

KRR Project - Documentation

Sonia Grzywacz 245492

Zofia Wrona 305803

Patryk Grochowicz 305779

Piotr Krzemiński 305785

Marcin Świerkot 294569

Warsaw University of Technology
Faculty of Computer Science and Information Systems

18th March 2024

Abstract

The aim of the task was to define an action description language $\mathcal{ADL}_{CS\mathcal{T}}$ and corresponding query language $\mathcal{QL}_{CS\mathcal{T}}$ that will allow describing a specific group of dynamic systems that takes into account cost-related aspects. Accordingly, this contribution consists of two parts - the theoretical description of the task and the documentation of its practical implementation. In the first part, the introduction provides a brief overview of the task, including the assumptions applied to the corresponding group of dynamic systems. Then, the action description language is defined by means of its syntax and semantics. Following that, the query language is discussed in terms of proposed queries and their satisfiability conditions. Lastly, the theory is concluded with descriptions of various examples that provide an understanding of how the proposed languages can be applied in practice. The second part focuses on the description of the technical aspects of the implemented solution – i.e. used technology, implemented interface and the performed tests.

Contents

1	Theoretical description	3
1.1	Introduction	3
1.1.1	Task description	3
1.2	Action description language	3
1.2.1	Syntax	4
1.2.2	Semantics	4
1.3	Query Language	6
1.3.1	Query statements	6
1.3.2	Satisfiability of queries	6
1.4	Examples of dynamic systems	6
1.4.1	Titanic	6
1.4.2	Shipping orders	9
1.4.3	Preparing for holidays	11

2	Technical description	14
2.1	Introduction	14
2.2	Technology	14
2.3	Classes	14
2.4	Sections	15
3	User guide	17
3.1	Application	17
3.2	Instruction how to run application	17
3.3	Application functionalities	17
4	Test description	22
4.1	Testing user experience (UX/UI testing)	23
4.2	Syntax and semantic correctness testing	26
4.3	Test 1 - Yale Shooting Problem	26
4.4	Test 2 - YSP modified with after statement	28
4.5	Test 3 - Action domain inconsistency testing	30
4.6	Test 4 - Titanic example	31
4.7	Test 5 - Shipping orders example	32
5	Individual contribution	34

1 Theoretical description

1.1 Introduction

Let DS_{CST} be a class of dynamic systems that are the primary focus of this task. DS_{CST} fulfills the following assumptions:

- A1. **Inertia law** – the observed changes, after performing a given action, are only the ones that are directly or indirectly induced by the aforementioned action
- A2. **Complete information about all actions and all fluents** – there are no unknown effects of the actions
- A3. **Determinism** – there is only one possible outcome of an action
- A4. **Branching model of time** – time steps are represented by branching points
- A5. **Only sequential actions are allowed** – the parallel actions are out of consideration
- A6. **Domain constraints are not admitted** – there are no restrictions set for the domain
- A7. **All actions can be performed in all states** – all actions are executable
- A8. **Characteristics of actions:**
 - ▷ precondition (a set of literals) reflecting a condition under which the action starts and leads to some effect. If a precondition does not hold, then the action is executed with an empty effect
 - ▷ postcondition (a set of literals) reflecting the effect of the action
 - ▷ cost $\kappa \in \mathbb{N}$ required to perform the action. It depends on the state in which the action starts

1.1.1 Task description

The task of this project is stated as follows:

Define an action description language ADL_{CST} for representing dynamic systems of the class specified above, and define the corresponding query language QL_{CST} which would allow getting answers for the following queries:

- Q1. Does a condition γ (set of literals) hold after performing a program \mathcal{P} ?*
- Q2. Is cost κ sufficient to perform a program \mathcal{P} ?*

1.2 Action description language

Let ADL_{CST} be an action description language defined to represent a class of dynamic systems that satisfy the assumptions outlined in the previous section. The following terminology will be used to specify the syntax of the ADL_{CST} language:

$\Upsilon = (\mathcal{F}, \mathcal{A}_c)$ - signature of the language, where:

- \mathcal{F} is a set of fluents
- \mathcal{A}_c is a set of actions

\bar{f} - literal corresponding to the fluent $f \in \mathcal{F}$ or its negation $\neg f$

κ - cost corresponding to the action execution ($k \in \mathbb{N}$)

$\mathcal{P} = (A_1, \dots, A_n)$ - program, sequence of actions.

1.2.1 Syntax

Two types of statements were defined for the $\mathcal{ADL}_{\mathcal{CS}\mathcal{T}}$ language. The first ones are the **value statements**. They describe the state (more precisely, fluents) that initially holds in the system or holds in the system after performing a particular sequence of actions. The second types of statement are the **effect statements**. They describe how the system's state changes after performing a given action, namely, what state will result from performing a given action under specified preconditions.

1. Value statements

It is important to stress that the second value statement is, in fact, an abbreviation of the first one (i.e. it is not a separate value statement).

1. \bar{f} **after** A_1, \dots, A_n (where $f \in \mathcal{F}$ and $A_1, \dots, A_n \in \mathcal{A}_c$)

Meaning: \bar{f} holds after performing a set of actions A_1, \dots, A_n in the initial state

2. **initially** \bar{f} where $f \in \mathcal{F}$

Meaning: \bar{f} holds in the initial state

2. Effect statements

Similarly to the previous section, the second effect statement is an abbreviation of the first one. It emphasizes that the action is executed with no specific preconditions that have to be met before its execution.

1. A **causes** \bar{f} if $\bar{g}_1, \dots, \bar{g}_k$ **cost** κ where $f, g_1, \dots, g_k \in \mathcal{F}$, $A \in \mathcal{A}_c$ and $\kappa \in \mathbb{N}$

Meaning: If A is performed in any state satisfying $\bar{g}_1, \dots, \bar{g}_k$ then in the resulting state \bar{f} holds, and the cost that is required to perform such an action is equal to κ

2. A **causes** \bar{f} **cost** κ where $f \in \mathcal{F}$ and $\kappa \in \mathbb{N}$

Meaning: The performance of action A leads to the state for which \bar{f} holds and the cost that is required to perform such an action is equal to κ

1.2.2 Semantics

The **structure** of a language $\mathcal{ADL}_{\mathcal{CS}\mathcal{T}}$ is a pair $S = (\Psi, \sigma_0)$ where Ψ is a transition function and $\sigma_0 \in \Sigma$ is the initial state.

A **state** is defined as a mapping $\sigma: \mathcal{F} \rightarrow \{0, 1\}$, for any $f \in \mathcal{F}$.

If $\sigma(f) = 1$, then it means that f holds in the state σ and it is denoted by $\sigma \models f$.

If $\sigma(f) = 0$, then it means that f doesn't hold in the state σ and it is denoted $\sigma \models \neg f$.

Furthermore, let Σ denote a set of all states.

1. Transition function

A **transition function** is defined as a mapping $\Psi: (\mathcal{A}_c, \kappa) \times \Sigma \rightarrow \Sigma$.

$\Psi((A, \kappa), \sigma)$ defined for any $A \in \mathcal{A}_c$, for any $\sigma \in \Sigma$ and for any $\kappa \in \mathbb{N}$ gives a resulting state after performing the action A from the state σ with cost κ , where κ is a sum of all costs associated with effects of the action A .

The aforementioned function can also be generalized to the mapping $\Psi^*: (\mathcal{A}_c, \kappa)^* \times \Sigma \rightarrow \Sigma$ in a following way:

1. $\Psi^*((\epsilon, 0), \sigma) = \sigma$

$$2. \Psi^*((A_1, \kappa_1), \dots, (A_n, \kappa_n)), \sigma) = \Psi((A_n, \kappa_n), \Psi^*((A_1, \kappa_1), \dots, (A_{n-1}, \kappa_{n-1})), \sigma))$$

However, for the sake of simplification, the Ψ^* will be further denoted as Ψ .

2. Satisfiability of language statements

Let $S = (\Psi, \sigma_0)$ be a structure for a language $\mathcal{ADL}_{\mathcal{CS}\mathcal{T}}$. A statement s is true in S (denoted by $S \models s$), if and only if:

1. s is of the form: \bar{f} **after** A_1, \dots, A_n

$S \models s$ if and only if for an initial state $\sigma_0 \in \Sigma$ and the set of actions $A_1, \dots, A_n \in \mathcal{A}_c$, the following holds: $\Psi^*((A_1, \kappa_1), \dots, (A_n, \kappa_n)), \sigma)$

2. s is of the form: A **causes** \bar{f} if $\bar{g}_1, \dots, \bar{g}_k$ **cost** κ

$S \models s$ if and only if for every state $\sigma \in \Sigma$ such that $\sigma \models \bar{g}_i, i = 1, \dots, k$ and the action $A \in \mathcal{A}_c$, the following holds: $\Psi((A, \kappa), \sigma) \models \bar{f}$ and cost of performing the action is κ

3. Model of a language

Let D be an action domain (non-empty set of value or effect statements) in the language $\mathcal{ADL}_{\mathcal{CS}\mathcal{T}}$ over a signature $\Upsilon = (\mathcal{F}, \mathcal{A}_c)$. A structure for this language $S = (\Psi, \sigma_0)$ is a model of D if and only if:

1. for every $s \in D$, $S \models s$
2. for every $A \in \mathcal{A}_c$, for every $f, g_1, \dots, g_k \in \mathcal{F}$, for every $\sigma \in \Sigma$ and for every $\kappa \in \mathbb{N}$, if one of the following conditions holds:

(a) for every effect statement in D of the form A **causes** \bar{f} if $\bar{g}_1, \dots, \bar{g}_k$ **cost** κ
 $\sigma \not\models \bar{g}_i$ for some $i = 1, \dots, k$

(b) D does not contain an effect statement A **causes** \bar{f} if $\bar{g}_1, \dots, \bar{g}_k$ **cost** κ

then $\sigma \models f$ if and only if $\Psi((A, \kappa), \sigma) \models f$

3. for every $A \in \mathcal{A}_c$, for every $f, g_1, \dots, g_k \in \mathcal{F}$ and for every $\kappa_1, \kappa_2 \in \mathbb{N}$, if one of the following conditions holds:

(a) if D contains two effect statements D of the form:

A **causes** \bar{f} if $\bar{g}_1, \dots, \bar{g}_k$ **cost** κ_1

A **causes** \bar{f} if $\bar{g}_1, \dots, \bar{g}_k$ **cost** κ_2

then $\kappa_1 = \kappa_2$

(b) D does not contain an effect statement A **causes** \bar{f} if $\bar{g}_1, \dots, \bar{g}_k$ **cost** κ_1 or does not contain A **causes** \bar{f} if $\bar{g}_1, \dots, \bar{g}_k$ **cost** κ_2

4. for every $A \in \mathcal{A}_c$, for every $f_1, \dots, f_n, g_1, \dots, g_l \in \mathcal{F}$, for every $\sigma \in \Sigma$ and for every $\kappa_1, \dots, \kappa_n \in \mathbb{N}$, the following condition holds:

(a) for all effect statements in D of the form:

A **causes** \bar{f}_1 if $\bar{g}_1, \dots, \bar{g}_l$ **cost** κ_1

\vdots

A **causes** \bar{f}_n if $\bar{g}_1, \dots, \bar{g}_l$ **cost** κ_n

if $\sigma \models g_i$ for all $i \in \{1, \dots, l\}$ then κ in $\Psi((A, \kappa), \sigma)$ is such that $\kappa = \kappa_1 + \dots + \kappa_n$

1.3 Query Language

Let $\mathcal{QL}_{\mathcal{CS}\mathcal{T}}$ be the query language corresponding to the $\mathcal{ADL}_{\mathcal{CS}\mathcal{T}}$.

