language has compositional semantics in terms of CSP processes, which uses the same encoding for CSP events used in the underlying semantics of RoboChart models. This allows us to verify the specified properties against RoboChart models using FDR. We have implemented a plug-in for the Astah UML tool to specify these properties using extended activity diagrams and verify them against the specifications of RoboChart models in an automated manner. Finally, we applied the approach to model and verify properties of two different robotic systems.

Another contribution of our work is to provide traceability of the counterexample yielded by FDR to the diagrammatic level. We have implemented in the plugin the translation of the counterexample as a sequence diagram. In this approach, instead of reading a sequence of CSP events, the user analyses the sequence of events and operations exchanged between the RoboChart component and its environment.

The mechanisation of the approach is relevant because, currently, the users of RoboTool must specify application-specific properties directly in CSP and understand the counterexample of FDR if the property does not hold. The current work enables RoboChart designers to specify both the system and the properties at the same level of abstraction, given that models and properties use diagrammatic notations. We hope that hiding the formal notation of CSP can facilitate and increase the adoption of RoboChart by the robotics community.

The proposed notation is based on activity diagrams that are very popular and have a repertoire of constructs that allow the specification of a wide range of behaviours. Nonetheless, it is not expressive as the CSP notation that has a richer language. Comparing the expressiveness of the proposed language with the expressiveness of CSP is left as future work.

Application-specific properties over RoboChart models are formulated as process refinement assertions. Different CSP models can be used to verify safety (traces model) and liveness properties (failures-divergence and refusal traces). The properties specified by the proposed language can be verified using any of the existing CSP models; however, this work considered only safety properties. A future plan is to consider other kinds of properties.

Another point of improvement is to allow the specification of visual properties in the RoboTool platform instead of using a different tool to model and verify the properties.

Although the CSP specifications generated from our diagrammatic properties are usually larger than those specified directly in CSP, we verified that they have equivalent semantics whenever the corresponding property is available in CSP. Using FDR compression functions, the complexity of CSP specifications is significantly reduced, and this allowed us to perform the analysis at similar times compared to properties specified in CSP. Nevertheless, we plan to perform a more concrete study on scalability in the future.

Our tool supports a considerable number of activity diagram constructors, which allows the designing of potentially elaborated properties. However, we plan to increase this number to augment expressiveness. For instance, we do not cover timing aspects of the properties in the current version of our semantics. At last, we plan to develop more case studies to explore the proposed language and the reasoning strategy.

A
APPENDIX

A.1 CSP specification for the Angular property

This appendix presents the contents of the CSP specification generated by our plugin for the Angular property presented in Chapter 3. For better understanding and organisation, we split the specification in four snippets that are presented by Figures A.1, A.2, A.3 and A.4.

Figure A.1 introduces the constants, channels, datatypes for the CSP specification of the property. Moreover, this snippet includes the CSP specification generated by RoboTool for the RoboChart model for SimFW (Line 2). In Line 3, we can see the diagram identifier declaration and the number of existing instances. In Line 4, we have the datatype that specifies the tokens which identify each node of the diagram. In Line 5, we have the alphabet of the events from RoboChart that occur in the modelled property. In Lines 6, 7 and 8, we have a counter for control edges, another counter for active tokens (only when there is concurrency this number is greater than one) and another counter for the number of activity final nodes. In Line 9, we determine a limit of the arithmetic operations performed on the counting of flows to allow model checking. This limit is calculated based on the possible concurrent flows. From Lines 10 to 17, we have the channels: to start and end the activity, the control edges, to clean tokens, to control active tokens within the diagram and synchronise the termination of activity nodes. In Line 16, we use the *dc* channel to capture/generate non-determinism on the output edges of decision nodes. In Line 17 we have two auxiliary events to control the UNTIL pattern. In Line 19 we define an alphabet of internal control events which is used later to make only RoboChart events visible.

