Verifying deadlock and nondeterminism of UML/SysML state machines integrated with activities

Diego Ferreira[1] and Lucas Lima[1]

Universidade Federal Rural de Pernambuco, Rua Dom Manuel de Medeiros, s/n, Dois Irmãos - CEP: 52171-900 - Recife/PE

Abstract. This paper presents a framework for verifying deadlock and nondeterminism in UML/SysML state machines integrated with activities, addressing the critical need for automated checks in UML projects. The framework aims to provide architects and system designers with an automated way to model and verify properties in state machine diagrams integrated with activity diagrams, emphasizing the absence of deadlock and nondeterminism crucial aspects in critical systems. The framework is implemented as a plug-in for the Astah modeling environment, utilizing the Astah API to read the components used in state machine and activity diagrams. The translation of the diagram is performed into the formal CSP language and is verified using the FDR tool. In the case of deadlock or nondeterminism, an interactive counterexample is generated in the modeling platform, facilitating the identification of the reasons for the failure. The paper also discusses the developed semantics, a case study, and the functionalities of the framework. Additionally, it compares this work with related approaches and highlights the limitations and future directions for the framework.

Keywords: state machine · activity · deadlock · nondeterminism · CSP

1 Introduction

The increasing complexity of systems and software projects has prompted the intensive use of models for representing and refining ideas. In this context, the Unified Modeling Language (UML) [14] stands out as a widely adopted modeling language. Within the realm of UML, state machine and activity diagrams are frequently employed for describing system behavior [18].

As modern systems become increasingly complex, there is a growing demand for a more robust and secure approach to representing and analyzing system behaviors. Many studies emphasize the importance of conducting verifications to identify flaws during the system modeling phase, as unnoticed defects can lead to detrimental consequences for the project, such as high costs associated with bug correction and significant time loss due to necessary rework [6]. However, when using informal models, such as those expressed in UML, there is a significant

risk of ambiguities in understanding the system's behavior, leading to imprecise decisions prone to errors due to subjective interpretation.

In industry, project development typically evolves from an abstract model to a concrete model, reflecting the progression of understanding system concepts. This evolution occurs through the production of a series of models, each refining its predecessor. However, the choice between formal or informal semantics for these models can significantly impact project quality. While informal models are more accessible, their interpretation is subjective and depends on the designer's experience, presenting significant risks [8]. On the other hand, formal models offer precise semantics and tool support to increase reliability, but understanding these models is more challenging due to the need for manipulation of mathematical concepts.

The UML language, recognized as an essential standard for software and system modeling, encompasses both structural and behavioral aspects. When considering behavioral diagrams, the state machine diagram stands out as a widely used notation in the industry to describe event-oriented objects in reactive systems [18]. This diagram, composed of vertices connected by transitions, is activated by events, outlining the possible state flows in a system. The visual representation of this model incorporates simple and composite states, pseudostates, and various notations, enriching the understanding of the dynamic behavior of the system. Simultaneously, the activity diagram, also part of UML, provides a comprehensive view of the system's dynamics, illustrating how different actions are interconnected and how the system responds to specific events. Both diagrams play crucial roles in behavioral modeling, providing a holistic understanding of the system at different levels of abstraction.

It is essential to note that state machine and activity diagrams already have initiatives for hidden verification with formal methods [11, 12]. Hidden verification involves checking an abstract model of the system through a semantic translation into a logical formula. Subsequently, the formula is verified using formal methods.

For example, a tool for activity diagram verification undergoes the process of transforming an activity diagram into a formal notation, performing the necessary checks, and returning a counterexample if needed [11].

Another relevant framework is the RoboTool [1], which addresses the formal verification of state machine diagrams in the context of the RoboChart language. The research highlights the importance of applying formal methods to ensure the correctness of robotic state machine-based systems [12].

However, it is quite common for system engineers to model system behavior using both state machines and activities. More precisely, state machines whose actions can invoke activities. As an example, we can observe these dynamics in the open SysML model of the Thirty-Meter Telescope [17], which has been developed in the context of the OpenMBEE community [16] as an industrial scale application. However, there is still a lack of specific automated verification

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

tools for these integrated diagrams. Our work aims to fill this gap, contributing to the advancement of critical systems modeling and verification.

The integration of these two models allows for a more comprehensive and accurate representation of the system's behavior. This integration enables a state machine diagram to invoke activities during actions performed by states and transitions. This cohesion between the two diagrams offers a more complete and detailed view of the system, enabling the modeling of complex behaviors involving the execution of activities in response to specific events, as well as handling the received feedback.

However, it is essential to highlight that, to date, there is a shortage of tools for checking integrated state machines and activity diagrams. This research gap is addressed in our work, where we propose an innovative approach for the automatic verification of these diagrams, combining the advantages of UML modeling with the robustness provided by formal semantics, specifically using the CSP process algebra [7]. It is noteworthy to mention that while our discussions primarily focus on UML, the techniques and methodologies introduced herein are equally applicable to the equivalent diagrams in SysML.