1.3.1 Query statements

There were two types of queries defined for $\mathcal{QL}_{\mathcal{CS}\mathcal{T}}$, namely the **value queries** and **sufficiency queries**. The value queries answer the question of whether a given condition is satisfied after performing a sequence of actions. The sufficiency queries evaluate if a given cost is sufficient for execution of sequence of actions.

1. Value queries

The value query is defined in the following way:

1. γ after $A_1 \dots A_n$

Meaning: does the condition γ hold after executing the sequence of actions $A_1 \dots A_n$?

2. Sufficiency queries

The total cost of the execution of the sequence of actions $A_1 \dots A_n$ is defined as the sum of all costs associated with executed actions. Accordingly, the sufficiency query is defined in the following way:

1. **sufficient** κ in $A_1 \dots A_n$

Meaning: is the cost κ sufficient to execute the sequence of action $A_1 \dots A_n$?

1.3.2 Satisfiability of queries

Let D be an action domain. A query Q is a consequence of D , denoted by $D \models Q$, if and only if Q is true in every model of D :

1. Q is of the form: γ **after** $A_1 \dots A_n$

$D \models Q$ if and only if for any model $S = (\Psi, \sigma_0)$ of D , for every state $\sigma \in \Sigma$ and actions $A_1, \dots, A_n \in \mathcal{A}_c$ (i.e. A_1, \dots, A_n are all actions executed in that exact order), the following holds $\Psi(((A_1, \kappa_1), \dots, (A_n, \kappa_n)), \sigma) \models \gamma$ (i.e. the resulting state after execution of $A_1 \dots A_n$ is γ)

2. Q is of the form: **sufficient** κ in $A_1 \dots A_n$

$D \models Q$ if and only if for any model $S = (\Psi, \sigma_0)$ of D and for every state $\sigma \in \Sigma$ the sum of costs associated with performing the set of actions A_1, \dots, A_n is smaller or equal to κ (i.e. if a transition function Ψ for a particular set of actions $A_1, \dots, A_n \in \mathcal{A}_c$ and $\sigma \in \Sigma$ is associated with the costs $\kappa_1, \dots, \kappa_n$ then for $\Psi(((A_1, \kappa_1), \dots, (A_n, \kappa_n)), \sigma)$ total cost $K = \kappa_1 + \dots + \kappa_n$, the query is then satisfied iff $K \leq \kappa$)

1.4 Examples of dynamic systems

1.4.1 Titanic

We want to model the survival process of Jack and Rose after the Titanic ship sinks. We will use a class of dynamic systems to represent the actions and decisions that Jack and Rose could make in order to survive. We can assume that at the beginning of the scene, both Jack and Rose are in the water and not at the door. Let us define the possible actions:

1. Jack moves onto the door
2. Rose moves onto the door
3. Jack and Rose move onto the door
4. Jack stays in the water

Furthermore, the set of all possible considered states is: $\Sigma = \{\sigma_0, \sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_6, \sigma_7\}$, where:

$$\begin{aligned}
\sigma_0 &= \{\neg \text{Jack on the door}, \neg \text{Rose on the door}, \text{Jack alive}\} \\
\sigma_1 &= \{\text{Jack on the door}, \neg \text{Rose on the door}, \text{Jack alive}\} \\
\sigma_2 &= \{\neg \text{Jack on the door}, \text{Rose on the door}, \text{Jack alive}\} \\
\sigma_3 &= \{\text{Jack on the door}, \text{Rose on the door}, \text{Jack alive}\} \\
\sigma_4 &= \{\neg \text{Jack on the door}, \neg \text{Rose on the door}, \neg \text{Jack alive}\} \\
\sigma_5 &= \{\text{Jack on the door}, \neg \text{Rose on the door}, \neg \text{Jack alive}\} \\
\sigma_6 &= \{\neg \text{Jack on the door}, \text{Rose on the door}, \neg \text{Jack alive}\} \\
\sigma_7 &= \{\text{Jack on the door}, \text{Rose on the door}, \neg \text{Jack alive}\}
\end{aligned}$$

The scenario structure is as follows:

$$\begin{aligned}
&\mathbf{initially} \neg \text{Jack on the door} \\
&\mathbf{initially} \neg \text{Rose on the door} \\
&\mathbf{initially} \text{Jack alive} \\
&\text{Jack moves onto the door} \mathbf{causes} \text{Jack on the door} \\
&\quad \mathbf{if} \neg \text{Jack on the door}, \text{Jack alive} \mathbf{cost} 1 \\
&\text{Rose moves onto the door} \mathbf{causes} \text{Rose on the door} \\
&\quad \mathbf{if} \neg \text{Rose on the door} \mathbf{cost} 1 \\
&\text{Jack and Rose move onto the door} \mathbf{causes} \text{Rose on the door} \\
&\quad \mathbf{if} \neg \text{Rose on the door} \mathbf{cost} 1 \\
&\text{Jack and Rose move onto the door} \mathbf{causes} \text{Jack on the door} \\
&\quad \mathbf{if} \neg \text{Jack on the door}, \text{Jack alive} \mathbf{cost} 1 \\
&\text{Jack stays in the water} \mathbf{causes} \neg \text{Jack alive} \\
&\quad \mathbf{if} \neg \text{Jack on the door}, \text{Jack alive} \mathbf{cost} 0
\end{aligned}$$

Let us illustrate all possible transitions.

$$\begin{aligned}
\Psi((\text{Jack moves onto the door}, 1), \sigma_0) &= \sigma_1 \\
\Psi((\text{Rose moves onto the door}, 1), \sigma_0) &= \sigma_2 \\
\Psi((\text{Jack and Rose move onto the door}, 2), \sigma_0) &= \sigma_3 \\
\Psi((\text{Jack stays in the water}, 0), \sigma_0) &= \sigma_4 \\
\Psi((\text{Jack moves onto the door}, 1), \sigma_1) &= \sigma_1 \\
\Psi((\text{Rose moves onto the door}, 1), \sigma_1) &= \sigma_3 \\
\Psi((\text{Jack and Rose move onto the door}, 2), \sigma_1) &= \sigma_3 \\
\Psi((\text{Jack stays in the water}, 0), \sigma_1) &= \sigma_1 \\
\Psi((\text{Jack moves onto the door}, 1), \sigma_2) &= \sigma_3 \\
\Psi((\text{Rose moves onto the door}, 1), \sigma_2) &= \sigma_2 \\
\Psi((\text{Jack and Rose move onto the door}, 2), \sigma_2) &= \sigma_3 \\
\Psi((\text{Jack stays in the water}, 0), \sigma_2) &= \sigma_6
\end{aligned}$$

$\Psi((\text{Jack moves onto the door}, 1), \sigma_3) = \sigma_3$
 $\Psi((\text{Rose moves onto the door}, 1), \sigma_3) = \sigma_3$
 $\Psi((\text{Jack and Rose move onto the door}, 2), \sigma_3) = \sigma_3$
 $\Psi((\text{Jack stays in the water}, 0), \sigma_3) = \sigma_3$
 $\Psi((\text{Jack moves onto the door}, 1), \sigma_4) = \sigma_3$
 $\Psi((\text{Rose moves onto the door}, 1), \sigma_4) = \sigma_3$
 $\Psi((\text{Jack and Rose move onto the door}, 2), \sigma_4) = \sigma_6$
 $\Psi((\text{Jack stays in the water}, 0), \sigma_4) = \sigma_4$
 $\Psi((\text{Jack moves onto the door}, 1), \sigma_5) = \sigma_5$
 $\Psi((\text{Rose moves onto the door}, 1), \sigma_5) = \sigma_7$
 $\Psi((\text{Jack and Rose move onto the door}, 2), \sigma_5) = \sigma_7$
 $\Psi((\text{Jack stays in the water}, 0), \sigma_5) = \sigma_5$
 $\Psi((\text{Jack moves onto the door}, 1), \sigma_6) = \sigma_6$
 $\Psi((\text{Rose moves onto the door}, 1), \sigma_6) = \sigma_6$
 $\Psi((\text{Jack and Rose move onto the door}, 2), \sigma_6) = \sigma_6$
 $\Psi((\text{Jack stays in the water}, 0), \sigma_6) = \sigma_6$
 $\Psi((\text{Jack moves onto the door}, 1), \sigma_7) = \sigma_7$
 $\Psi((\text{Rose moves onto the door}, 1), \sigma_7) = \sigma_7$
 $\Psi((\text{Jack and Rose move onto the door}, 2), \sigma_7) = \sigma_7$
 $\Psi((\text{Jack stays in the water}, 0), \sigma_7) = \sigma_7$

Figure 1 presents all aforementioned transitions on a graph.