The main process and auxiliary functions that calculate the alphabets of the activity nodes are presented in Figure A.2. In this figure, Line 21 presents the main process of the translated diagram. In Line 22, we have an auxiliary process used to synchronise events when the diagram concludes. In Line 24, we have the process that is invoked by the main process that composes the internal part of the diagram with the Token Manager, which defines the beginning, the execution and the end of the diagram. From Lines 25 to 30 we have the alphabet functions that use the previously mentioned tokens for each node. Such functions return the alphabet of each node. In Line 31, we have the union of all these alphabets.

```
1  transparent normal
2  include "file_SimFW_coreassertions.csp"
3  ID_SimFW = {1..1}
4  datatype alphabet_SimFW = InitialNode0_SimFW_t_alphabet | accept_obstacle_1_SimFW_t_alphabet|
   DecisionNode_MergeNode0_SimFW_t_alphabet| move_0_2__1_SimFW_t_alphabet
5  robochart_alphabet_SimFW = {|SMovement::moveCall,SMovement::obstacle.in,SMovement::obstacle.out|}
6  countCe_SimFW = {1..4}
7  countUpdate_SimFW = {1..1}
8  countClear_SimFW = {1..0}
9  limiteUpdate_SimFW = {(1)..(1)}
10 channel startActivity_SimFW: ID_SimFW
11 channel endActivity_SimFW: ID_SimFW
12 channel ce_SimFW: ID_SimFW.countCe_SimFW
13 channel clear_SimFW: ID_SimFW.countClear_SimFW
14 channel update_SimFW: ID_SimFW.countUpdate_SimFW.limiteUpdate_SimFW
15 channel endDiagram_SimFW: ID_SimFW
16 channel dc
17 channel begin, end:  {1..1}
18
19 alphabet_Astah_SimFW = {| startActivity_SimFW, endActivity_SimFW, ce_SimFW, clear_SimFW,
   update_SimFW, endDiagram_SimFW, dc |}
20
```

Figure A.1: Angular - Channels and datatypes

```
21 MAIN = normal(SimFW(1))
22 END_DIAGRAM_SimFW(id) = endDiagram_SimFW.id -> SKIP
23 SimFW(ID_SimFW) = (Internal_SimFW(ID_SimFW) [|{|update_SimFW,clear_SimFW,endDiagram_SimFW|}|]
   TokenManager_SimFW_t(ID_SimFW,0,0))
24 Internal_SimFW(id) = StartActivity_SimFW(id); Node_SimFW(id); EndActivity_SimFW(id)
25 StartActivity_SimFW(id) = startActivity_SimFW.id -> SKIP
26 EndActivity_SimFW(id) = endActivity_SimFW.id -> SKIP
27 AlphabetDiagram_SimFW(id,InitialNode0_SimFW_t_alphabet) = {|ce_SimFW.id.1,endDiagram_SimFW.id|}
28 AlphabetDiagram_SimFW(id,accept_obstacle_1_SimFW_t_alphabet) = {|ce_SimFW.id.3,ce_SimFW.id.4,
   endDiagram_SimFW.id|}
29 AlphabetDiagram_SimFW(id,DecisionNode_MergeNode0_SimFW_t_alphabet) = {|ce_SimFW.id.1,ce_SimFW.id.2,
   ce_SimFW.id.3,endDiagram_SimFW.id|}
30 AlphabetDiagram_SimFW(id,move_0_2__1_SimFW_t_alphabet) = {|ce_SimFW.id.4,ce_SimFW.id.2,
   endDiagram_SimFW.id|}
31 AlphabetDiagram_SimFW_t(id) = union(union(union(AlphabetDiagram_SimFW(id,
   InitialNode0_SimFW_t_alphabet),AlphabetDiagram_SimFW(id,accept_obstacle_1_SimFW_t_alphabet)),
   AlphabetDiagram_SimFW(id,DecisionNode_MergeNode0_SimFW_t_alphabet)),AlphabetDiagram_SimFW(id,
   move_0_2__1_SimFW_t_alphabet))
32
```

Figure A.2: Angular - Main Process and Alphabet diagram

In Figure A.3 we have from Lines 33 to 36 the functions that yield the processes for the nodes. Given the node token, each function yields the CSP process for a particular node. From lines 37 to 44 we have the process of each node, note that there are two very similar lines for each process, one representing the behaviour of the node and the other guaranteeing termination, that is, it can be interrupted by the *endDiagram* event. In lines 46 and 47, we have the Token Manager process that controls the amount of active tokens in the diagram.