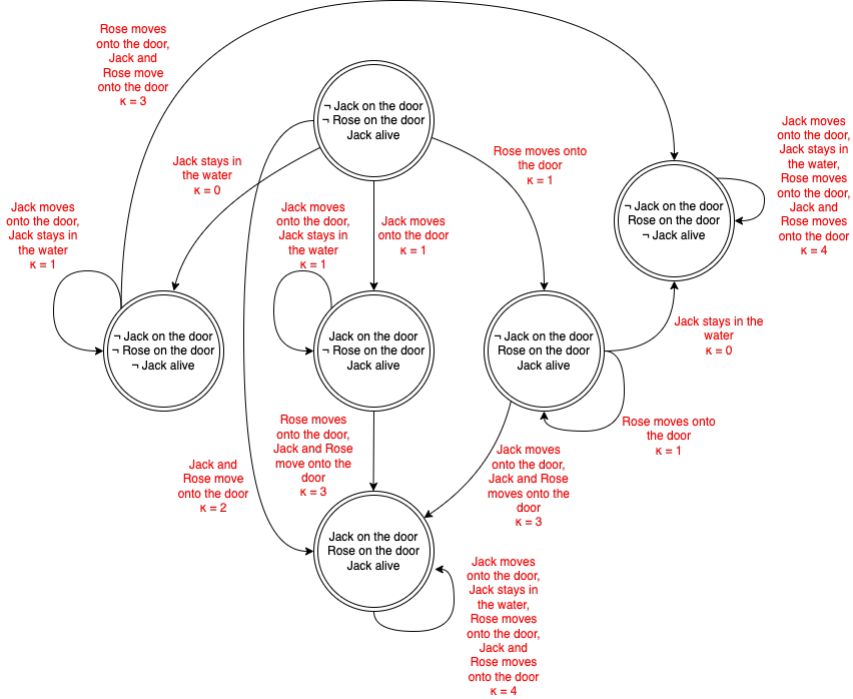


Figure (1) Titanic scenario - graph of all possible transitions

Let us demonstrate an example of asking queries using the structure of the query language $\mathcal{QL}_{\mathcal{CST}}$. Assume, that the initial state is σ_0 and the program $\mathcal{P} = (\text{Jack moves onto the door, Rose moves onto the door})$. Let us consider the following query statements:

1. *Jack alive* **after** (*Jack moves onto the door, Rose moves onto the door*)
2. **sufficient 1 in** (*Jack moves onto the door, Rose moves onto the door*)

The program execution will consist of the following transitions:

$$\begin{aligned}\Psi((\text{Jack moves onto the door}, 1), \sigma_0) &= \sigma_1 \\ \Psi((\text{Rose moves onto the door}, 1), \sigma_1) &= \sigma_3\end{aligned}$$

Hence, after the program execution, the state σ_3 will hold. Due to that, query number one will retrieve a positive response (i.e. *true*). However, notice that the total cost of the aforementioned transitions is equal to 2. Thus, cost 1 is insufficient to execute such a program and consequently, query number 2 will respond with *false*.

1.4.2 Shipping orders

There is an order that needs to be processed and delivered to the customer. The status of an order can be described by the following literals: received, packed, shipped, delivered and empty. Initially, the order is neither received nor packed, nor shipped, not delivered, and the shipping box for an order is empty. Placing an order makes it received. Packing an order makes it packed and also the shipping box becomes non-empty. Shipping an order makes it shipped and finally delivering it makes it delivered. Each of those aforementioned actions has an associated cost attached in case the initial conditions for performing a specific action are met.

The set of all considered states in this scenario is defined as $\Sigma = \{\sigma_0, \sigma_1, \sigma_2, \sigma_3, \sigma_4\}$, where:

$$\begin{aligned}\sigma_0 &= \{\neg\text{received}, \neg\text{packed}, \neg\text{shipped}, \neg\text{delivered}, \text{empty}\} \\ \sigma_1 &= \{\text{received}, \neg\text{packed}, \neg\text{shipped}, \neg\text{delivered}, \text{empty}\} \\ \sigma_2 &= \{\text{received}, \text{packed}, \neg\text{shipped}, \neg\text{delivered}, \neg\text{empty}\} \\ \sigma_3 &= \{\text{received}, \text{packed}, \text{shipped}, \neg\text{delivered}, \neg\text{empty}\} \\ \sigma_4 &= \{\text{received}, \text{packed}, \text{shipped}, \text{delivered}, \neg\text{empty}\}\end{aligned}$$

The structure of the scenario is as follows:

$$\begin{aligned}&\textbf{initially } \neg\text{received} \\ &\textbf{initially } \neg\text{packed} \\ &\textbf{initially } \neg\text{shipped} \\ &\textbf{initially } \neg\text{delivered} \\ &\textbf{initially } \text{empty} \\ &\textit{Place} \textbf{ causes } \textit{received} \textbf{ if } \neg\text{received} \textbf{ cost } 1 \\ &\textit{Pack} \textbf{ causes } \textit{packed} \textbf{ if } \textit{received}, \neg\text{packed} \textbf{ cost } 2 \\ &\textit{Pack} \textbf{ causes } \neg\text{empty} \textbf{ if } \textit{received}, \neg\text{packed} \textbf{ cost } 0 \\ &\textit{Ship} \textbf{ causes } \textit{shipped} \textbf{ if } \textit{packed}, \neg\text{shipped} \textbf{ cost } 3 \\ &\textit{Deliver} \textbf{ causes } \textit{delivered} \textbf{ if } \textit{shipped}, \neg\text{delivered} \textbf{ cost } 2\end{aligned}$$

Hence, the following states are considered in this case:

$$\begin{aligned}\Psi((\text{Place}, 1), \sigma_0) &= \sigma_1 \\ \Psi((\text{Pack}, 2), \sigma_0) &= \sigma_0 \\ \Psi((\text{Ship}, 3), \sigma_0) &= \sigma_0\end{aligned}$$

$$\Psi((\text{Deliver}, 2), \sigma_0) = \sigma_0$$

$$\Psi((\text{Place}, 1), \sigma_1) = \sigma_1$$

$$\Psi((\text{Pack}, 2), \sigma_1) = \sigma_2$$

$$\Psi((\text{Ship}, 3), \sigma_1) = \sigma_1$$

$$\Psi((\text{Deliver}, 2), \sigma_1) = \sigma_1$$

$$\Psi((\text{Place}, 1), \sigma_2) = \sigma_2$$

$$\Psi((\text{Pack}, 2), \sigma_2) = \sigma_2$$

$$\Psi((\text{Ship}, 3), \sigma_2) = \sigma_3$$

$$\Psi((\text{Deliver}, 2), \sigma_2) = \sigma_2$$

$$\Psi((\text{Place}, 1), \sigma_3) = \sigma_3$$

$$\Psi((\text{Pack}, 2), \sigma_3) = \sigma_3$$

$$\Psi((\text{Ship}, 3), \sigma_3) = \sigma_3$$

$$\Psi((\text{Deliver}, 2), \sigma_3) = \sigma_4$$

$$\Psi((\text{Place}, 1), \sigma_4) = \sigma_4$$

$$\Psi((\text{Pack}, 2), \sigma_4) = \sigma_4$$

$$\Psi((\text{Ship}, 3), \sigma_4) = \sigma_4$$

$$\Psi((\text{Deliver}, 2), \sigma_4) = \sigma_4$$

Figure 2 outlines all of the aforementioned transitions.

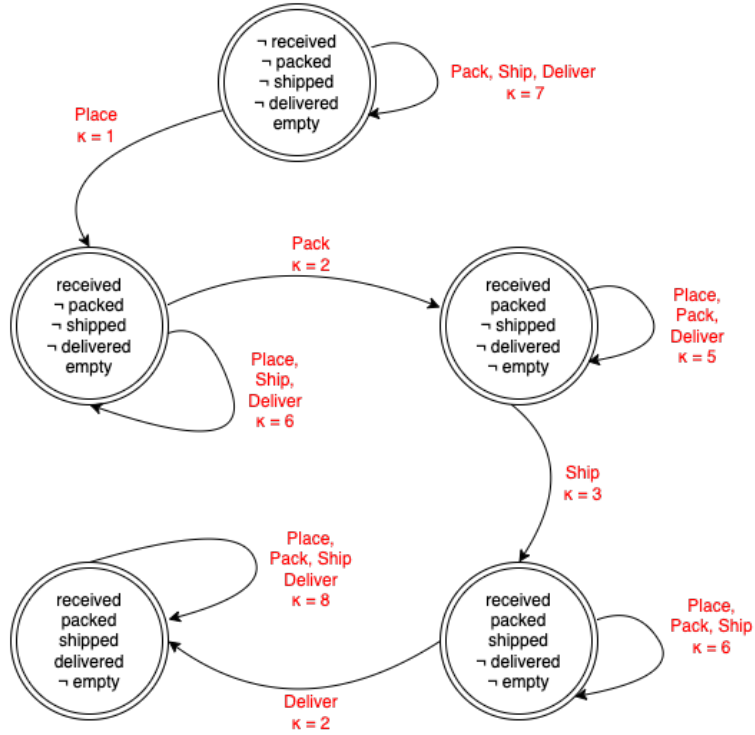


Figure (2) Shipping orders - all possible transitions

Similarly, as in the previous scenario, let us demonstrate an example of asking queries using the structure of the query language $\mathcal{QL}_{\mathcal{CST}}$. Assume, that the initial state is σ_0

and the program $\mathcal{P} = (\text{Place}, \text{Pack}, \text{Ship}, \text{Deliver})$. Let us consider the following query statements:

1. $\neg \text{delivered}$ **after** $(\text{Place}, \text{Pack}, \text{Ship}, \text{Deliver})$
2. **sufficient** 10 **in** $(\text{Place}, \text{Pack}, \text{Ship}, \text{Deliver})$

The program execution will consist of the following transitions:

$$\begin{aligned}\Psi((\text{Place}, 1), \sigma_0) &= \sigma_1 \\ \Psi((\text{Pack}, 2), \sigma_1) &= \sigma_2 \\ \Psi((\text{Ship}, 3), \sigma_2) &= \sigma_3 \\ \Psi((\text{Deliver}, 2), \sigma_3) &= \sigma_4\end{aligned}$$

As it can be observed, after the program execution, the state σ_4 will hold. Due to that, query number one will retrieve a negative response (i.e. *false*), as in the final state *delivered* holds. On the other hand, the second query will respond with *true* as the total cost 8 is smaller than 10.