In Figure A.4, we have in Line 49 a refinement expression in the traces model. From Lines 51 to 70 we have the functions used to describe the modelled property. In the other lines we have the call for the composition of the nodes that will use both the alphabet functions and the function that return the processes of the nodes.

```
ProcessDiagram_SimFW(id,InitialNode0_SimFW_t_alphabet) = normal(InitialNode0_SimFW_t(id))
ProcessDiagram_SimFW(id,accept_obstacle_1_SimFW_t_alphabet) = normal(accept_obstacle_1_SimFW_t(id))
ProcessDiagram_SimFW(id,DecisionNode_MergeNode0_SimFW_t_alphabet) = normal (DecisionNode_MergeNode0_SimFW_t(id))
ProcessDiagram_SimFW(id,move_0_2__1_SimFW_t_alphabet) = normal(move_0_2__1_SimFW_t(id))
InitialNode0_SimFW(id) = update_SimFW.id.1!(1-0) -> ((ce_SimFW.id.1 -> SKIP))
InitialNode0_SimFW_t(id) = InitialNode0_SimFW(id) /\ END_DIAGRAM_SimFW(id)
DecisionNode_MergeNode0_SimFW(id) = ((ce_SimFW.id.1 -> SKIP) [] (ce_SimFW.id.2 -> SKIP)); ce_SimFW.id.3 -> DecisionNode_MergeNode0_SimFW(id)
DecisionNode_MergeNode0_SimFW_t(id) = DecisionNode_MergeNode0_SimFW(id) /\ END_DIAGRAM_SimFW(id)
accept_obstacle_1_SimFW(id) = ((ce_SimFW.id.3 -> SKIP)); begin.1 -> end.1 -> SKIP; ((ce_SimFW.id.4 -> SKIP)); accept_obstacle_1_SimFW(id)
accept_obstacle_1_SimFW_t(id) = accept_obstacle_1_SimFW(id) /\ END_DIAGRAM_SimFW(id)
move_0_2__1_SimFW(id) = ((ce_SimFW.id.4 -> SKIP)); SMovement::moveCall.0.2 -> ((ce_SimFW.id.2 -> SKIP)); move_0_2__1_SimFW(id)
move_0_2__1_SimFW_t(id) = move_0_2__1_SimFW(id) /\ END_DIAGRAM_SimFW(id)

TokenManager_SimFW(id,x,init) = update_SimFW.id?c?y:limiteUpdate_SimFW -> x+y < 10 & x+y > -10 & TokenManager_SimFW(id,x+y,1) [] clear_SimFW.id?c -> endDiagram_SimFW.id -> SKIP [] x == 0 & init == 1 & endDiagram_SimFW.id -> SKIP
TokenManager_SimFW_t(id,x,init) = TokenManager_SimFW(id,x,init)

```

Figure A.3: Angular - Process diagram and Token manager

```
assert Prop_SimFW [T= P_SMovement

NRecurse(S, P) = |~| ev : S @ ev -> P

WAIT(alphabet,event) =
    NRecurse(diff(alphabet, {event}), WAIT(alphabet,event))
    |~|
    event -> SKIP

WAIT_PROCCESSES_SimFW(processes) = ( ||| CONTROL : processes @ CONTROL )  /\ endDiagram_SimFW?id -> SKIP

Prop_SimFW = PROP_SimFW(Wait_control_processes_SimFW) \ alphabet_Astah_SimFW

alphabet_robochart_SimFW = {|SMovement::moveCall,SMovement::obstacle.in,SMovement::obstacle.out|}

PROP_SimFW(processes) = (MAIN [|{|begin, end, endDiagram_SimFW|}|] WAIT_PROCCESSES_SimFW (processes) ) \ {|begin, end|}

Wait_SimFW_1 = WAIT(alphabet_robochart_SimFW, SMovement::obstacle.in)

Wait_SimFW_control_1 = begin.1 -> Wait_SimFW_1; end.1 -> Wait_SimFW_control_1

Wait_control_processes_SimFW = {Wait_SimFW_control_1}

Node_SimFW(id) = composeNodes_SimFW(id)

composeNodes_SimFW(id) =
    let
        alphabet_SimFW_s = seq(alphabet_SimFW)
        composeNodes_(id,<ev>,_) = ProcessDiagram_SimFW(id,ev)
        composeNodes_(id,<ev>^tail,past) =
            ProcessDiagram_SimFW(id,ev)
                [|union(diff(AlphabetDiagram_SimFW(id,ev),past),{endDiagram_SimFW.id})|]
            ( composeNodes_(id,tail,union(past,AlphabetDiagram_SimFW(id,ev))) )
    within
        composeNodes_(id,alphabet_SimFW_s,{})

```

Figure A.4: Angular - Refinement and *composeNodes*

References

[1] *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the Volumes Are Based on the Advanced Course on Petri Nets*, Berlin, Heidelberg, 1996. Springer-Verlag.

[2] Sonya Alexandrova, Zachary Tatlock, and Maya Cakmak. Roboflow: A flow-based visual programming language for mobile manipulation tasks. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5537–5544. IEEE, 2015.

[3] Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[4] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proceedings of the 14th International Conference on Computer Aided Verification*, CAV '02, page 359–364, Berlin, Heidelberg, 2002. Springer-Verlag.

[5] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions Programming Languages Systems*, 8(2):244–263, 1986.

[6] Bianca Darolți. Software engineering for robotics: an autonomous robotic vacuum cleaner for solar panels. Master's thesis, University of York, 2019.