1.4.3 Preparing for holidays

A family is preparing for the holidays. Due to the fact that they are traveling by plane, there is a limit on the weight of their luggage. In this regard, packing the next item is associated with some arbitrary cost corresponding to its weight. It is assumed that all necessary items have been packed by the family, and there is still some space available in the luggage. Due to this, they decided to take some additional equipment that would make their trip more entertaining. They considered a laptop, books, or sports equipment. When taking the laptop, the family must also take the charger. It is assumed that initially, the luggage is empty. Following, are the states considered in the system $\Sigma = \{\sigma_0, \sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, \sigma_6, \sigma_7, \sigma_8, \sigma_9\}$, where:

$$\begin{aligned}\sigma_0 &= \{\neg \text{has laptop}, \neg \text{has charger}, \neg \text{has books}, \neg \text{has sports equipment}\} \\ \sigma_1 &= \{\neg \text{has laptop}, \text{has charger}, \neg \text{has books}, \neg \text{has sports equipment}\} \\ \sigma_2 &= \{\neg \text{has laptop}, \text{has charger}, \text{has books}, \neg \text{has sports equipment}\} \\ \sigma_3 &= \{\neg \text{has laptop}, \text{has charger}, \neg \text{has books}, \text{has sports equipment}\} \\ \sigma_4 &= \{\text{has laptop}, \text{has charger}, \neg \text{has books}, \neg \text{has sports equipment}\} \\ \sigma_5 &= \{\text{has laptop}, \text{has charger}, \text{has books}, \neg \text{has sports equipment}\} \\ \sigma_6 &= \{\text{has laptop}, \text{has charger}, \neg \text{has books}, \text{has sports equipment}\} \\ \sigma_7 &= \{\text{has laptop}, \text{has charger}, \text{has books}, \text{has sports equipment}\} \\ \sigma_8 &= \{\neg \text{has laptop}, \neg \text{has charger}, \text{has books}, \neg \text{has sports equipment}\} \\ \sigma_9 &= \{\neg \text{has laptop}, \neg \text{has charger}, \text{has books}, \text{has sports equipment}\} \\ \sigma_{10} &= \{\neg \text{has laptop}, \text{has charger}, \text{has books}, \text{has sports equipment}\} \\ \sigma_{11} &= \{\neg \text{has laptop}, \neg \text{has charger}, \neg \text{has books}, \text{has sports equipment}\}\end{aligned}$$

Furthermore, the actions that were taken into account are:

1. Pack laptop
2. Pack books
3. Pack sports equipment
4. Pack charger

The system structure is described in the following way:

initially $\neg has\ laptop$
initially $\neg has\ charger$
initially $\neg has\ books$
initially $\neg has\ sports\ equipment$
Pack laptop **causes** *has laptop* **if** $\neg has\ laptop, has\ charger$ **cost** 10
Pack charger **causes** *has charger* **if** $\neg has\ charger$ **cost** 5
Pack books **causes** *has books* **if** $\neg has\ books$ **cost** 15
Pack sports equipment **causes** *has sports equipment* **if** $\neg has\ sports\ equipment$ **cost** 30

Hence, let us list all possible transitions:

$\Psi((Pack\ laptop, 10), \sigma_0) = \sigma_0$
 $\Psi((Pack\ charger, 5), \sigma_0) = \sigma_1$
 $\Psi((Pack\ books, 15), \sigma_0) = \sigma_8$
 $\Psi((Pack\ sports\ equipment, 30), \sigma_0) = \sigma_{11}$

 $\Psi((Pack\ laptop, 10), \sigma_1) = \sigma_4$
 $\Psi((Pack\ charger, 5), \sigma_1) = \sigma_1$
 $\Psi((Pack\ books, 15), \sigma_1) = \sigma_2$
 $\Psi((Pack\ sports\ equipment, 30), \sigma_1) = \sigma_3$

 $\Psi((Pack\ laptop, 10), \sigma_2) = \sigma_5$
 $\Psi((Pack\ charger, 5), \sigma_2) = \sigma_2$
 $\Psi((Pack\ books, 15), \sigma_2) = \sigma_2$
 $\Psi((Pack\ sports\ equipment, 30), \sigma_2) = \sigma_{10}$

 $\Psi((Pack\ laptop, 10), \sigma_3) = \sigma_6$
 $\Psi((Pack\ charger, 5), \sigma_3) = \sigma_3$
 $\Psi((Pack\ books, 15), \sigma_3) = \sigma_{10}$
 $\Psi((Pack\ sports\ equipment, 30), \sigma_3) = \sigma_3$

 $\Psi((Pack\ laptop, 10), \sigma_4) = \sigma_4$
 $\Psi((Pack\ charger, 5), \sigma_4) = \sigma_4$
 $\Psi((Pack\ books, 15), \sigma_4) = \sigma_5$
 $\Psi((Pack\ sports\ equipment, 30), \sigma_4) = \sigma_6$

 $\Psi((Pack\ laptop, 10), \sigma_5) = \sigma_5$
 $\Psi((Pack\ charger, 5), \sigma_5) = \sigma_5$
 $\Psi((Pack\ books, 15), \sigma_5) = \sigma_5$
 $\Psi((Pack\ sports\ equipment, 30), \sigma_5) = \sigma_7$

 $\Psi((Pack\ laptop, 10), \sigma_6) = \sigma_6$
 $\Psi((Pack\ charger, 5), \sigma_6) = \sigma_6$
 $\Psi((Pack\ books, 15), \sigma_6) = \sigma_7$
 $\Psi((Pack\ sports\ equipment, 30), \sigma_6) = \sigma_6$

 $\Psi((Pack\ laptop, 10), \sigma_7) = \sigma_7$
 $\Psi((Pack\ charger, 5), \sigma_7) = \sigma_7$
 $\Psi((Pack\ books, 15), \sigma_7) = \sigma_7$
 $\Psi((Pack\ sports\ equipment, 30), \sigma_7) = \sigma_7$

 $\Psi((Pack\ laptop, 10), \sigma_8) = \sigma_8$

$$\begin{aligned}\Psi((\text{Pack laptop}, 10), \sigma_9) &= \sigma_9 \\ \Psi((\text{Pack charger}, 5), \sigma_9) &= \sigma_{10} \\ \Psi((\text{Pack books}, 15), \sigma_9) &= \sigma_9 \\ \Psi((\text{Pack sports equipment}, 30), \sigma_9) &= \sigma_9\end{aligned}$$

$$\begin{aligned}\Psi(\text{(Pack laptop, 10)}, \sigma_{11}) &= \sigma_{11} \\ \Psi(\text{(Pack charger, 5)}, \sigma_{11}) &= \sigma_3 \\ \Psi(\text{(Pack books, 15)}, \sigma_{11}) &= \sigma_9 \\ \Psi(\text{(Pack sports equipment, 30)}, \sigma_{11}) &= \sigma_{10}\end{aligned}$$

13

Finally, let us demonstrate queries for the above example. Assume, that the initial state is σ_0 and the program $\mathcal{P} = (\text{Pack sports equipment}, \text{Pack books}, \text{Pack laptop})$. Let us consider the following query statements:

1. *has laptop* **after** (*Pack sports equipment*, *Pack books*, *Pack laptop*)
2. **sufficient** 40 **in** (*Pack sports equipment*, *Pack books*, *Pack laptop*)

The program execution will consist of the following transitions:

$$\Psi(\text{Pack sports equipment}, 30, \sigma_0) = \sigma_{11}$$

$$\Psi(\text{Pack books}, 15, \sigma_{11}) = \sigma_9$$

$$\Psi(\text{Pack laptop}, 10, \sigma_9) = \sigma_9$$

It can be noted that the last transition is associated with a cost 10 although the state doesn't change. Due to the fact that preconditions that are required to successfully execute the *Pack laptop* action are not met, the action has empty effects. After the program execution, the last state is, therefore, σ_9 . Hence, the response of the first query will be *false*. The response to the second query will also be *false*, as the total cost of the program execution is 55 which is greater than 40.

2 Technical description

2.1 Introduction

The goal of the application was to provide the user with the handy and intuitive tool that could be used to model the behaviour of different dynamic systems that adhere to a specific set of requirements (as mentioned in the introduction). These assumptions were taken into account during the implementation. It is important to point out that they need to be taken into consideration by the user, who shall be aware of the potential limitations of the modelled systems. Nevertheless, within those defined assumptions, the application's features can facilitate the exploration and analysis of the behaviour of this subset of dynamic systems.

2.2 Technology

The application was implemented as a web application using **Windows Forms**, which is an open-source UI framework for building Windows desktop app that enables quick delivery and ensures intuitive and visually appealing design. In the development process, this framework allowed the creation of a user-friendly interface that provides a way of managing data such as *fluents* and *actions*. In addition to that, components from this library were used to produce a convenient view responsible for program creation. The desktop application also allows displaying the graphs associated with the created program. It was done with the help of the **GoDiagram** component.

2.3 Classes

The definition of the data type in the web application was mostly done with the help of classes from external libraries, such as **Syncfusion**. However, to ensure more readable and understandable code, five classes were added, mostly associated with the statements created by the user and the state of the program used in graph creation:

- *State*: It represents a state and holds information about its fluents and cost. The *literals* attribute is a list of strings representing the fluents associated with the

state. The class also defines an *equals* method that compares two State objects for equality based on their fluents.

- *Statement*: It serves as a base class for different types of statements. It has two attributes: *type*, which represents the type of the statement, and *text*, which holds the markdown content of the statement.
- *InitiallyStatement*: This class inherits from the Statement class. It represents statements of the type: **initially** \bar{f} . The *type* attribute is set to the enumerable value *INITIALLY*. It also has an additional attribute *condition* that indicates which fluents hold in the initial state of the system.
- *CausesStatement*: This class inherits from the Statement class. It represents statements of the type: A **causes** \bar{f} if $\bar{g}_1, \dots, \bar{g}_k$ **cost** κ . The *type* attribute is set to the enumerable value *CAUSES*. It has additional attributes: *action* represents the action that is to be executed, *postcondition* denotes a literal that will hold in the resulting state, *precondition* represents a condition that has to be met for an action to execute (if this condition is not met the action has empty effects) and *cost* that represents the cost associated with the action.
- *AfterStatement*: This class inherits from the Statement class. It represents statements of the type: \bar{f} **after** A_1, \dots, A_n . The *type* attribute is set to the enumerable value *AFTER*. It has additional attributes: *postcondition*, which represents the literal that will hold after actions execution, and *actions*, which is a list of strings representing the sequence of actions performed.

2.4 Sections

The applications consist of the following sections, each of which provides the user with the part of the functionality needed to fully simulate the behaviour of the desired dynamic system:

- **Fluents and actions.** This section allows users to manage fluents and actions. The application provides the following functionalities:
 - **Adding Fluents and Actions:** Provides text input fields and buttons for users to add new fluents and actions to the application. Upon clicking the *Add* button, the respective inputs are processed and stored (both fluents and actions are held as a list of strings)
 - **Fluents View:** Displays the added fluents in a view. Each fluent is shown along with a checkbox. The user is able to mark multiple fluents and delete them using the dedicated *Delete* button. There is also an option to delete all fluents at once.
 - **Actions View:** Displays the added actions in a view. Each action is shown along with a checkbox that works in the same manner as in the case of the fluents. There is also an option to delete all actions at once.
- **Add Statement.** This part allows users to create and manage an action domain consisting of a set of value and effect statements. The following functionalities are provided by the application:
 - The application provides a panel for adding statements, where users can select the statement type (definition of the initial state is represented by *Initially statement*, effect statement is represented by *Effect statement*, value statement is represented by *Value statement*) and input relevant details based on the selected type.

- For *Initially* statements, users can select a literal that holds in the initial state (the set of fluents from which the user chooses is provided to him based on the input values on the previous page). The user can either select the literal from the list or write it by hand.
 - For *Effect* statements, users can select an action, postcondition and precondition (similarly, the list of available actions and fluents is provided by the application, and users can also write literals/actions by hand).
 - For *Value* statements, users can select a fluent that will hold in the resulting state and the set of actions causing the change (the list of available actions and fluents is also provided by the application).
 - Users can add a statement by clicking the *Add* button in the *ADD STATEMENT* section, and the inputted statement is processed and added to the session state (each statement is an object of the class presented above).
 - The Action Domain panel displays the added statements, where each statement is shown along with a button to delete it. The user can also delete here all statements at once or display the visualization of a given action domain.
- **Execute program.** This part allows users to specify the sequence of actions and the initial state and execute the program (based on the previously defined action domain), in order to see what will be its total cost and final state
 - The user can specify the sequence of actions separated by semicolons and the initial state (the possibility of the selection of the initial state was added to the fact that some domains can have more than 1 model)
 - After pressing the *EXECUTE PROGRAM* button, the given action sequence will be executed and the final state as well as the total cost of the program execution will be returned (displayed next to the *FINAL STATE* and *FINAL COST* labels)
- **Visualisation window.** This part allows users to see the graph depicting all possible states of the models, together with all possible transitions between them. The displayed graph has the following properties:
 - Each node of the graph represents the state of the system that is reachable in the system from a given initial state.
 - The arrows between the nodes illustrate the transitions between the particular states as a result of an execution of a particular action.
 - Each arrow is labelled with information about the action performed and the cost of transition between those two states.
 - The initial state is marked green colour. If there are multiple green-coloured nodes, then it means that there are several models in a given domain, and each green node represents the initial state of every one of them.
 - The user is able to drag the nodes and links in order to get a better view of the graph.
- **Query and Query result.** This part allows users to insert query statements and see below query result.
 - The view provides a query functionality that allows users to perform two types of queries: *Value* and *Cost* queries.
 - For a *Value* query, it checks if a provided condition is satisfied after the execution of the sequence of actions from the given initial state

- For a *Cost* query, it compares the provided cost value with the overall cost of the execution of the sequence of actions.
- The query results are displayed in the form of the pop-up for the users with the corresponding message (QUERY CORRECT/QUERY INCORRECT) and blue information indicator when query is satisfied or red cross otherwise

3 User guide

3.1 Application

This application is a practical solution for executing the “Actions with Costs” task. With this program, users have the ability to create scenarios to test the Action Description Language $AD\mathcal{L}_{CST}$ and its corresponding Query Language $Q\mathcal{L}_{CST}$. These languages enable the description of specific groups of dynamic systems, considering concepts related to costs.

3.2 Instruction how to run application

For proper program execution, Windows OS is required. To run the application, User should double click on file

`actions_with_costs.exe`

3.3 Application functionalities

The application has one main window where the User can define the action domains and execute the programs. In order to see the visualization result, the User should click the button *OPEN VISUALIZATION* that brings up a separate window with a state diagram.

Figure (4) Main window

- **Add Fluents and Actions**

To add Fluent, User should insert the name of the fluent in the text box under *ADD FLUENT* label and click *ADD* button below (or press Enter).

Figure (5) Setting fluent

To add Action, User should insert the name of the action in the text box under *ADD ACTION* label and click *ADD* button below (or press Enter).

Figure (6) Setting action

Inserted items will appear in the text boxes on the right-hand side, for both Fluents and Actions separately.

User is able to remove Fluent or Action, by first, selecting all fluents that are to be removed, and then clicking *DELETE* button to remove them from the list. The user is also able to remove all fluents (or actions) at once, by pressing *REMOVE ALL* button.

Figure (7) Fluent and actions tables

- **Add Statements to Action Domain**

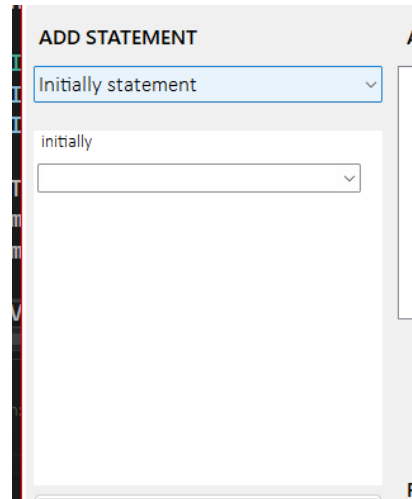
After selecting fluents and actions, the User can go to the next section, *ADD STATEMENT*, where the User is able to create statements.

Under *ADD STATEMENT* there is a drop-down list where the User can select between *Initially statement*, *Value statement*, *Effect statement*.

Figure (8) Setting statements

To add a statement User should click *ADD STATEMENT* button at the bottom of the section.

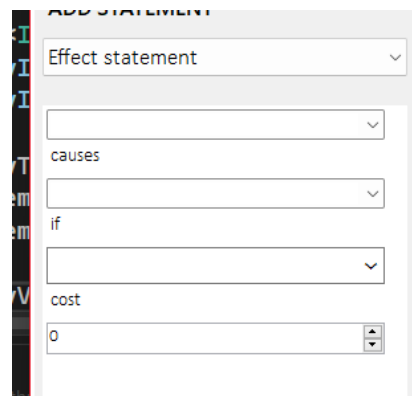
- * *Initially statement* - choosing it from a drop-down list will display the selector field to provide literals. The User can select the literals either from the list, or provide them by hand.



The screenshot shows a dialog box titled "ADD STATEMENT". At the top, there is a dropdown menu with "Initially statement" selected. Below this, there is a text input field with the label "initially" to its left. The input field is currently empty.

Figure (9) Setting Initially statement

- * *Effect statement* - choosing it from a drop-down list will display 4 input fills by which the User can:
 - select Action on the top of the label *CAUSES*
 - select resulting fluents on the top of the label *IF*
 - select conditional fluents on the top of the label *COST*
 - Insert number that corresponds to the cost assigned to the fluent



The screenshot shows the "ADD STATEMENT" dialog box with "Effect statement" selected in the dropdown. Below the dropdown are four input fields:

- A dropdown menu for the "causes" label.
- A dropdown menu for the "if" label.
- A dropdown menu for the "cost" label.
- A numeric input field with the value "0" and up/down arrow buttons.

Figure (10) Setting Effect statement

- * *Value statement* - choosing it from a drop-down list will display two input fields, where the User is able to select literal on the top of the label *AFTER* and specify the action (underneath the label *AFTER*)

Figure (11) Setting Value statement

All statements specified in the forms described above, will appear in the *ACTION DOMAIN* section. There, the user can select and remove statements (using *DELETE* button), remove all statements (using *DELETE ALL* button) or display the visualization window by pressing *OPEN VISUALIZATION* button).

Figure (12) Action Domain section

- **Program specification and its execution**

The user is able to execute the program that will use a specified action domain. To do so, the user has the possibility to specify the sequence of actions that are to be executed and the initial state from which the program will start the execution. To do so, in checkboxes named *Type – in – actions* and *Choose – state* user can respectively provide a sequence of actions and the state from which execution begins.

Figure (13) Choose action and Choose state for program execution

In order to execute the program, the User should click on the button *EXECUTE PROGRAM*. The results of its execution, including the resulting state and the cost are shown below.

FINAL STATE: ~alive, ~loaded FINAL COST: 5

Figure (14) Final state and cost

- **Visualisation**

The user can also open a visualisation of an action domain by clicking on the *OPEN VISUALISATION* button. The resulting graph depicts all reachable states (given possible initial states), together with all possible transitions between them. It has the following properties: each node of the graph represents the state of the system, and the number of nodes is equal to the number of all possible combinations of literals corresponding to fluents minus the states that have to be excluded because of the conditions imposed by statements from the action domain. The arrows between the nodes illustrate the transitions between the particular states as a result of an execution of a particular action. Each arrow is labelled with information about the action performed and the transition cost between those two states. Finally, the initial state is marked as a green node.

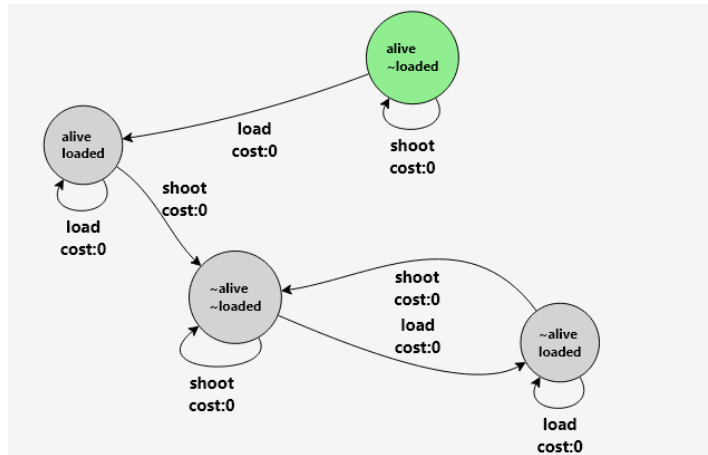


Figure (15) State graph

- **Query**

The user can post queries using the 'Query' side panel.