[7] Saadia Dhouib, Selma Kchir, Serge Stinckwich, Tewfik Ziadi, and Mikal Ziane. Robotml, a domain-specific language to design, simulate and deploy robotic applications. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 149–160. Springer, 2012.

[8] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3 17 A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 187–201, 2014.

[9] David Harel and PS Thiagarajan. Message sequence charts. In *UML for Real*, pages 77–105. Springer, 2003.

[10] James A Hilder, Nick DL Owens, Mark J Neal, Peter J Hickey, Stuart N Cairns, David PA Kilgour, Jon Timmis, and Andy M Tyrrell. Chemical detection using the receptor density algorithm. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(6):1730–1741, 2012.

[11] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23:279–295, May 1997.

[12] Louis Hugues and Nicolas Bredeche. Simbad: an autonomous robot simulation package for education and research. In *International Conference on Simulation of Adaptive Behavior*, pages 831–842. Springer, 2006.

[13] Xuandong Li, Jun Hu, Lei Bu, Jianhua Zhao, and Guoliang Zheng. Consistency checking of concurrent models for scenario-based specifications. In *International SDL Forum*, pages 298–312. Springer, 2005.

[14] Lucas Lima, Amaury Tavares, and Sidney C. Nogueira. A framework for verifying deadlock and nondeterminism in uml activity diagrams based on csp. *Science of Computer Programming*, 197:102497, 2020.

[15] Waldeck Lindoso, Sidney C. Nogueira, Renato Domingues, and Lucas Lima. Visual specification of properties for robotic designs. In *Formal Methods: Foundations and Applications: 24th Brazilian Symposium, SBMF 2021, Virtual Event, December 6–10, 2021, Proceedings*, page 34–52, Berlin, Heidelberg, 2021. Springer-Verlag.

[16] Gavin Lowe. Specification of communicating processes: Temporal logic versus refusals-based refinement. *Form. Asp. Comput.*, 20(3):277–294, May 2008.

[17] Matt Luckcuck, Marie Farrell, Louise A. Dennis, Clare Dixon, and Michael Fisher. Formal specification and verification of autonomous robotic systems: A survey. *ACM Comput. Surv.*, 52(5), September 2019.

[18] Claudio Menghi, Christos Tsigkanos, Thorsten Berger, and Patrizio Pelliccione. Psalm: Specification of dependable robotic missions. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 99–102. IEEE, 2019.

[19] Claudio Menghi, Christos Tsigkanos, Patrizio Pelliccione, Carlo Ghezzi, and Thorsten Berger. Specification patterns for robotic missions. *IEEE Transactions on Software Engineering*, 47(10):2208–2224, 2019.

[20] Bart Meyers, Hans Vangheluwe, Joachim Denil, and Rick Salay. A framework for temporal verification support in domain-specific modelling. *IEEE Transactions on Software Engineering*, 46(4):362–404, 2018.

[21] A. Miyazawa and A. L. C. Cavalcanti. Chemical detector. Available in: `https://robostar.cs.york.ac.uk/case_studies/autonomous-chemical-detector/autonomous-chemical-detector.html`. Access at: may 05, 2022.

[22] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, and Jon Timmis. Automatic property checking of robotic applications. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3869–3876. IEEE, 2017.

[23] Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, Jon Timmis, and Jim Woodcock. Robochart: modelling and verification of the functional behaviour of robotic applications. *Software & Systems Modeling*, 18(5):3097–3149, 2019.

[24] Yvonne Murray, David A. Anisi, Martin Sirevåg, Pedro Ribeiro, and Rabah Saleh Hagag. Safety assurance of a high voltage controller for an industrial robotic system. In Gustavo Carvalho and Volker Stolz, editors, *Formal Methods: Foundations and Applications*, pages 45–63, Cham, 2020. Springer International Publishing.

[25] Department of Computer Science: University of York. Robochart case studies. Available in: `https://robostar.cs.york.ac.uk/case_studies/`. Access at: may 04, 2022.

[26] OMG. OMG Unified Modeling Language (OMG UML), Version 2.5.1. Technical report, Object Management Group, December 2017.

[27] Izzet Pembeci, Henrik Nilsson, and Gregory Hager. Functional reactive robotics: An exercise in principled integration of domain-specific languages. In *Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 168–179, 2002.

[28] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, page 46–57, USA, 1977. IEEE Computer Society.

[29] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall PTR, 1998.

[30] A. W. Roscoe. On the expressive power of csp refinement. *Formal Aspects of Computing*, 17(2):93–112, 2005.

[31] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Huseyin Ergin. Atompm: A web-based modeling environment. In *Joint proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013): September 29-October 4, 2013, Miami, USA*, pages 21–25, 2013.