QUERY

SELECT QUERY TYPE:

Value query

after

from initial state

Figure (16) Query side panel - Value Query

QUERY

SELECT QUERY TYPE:

Cost query

sufficient

0

after

from initial state

Figure (17) Query side panel - Cost Query

There are 2 types of queries available: 'Value' and 'Cost', as shown above. Value query follows the idea of the value statement in the action domain. The user is able to select a fluent from the drop-down box that should appear in the output state after performing actions from the set initial state. The validators prevent an incorrect query execution should it contain corrupt/incorrect input data. The cost query is similar to the value query, however it checks whether the provided cost is sufficient to execute the list of actions from the specified initial state instead.

After successfully executing the query, a popup message with the result will show as follows:

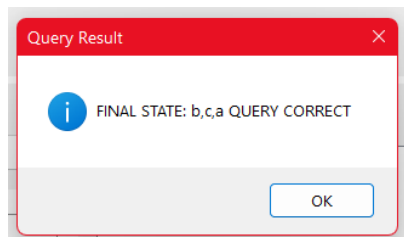


Figure (18) Correct query popup

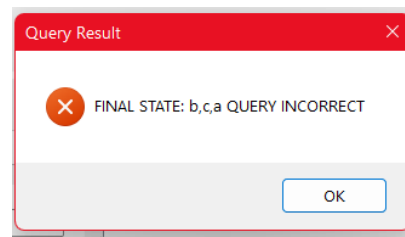


Figure (19) Incorrect query popup

For 'Value' queries, the popup provides the message about the final state of the system when executing the query from provided starting conditions. For the 'Cost' query, the final cost is provided instead.

4 Test description

The application tests were performed in two steps: 1) testing the user experience and 2) testing the semantic and syntax correctness of the prepared application.

4.1 Testing user experience (UX/UI testing)

The aim of UX/UI tests was to evaluate whether all of the functionalities offered by the application are working accordingly to their intended use. Specifically, this part of testing puts the emphasis on the evaluation of all available application components and controls, such as buttons, input fields, selectors etc. The list of the performed tests includes:

1. Verification of *Add Fluent/Add Action* section:

- (a) Verification if the user cannot add two fluents (or actions) with identical names
- (b) Verification if the user cannot add fluent beginning with \sim sign (since it is reserved to indicate fluent negation)
- (c) Verification if the user cannot add empty fluents (or actions)
- (d) Verification if the user can add fluents (or actions) using both *Add* button and after pressing the *Enter* key
- (e) Verification if the *Add* button is disabled when there are no fluents (or actions) that are to be added
- (f) Verification if the user can remove one or more fluents (or actions)
- (g) Verification if the user can remove all fluents (or actions) at once
- (h) Verification if the *Remove All* and *Remove* buttons are properly disabled when there are no fluents (or actions) that are to be removed
- (i) Verification if the user is prompted if she/he attempts to remove fluents (or actions) that are currently used in some statements of the action domain (in such case, the user should also be offered to automatically clear the action domain statements)

2. Verification of *Add Statement* section:

- (a) Verification if the user can select between three types of available statements: *Initially*, *Value (After)* and *Effect (Causes)* statements
- (b) Verification if upon selecting a given statement type, proper input-selector fields are displayed for the user
- (c) Verification if the user cannot add two equivalent statements
- (d) Verification of each type of statement:
 - *Initially*:
 - Verification if the user cannot add initially statement with no literals selected
 - Verification if the user cannot add initially statement that contains a complementary literal to one of the initially statements already existing in the domain (e.g. *alive* and *alive* at the same time)
 - Verification if the user can provide literals by both selecting them from the list and writing them by hand
 - Verification if the user cannot add literal (provided by hand) with a name of fluent that was not previously specified
 - *Value*:
 - Verification if the user cannot add a value statement without specifying the action and the postcondition literal (the abbreviation of value statement has been specified separately, therefore, specifying action here is obligatory)

- Verification if the user can select more than one action
- Verification if the user cannot specify two value statements with complementary literals that would hold after the same action sequence (i.e. preventing model inconsistency)
- Verifying if the user can provide postcondition literal both by selecting it from the list and by writing it by hand
- Verification if the literals provided by the user in the postcondition part correspond to existing literals
- *Effect*:
 - Verification if the user can provide postcondition and action both by hand and by selecting them from corresponding lists
 - Verification if the user cannot add the effect statement without specifying the action or the postcondition
 - Verification if the user cannot provide actions or literals that do not correspond to fluents or actions specified in the *Add Fluent/Add Action* section
 - Verification if the user cannot add two effect statements of the same action that lead to complementary literals when having the same preconditions (i.e. preventing model inconsistency)
 - Verification if the user cannot provide two effect statements with the same action, postcondition and preconditions, but different costs (i.e. preventing model inconsistency)
 - Verification if the user cannot give negative cost (i.e. cost that violates the assumptions)
- (e) Verification if the user will be prompted with a corresponding message when she/he provides incorrect input (i.e. inputs mentioned above)

3. Verification of *Action Domain* section:

- (a) Verification if all added statements appear in their corresponding order in the action domain field
- (b) Verification if the statements can be selected (one or many) and deleted using *Delete* button
- (c) Verification if the user can delete all statements using *Delete All* button
- (d) Verification if, when the provided action domain is inconsistent, the red text “*Domain is inconsistent*” is being displayed
- (e) Verification if the buttons are disabled when no statements are selected (or provided)
- (f) Verification if the *Open Visualization* button is disabled when inconsistent model is provided
- (g) Verification if after pressing *Open Visualization* button, the window containing *State Diagram* of action domain model(-s) is displayed (with possible initial states indicated using green colour)
- (h) Verification if the user cannot specify the program to be executed when the model is inconsistent or when no statements are provided
- (i) Verification if the user can specify the sequence of actions that are to be executed in the program using “,” as a separator
- (j) Verification if the input of action sequence is properly parsed

- (k) Verification if the user can select among all available initial states (including restriction of states put by *after* and *initially* statements) when specifying from which state the program should start
- (l) Verification if the user can press *Execute Program* button only when both aforementioned fields are filled
- (m) Verification if the executed program yields the correct final state and presents the correct overall cost

4. Verification of *Query* section:

- (a) Verification if the user can select among two available query options: *Value* and *Cost* queries
- (b) Verification if the selection of appropriate query displays all its input and selector fields
- (c) Verification if the *Execute Query* button is disabled when the model is inconsistent or when there are no statements in the action domain
- (d) Verification of each query type:
 - *Value*:
 - Verification if the user can provide literals, both by hand or from the list, and if all available literals are offered in the list
 - Verification if the user can provide the sequence of actions (i.e. program) by hand using “,” as separators between consecutively executed actions
 - Verification if the actions and literals provided by the user correspond to the fluents and actions given in *Add Fluent/Add Action* section
 - Verification if the user can repeat some actions in the provided program
 - Verification if the selector of initial states (i.e. state from which the program will be executed - necessary when the action domain has more than one model) contains all initial states corresponding to the states obtained from *Action Domain* section
 - Verification if the user cannot execute the query if not all fields are provided
 - Verification if the query yields a correct result
 - *Cost*:
 - Verification if the user cannot give a cost that is smaller than 0
 - Verification if the user can specify the program in the same manner as in the case of the *Value* query
 - Verification if the user can specify the initial state in the same manner as in the *Value* query
- (e) Verification if, in the case of any violation of the above conditions, the user is prompted with a warning message

5. Verification of *Menu* panel:

- Verification if the user can clear all input provided in the application using *Clear All* button

All of the above tests yielded successful results, hence, assuring that the application is working as intended. Therefore, the tests could proceed to the next phase, which was focused on the evaluation of how well the application is performing in real-case scenarios.

4.2 Syntax and semantic correctness testing

The main aim of the second testing phase was to evaluate if the application is adhering correctly to the model consistency. In particular, these tests focused on executing example scenarios in which the domain models could be either consistent or not and evaluating if:

1. Application can properly detect the inconsistency of the model
2. Application is visualizing the state graphs correctly
3. Based on a given action domain, program specification and initial state, the application can correctly calculate the final state and final cost

Therefore, the remaining part of the testing section will focus on describing various test scenarios, the simple and complex ones, that were used in performed tests.

4.3 Test 1 - Yale Shooting Problem

The *Yale Shooting Problem* was used as the first testing scenario since it is one of the most simple cases and, hence can give an indicator if the application works correctly on the “happy path”. To recall, the formulation of the scenario is as follows:

There is a shooter Bill and a turkey Fred. Initially Fred is alive and Bill has an unloaded gun. Bill can perform two actions: loading the gun (LOAD) and shooting the gun (SHOOT). Loading the gun makes the gun loaded, while shooting the gun makes it unloaded and, in addition, Fred is not alive anymore, provided that the gun was loaded.

The scenario was represented in the application as presented in Figure 20 (some arbitrary costs were also added). The first thing that could be observed was that there was no

The screenshot shows a web application titled "Actions with costs". It has a "CLEAR ALL" button at the top left. The interface is divided into several sections:

- ADD FLUENT:** A text input field and an "ADD" button.
- FLUENTS:** A list containing "loaded" and "alive" with checkboxes, and a "REMOVE ALL" button.
- ADD ACTION:** A text input field and an "ADD" button.
- ACTIONS:** A list containing "LOAD" and "SHOOT" with checkboxes, and a "REMOVE ALL" button.
- ADD STATEMENT:** A dropdown menu with "Initially statement" selected, and an "initially" input field.
- ACTION DOMAIN:** A list of statements with checkboxes: "Initially -loaded", "Initially alive", "LOAD causes loaded cost 2", "SHOOT causes -loaded cost 1", and "SHOOT causes -alive if loaded cost 3". There are "DELETE" and "DELETE ALL" buttons.
- SPECIFY PROGRAM:** A section with "ACTION SEQUENCE" (a text input), "INITIAL STATE" (a dropdown), and an "EXECUTE PROGRAM" button.
- QUERY:** A section with "SELECT QUERY TYPE:" (a dropdown set to "Cost query"), a "sufficient" input field, an "after" dropdown, and a "from initial state" dropdown. There is an "EXECUTE QUERY" button at the bottom right.

At the bottom, there are fields for "FINAL STATE:" and "FINAL COST:".

Figure (20) Representation of YSP simple scenario

indication of model inconsistency, hence, the application was able to properly recognize the consistent model. The state graph obtained for this scenario is presented in Figure 21. As can be seen, the visualization contains correct transitions. The initial state is marked on the green colour and corresponds to the state restricted with *initially* statements. The program execution was evaluated for the action sequence $\{LOAD\}$ and gave the result presented in Figure 22, which is correct (as can also be verified on the graph visualization).

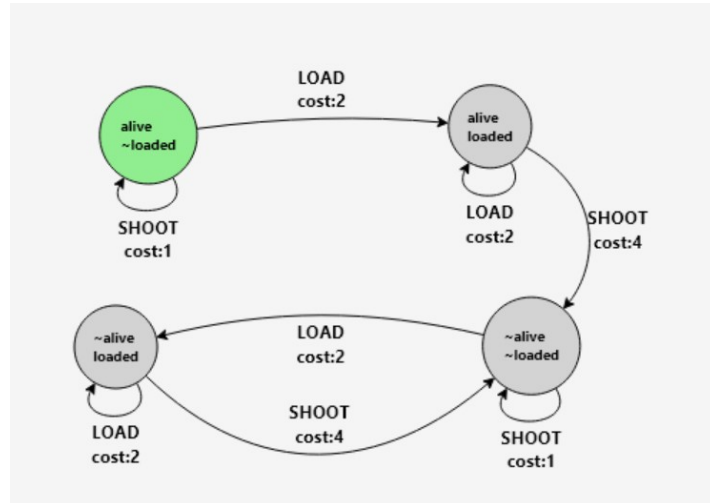


Figure (21) State diagram of YSP simple scenario visualization

SPECIFY PROGRAM

ACTION SEQUENCE

LOAD

INITIAL STATE

~loaded,alive

EXECUTE PROGRAM

FINAL STATE: loaded,alive

FINAL COST: 2

Figure (22) Result of program execution in YSP simple scenario for action sequence LOAD

4.4 Test 2 - YSP modified with after statement

The aim of the second test was to evaluate if the application will properly take into consideration the *after* statements. The test compared the results obtained after executing two similar YSP-based scenarios. The scenarios are as follows:

Scenario 1:

There is a shooter Bill and a turkey Fred. Initially, Fred is alive. Bill can perform two actions: loading the gun (LOAD) and shooting the gun (SHOOT). It is known that the shooting action makes the gun unloaded, and if the gun was prior loaded, then it kills Fred.

Scenario 2:

There is a shooter Bill and a turkey Fred. Initially, Fred is alive. Bill can perform two actions: loading the gun (LOAD) and shooting the gun (SHOOT). It is known that the shooting action makes the gun unloaded, and if the gun was prior loaded, then it kills Fred. Additionally, it was observed that after shooting, Fred was NOT alive.

The representation of both scenarios in the program can be seen in Figure 23 and Figure 24. As one can observe, both action domains are consistent, which is correct. The second action domain differs from the first one by the act that it contains the *after* statement. In both action domains, *initially* specifies only part of the initial state. However, as in the second case, there is an observation, it gives the indicator about the possible initial states, restricting their number. The state graphs are presented in

Figure (23) Representation of the YSP-based scenario without after statement

Figure 25 and Figure 26. The first visualization presents two possible models - in the first one, the initial state contains the negation of the fluent *loaded*, while the second one has an initial state with positive *loaded*. It corresponds to the defined action domain since there was no information about the initial state of that fluent. However, the second visualization presents only a single possible model. It comes from the fact that the *after* statement indicated that the *SHOOT* action caused Fred to be killed. The only possibility of that would be if the gun was prior loaded, hence only such state was taken under consideration. In addition, for the second scenario, the program was executed with action sequence $\{SHOOT\}$. The result (presented in Figure 27) was once again as expected

Actions with costs

CLEAR ALL

ADD FLUENT

ADD

FLUENTS

☐ loaded

☐ alive

REMOVE ALL

ADD ACTION

ADD

ACTIONS

☐ SHOOT

☐ LOAD

REMOVE ALL

ADD STATEMENT

Value statement

after

ADD

ACTION DOMAIN

☐ Initially alive

☐ SHOOT causes ~loaded cost 1

☐ SHOOT causes ~alive if loaded cost 0

☐ ~alive after SHOOT

DELETE

DELETE ALL

OPEN VISUALIZATION

SPECIFY PROGRAM

ACTION SEQUENCE

INITIAL STATE

FINAL STATE:

FINAL COST:

EXECUTE PROGRAM

QUERY

SELECT QUERY TYPE:

Value query

after

from initial state

EXECUTE QUERY

Figure (24) Representation of the YSP-based scenario with after statement

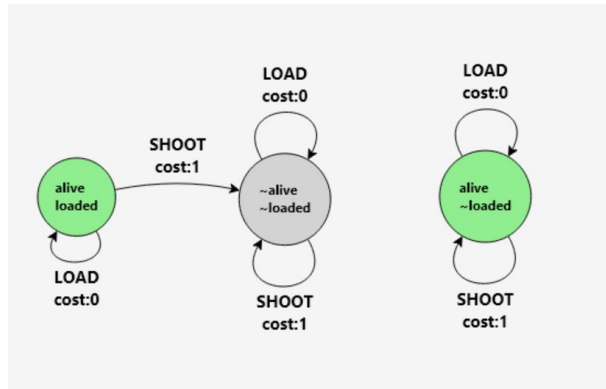


Figure (25) State diagram of YSP-based scenario without after statement

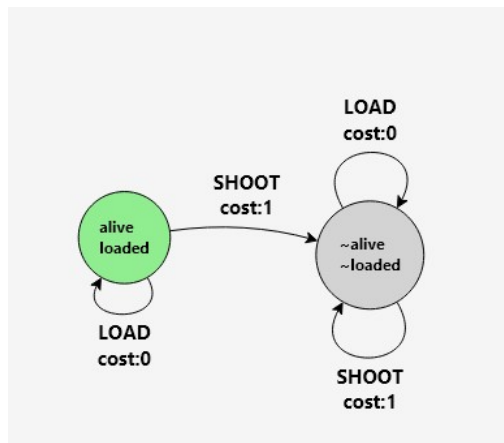


Figure (26) State diagram of YSP-based scenario with after statement

Figure (27) Result of program execution in YSP-based scenario with after statement for action sequence SHOOT

4.5 Test 3 - Action domain inconsistency testing

The aim of the tests from this group was to evaluate if the application could properly recognize the inconsistency in the action domain. Therefore, the following two test scenarios were used:

Scenario 1:

There is a student who did not have breakfast and therefore is initially hungry. The student is commuting to the university. After arriving at the place, the student is not hungry anymore.

Scenario 2:

There is a painter who aims to paint the wall. The painter can start the action PAINT, making the wall painted if he has some paint left. Initially, the painter had some paint left. It was observed that after painting, the wall had not been painted.

The representation of the above scenarios in the application is presented in Figure 28 and Figure 29 correspondingly.

Figure (28) Representation of the scenario with hungry student

As can be seen, in both examples, the application returned the information that the action domain is inconsistent. In the case of the hungry student, there were no statements which would specify the effects of the action *GO TO UNIVERSITY*. Therefore, as the

assumptions of this class of dynamic systems say that all effects of the actions are specified, it can be stipulated that the *GO TO UNIVERSITY* action has an empty effect. However, the execution of it led to a state which changed the initial state of the given fluent, and therefore such a domain is inconsistent. In the second example, the effect of the action *PAINT*, which could be executed in all initial states, as in all of them the painter *hasPaint*, had the complementary effect to the one which was observed (i.e. specified in *after* statement). Therefore, once again, the action domain is indeed inconsistent, which was correctly recognized by the application.

Figure (29) Representation of the scenario with painter

4.6 Test 4 - Titanic example

The last two tests considered more complicated scenarios which were previously presented in the theoretical examples section (specifically, the scenario of Titanic and Delivery). The first scenario of the story is as follows:

We want to model the survival process of Jack and Rose after the Titanic ship sank. We will use a class of dynamic systems to represent the actions and decisions that Jack and Rose could make in order to survive. We can assume that at the beginning of the scene, both Jack and Rose are in the water and not at the door.

The fluents that were considered in the scenario are:

1. **Jack on door**
2. **Rose on door**
3. **Jack alive**

The list of the actions considered in this scenario was:

1. **Jack moves**
2. **Rose moves**
3. **Jack and Rose move**

4. Jack stays in the water

The program defined using the application is presented in Figure 30 (the same definition as in the example from theoretical section). The visualization of the domain is presented

The screenshot shows the 'Actions with costs' application window. It has a title bar with a close button. The interface is divided into several sections:

- CLEAR ALL**: A button at the top left.
- ADD FLUENT**: A section with a text input field and an 'ADD' button.
- FLUENTS**: A list of fluents with checkboxes: 'Jack on door', 'Rose on door', and 'Jack alive'. There is a 'REMOVE ALL' button above the list.
- ADD ACTION**: A section with a text input field and an 'ADD' button.
- ACTIONS**: A list of actions with checkboxes: 'Jack moves', 'Rose moves', 'Jack and Rose move', and 'Jack stays in the water'. There is a 'REMOVE ALL' button above the list.
- DELETE**: A button located below the 'FLUENTS' and 'ACTIONS' lists.
- ADD STATEMENT**: A section with a dropdown menu for 'Effect statement', a text input field, and a 'causes' dropdown menu. Below this is an 'if' dropdown menu and a 'cost' dropdown menu with a value of '0'.
- ACTION DOMAIN**: A list of statements with checkboxes: 'initially ~Jack on door', 'initially ~Rose on door', 'initially Jack alive', 'Jack moves causes Jack on door if ~Jack', 'Rose moves causes Rose on door if ~Rose', 'Jack and Rose move causes Rose on door', and 'Jack and Rose move causes Jack on door'. There are 'DELETE' and 'DELETE ALL' buttons to the right.
- SPECIFY PROGRAM**: A section with 'ACTION SEQUENCE' and 'INITIAL STATE' dropdown menus. Below them are 'Type in actions' and 'Choose state' input fields, and an 'EXECUTE PROGRAM' button.
- FINAL STATE:** and **FINAL COST:** labels at the bottom.
- QUERY**: A section on the right with a 'SELECT QUERY TYPE:' dropdown menu (set to 'Value query'), a text input field, and a 'from initial state' dropdown menu. There is an 'EXECUTE QUERY' button at the bottom.

Figure (30) Program definition for the Titanic scenario

in Figure 31. It can be easily compared with the visualization prepared by hand in the theoretical section. Both obtained figures present the same results, hence, the application is working correctly. Additionally, to evaluate the program execution, the sequence of actions $\{Jack\ moves, Rose\ moves\}$ has been specified. Once again, the same sequence of actions was used in the theoretical example so the results are comparable. The obtained state is presented in Figure 32.

4.7 Test 5 - Shipping orders example

The second complicated scenario test used the example of package delivery. To recall, the formulation of the scenario is as follows:

There is an order that needs to be processed and delivered to the customer. The status of an order can be described by the following literals: received, packed, shipped, delivered and empty. Initially, the order is neither received nor packed, nor shipped, not delivered, and the shipping box for an order is empty. Placing an order makes it received. Packing an order makes it packed and also the shipping box becomes non-empty. Shipping an order makes it shipped and finally delivering it makes it delivered. Each of those aforementioned actions has an associated cost attached in case the initial conditions for performing a specific action are met.

The fluents that were considered in the scenario are:

1. **received**
2. **packed**
3. **shipped**

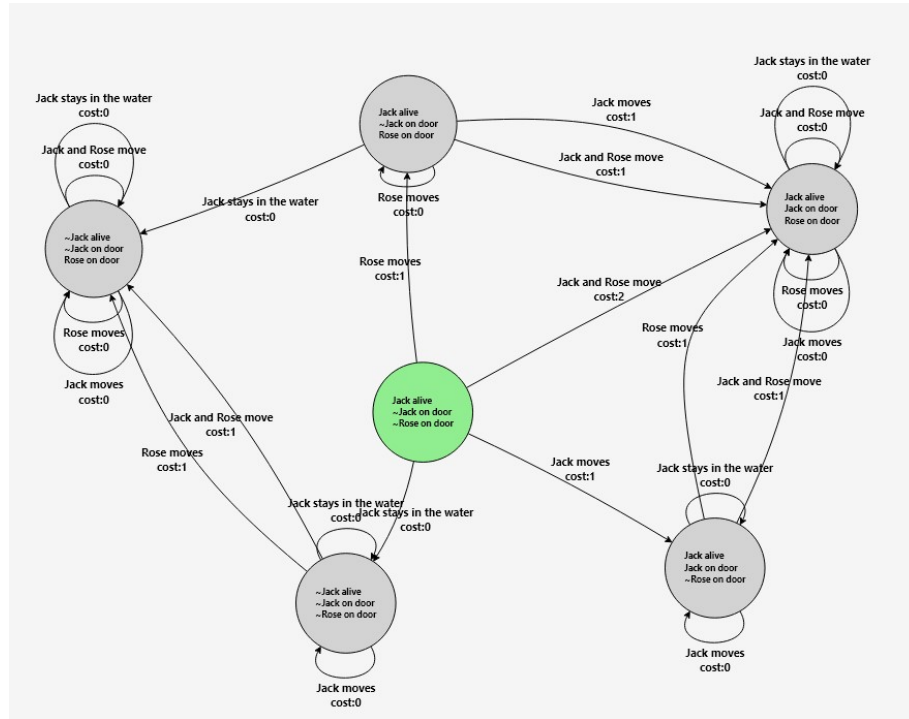


Figure (31) State diagram of Titanic scenario

SPECIFY PROGRAM

ACTION SEQUENCE

INITIAL STATE

EXECUTE PROGRAM

FINAL STATE:

Jack on door,Rose on door,Jack alive

FINAL COST:

2

Figure (32) Result of program execution in Titanic scenario

4. **delivered**

5. **empty**

The list of the actions considered in this scenario was:

1. **Place**

2. **Pack**

3. **Ship**

4. **Deliver**

The representation of the scenario prepared in the application is presented in Figure 33, while the graph representation is visible in Figure 34. As can be observed, the obtained graph representation corresponds to the one presented in the example. In order to evaluate the results of the execution of the program, the following sequence of actions has been specified: $\{Place, Pack, Ship, Delivery\}$. The resulting state and final cost, which are presented in Figure 35 correspond to the ones obtained in the theoretical section. Therefore, as all tests yielded positive results, it can be concluded that the application is working semantically and syntactically correct.

The screenshot shows a web application titled "Actions with costs". It features several panels for configuring a scenario:

- FLUENTS:** A list of fluents including "received", "packed", "shipped", and "delivered".
- ACTIONS:** A list of actions including "Place", "Pack", "Ship", and "Deliver".
- ADD STATEMENT:** A section for defining logical statements with fields for "Effect statement", "causes", "if", and "cost".
- ACTION DOMAIN:** A section for defining the domain of actions, including checkboxes for "initially ~received", "initially ~packed", "initially ~shipped", "initially ~delivered", "initially empty", and conditional rules like "Place causes received if ~received cost 1".
- SPECIFY PROGRAM:** A section for specifying the program, including fields for "ACTION SEQUENCE", "INITIAL STATE", and "EXECUTE PROGRAM".
- QUERY:** A section for defining a query, including a "SELECT QUERY TYPE:" dropdown and fields for "Value query", "after", and "from initial state".

Buttons for "ADD", "DELETE", "DELETE ALL", "OPEN VISUALIZATION", and "EXECUTE QUERY" are visible throughout the interface.

Figure (33) Representation of the package delivery scenario

5 Individual contribution

The following table summarizes the individual contribution of each of the team members. Bellow the table, each team member specified the percentage of her/his work in each of the two parts of the project: theoretical part and practical. All members contributed to the tests.

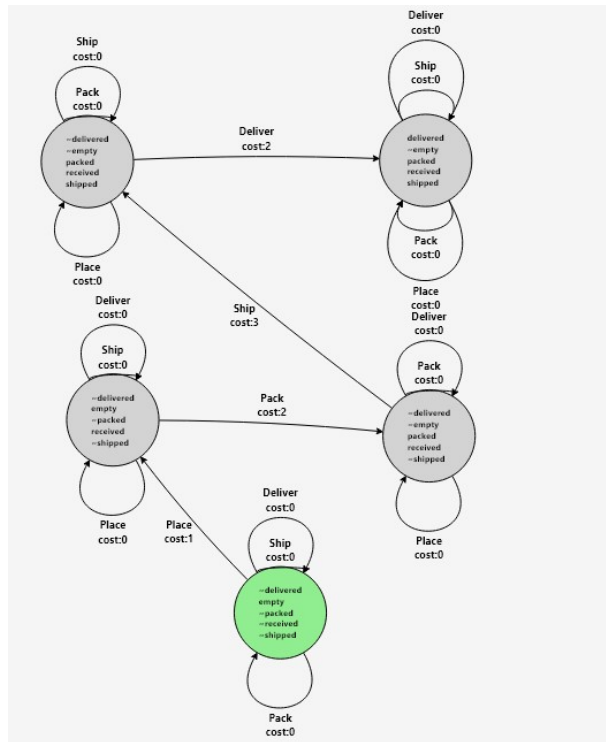


Figure (34) State diagram of package delivery scenario

SPECIFY PROGRAM

ACTION SEQUENCE

Place, Pack, Ship, Deliver

INITIAL STATE

sd, ~delivered, empty

EXECUTE PROGRAM

FINAL STATE:

received, packed, shipped, delivered, ~empty

FINAL COST:

8

Figure (35) Result of program execution for scenario with package delivery

Action description language (syntax, semantics)	Zofia Wrona Marcin Świerkot
Query language (query statements, satisfiability of queries)	Zofia Wrona Marcin Świerkot
Examples of dynamic systems (Titanic)	Zofia Wrona Sonia Grzywacz Marcin Świerkot
Examples of dynamic systems (Shipping orders)	Zofia Wrona Sonia Grzywacz Marcin Świerkot
Examples of dynamic systems (Preparing for holidays)	Zofia Wrona Marcin Świerkot
General layout of the application	Zofia Wrona Sonia Grzywacz Piotr Krzemiński Marcin Świerkot Patryk Grochowicz
Technical Description (Introduction, Technology, Classes, Sections)	Patryk Grochowicz Marcin Świerkot
Fluents and actions section (add fluents/actions, display fluents/actions)	Zofia Wrona Piotr Krzemiński Patryk Grochowicz Marcin Świerkot
Create action domain section (add statements, check consistency)	Zofia Wrona Piotr Krzemiński Marcin Świerkot
Create program section (add actions and states)	Zofia Wrona Piotr Krzemiński Marcin Świerkot
Create program section (evaluation of cost and final state)	Zofia Wrona Piotr Krzemiński Marcin Świerkot
Query section (value/sufficiency query implementation)	Piotr Krzemiński
Visualisation section (states and cost evaluation)	Zofia Wrona Piotr Krzemiński
Visualisation section (graph visualisation)	Zofia Wrona Piotr Krzemiński
Syntax and semantic testing	Zofia Wrona Sonia Grzywacz Piotr Krzemiński Patryk Grochowicz Marcin Świerkot
Scenario testing (Titanic example)	Zofia Wrona Sonia Grzywacz Piotr Krzemiński Patryk Grochowicz Marcin Świerkot
Scenario testing (Shipping orders example)	Zofia Wrona Sonia Grzywacz Piotr Krzemiński Patryk Grochowicz Marcin Świerkot

Table (1) Responsibilities

1. **Zofia Wrona** – Theoretical part: 70% Practical part: 30%
2. **Marcin Świerkot** – Theoretical part: 60% Practical part: 40%
3. **Sonia Grzywacz** – Theoretical part: 60% Practical part: 40%
4. **Piotr Krzemiński** – Theoretical part: 40% Practical part: 60%
5. **Patryk Grochowicz** – Theoretical part: 50% Practical part: 50%