Our research proposes an innovative approach, integrating state machine and activity diagrams, considering both the ease of modeling offered by UML and the robustness provided by formal semantics. The application of this approach will allow for the automatic verification of crucial properties, such as deadlock and nondeterminism, contributing to the construction of more reliable and robust systems.

We concretize our approach as an implementation of a plug-in for the Astah modeling environment. We hope that this tool will provide architects and system designers with an automated way to model and verify properties in state machine diagrams, with a focus on the absence of deadlock and nondeterminism, essential aspects in critical systems.

The remaining sections of this paper are organized as follows. In Section 2, we present the fundamental concepts used in our work, encompassing UML/SysML concepts for state machines and activity diagrams, along with the formal language CSP. Section 3 introduces the semantics of activities and state machines in terms of CSP. Section 4 outlines the tool support we provide to facilitate verification. Following that, Section 5 illustrates verification strategies through a case study. Section 6 discusses related works, placing our contributions within the context of existing research. The conclusion and discussion of initiatives for future work are presented in Section 7.

2 Background

In this section, we present the baseline concepts that are used in our approach. Section 2.1 details the behavioral elements of UML/SysML that we support, in this case, state machines and activities. Section 2.2 describes the semantic domain we use to represent the behavior semantics of state machines and activities, which is the process algebra CSP.

2.1 UML/SysML models

Activity Diagram is a type of UML diagram consisting of activity nodes connected by edges. Activity nodes come in three types: action, control, or object nodes. Action nodes perform a behavior determined to be executed upon passing through that node. There are various types of action nodes, such as basic action, accept event, send signal, and call behavior.

Control nodes, on the other hand, organize the control flows of the activity diagram. Control nodes play a crucial role in sequencing flows, acting as traffic controllers on the edges of the diagram. The types of control nodes include initial, flow final, activity final, merge, decision, fork, and join.

Finally, object nodes hold data arriving at their incoming edges and offer them to the outgoing edges. Object nodes have some variations, like: basic objects, pins, and parameters.

It is important to emphasize that the execution semantics of an activity diagram involve the flow of tokens through directed edges and nodes. Tokens move from the origin activity node to the destination activity node, but this flow is contingent upon the readiness of the destination to accept the token. Certain nodes generate tokens, such as an initial node at the start of an activity, while others, like flow final and activity final nodes, only consume tokens. Object nodes have the capacity to hold multiple tokens before passing them on to subsequent nodes. For an action node to execute, all incoming edges must offer tokens, and upon completion, the node must provide tokens on its outgoing edges.

The detailed semantics of each constructor and its specific execution can be found in the UML specification [14].

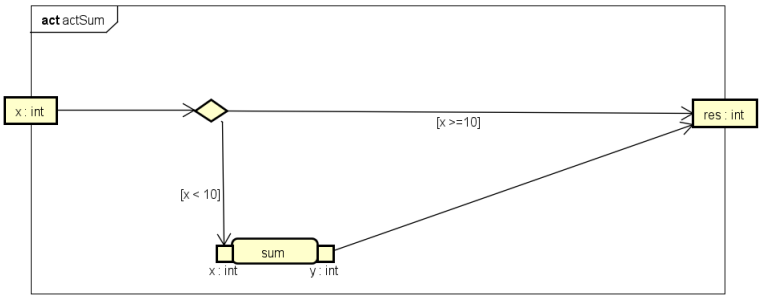

Fig. 1. A simple activity diagram that increments a received input when less than 10.

In Figure 1, a simple activity diagram named `actSum` is depicted featuring an action named `sum` with an input pin and an output pin, a decision node, two activity parameters, and four edges, two of which include guards. It is not explicit here, but assume that the `sum` action increments the input received in `x` and outputs the result in `y`.

This diagram functions as follows: a value is passed as an attribute to the input parameter of the diagram. If it is greater than or equal to 10, it will be returned through the output parameter of the diagram. If the value is less than 10, the designated action in `sum` occurs, defined in this example as the addition of one unit, and is then returned by the output parameter.

State Machine Diagram is comprised of various vertices connected by transitions, activated by a series of events, determining possible activity flows. Vertices can be categorized into states and pseudostates, with states further divided into simple and composite states.

Simple states are those without substates, and they can have the following actions: entry, do, exit. Entry performs a behavior upon entering a state; Do may or may not execute a behavior during the state passage; Exit performs a behavior upon exiting the state. These behaviors can be treated as actions, allowing, for instance, the invocation of activity diagrams. Thus, there is an integration between state machine diagrams and activity diagrams, which can occur both within entry, do, and exit actions, as well as within actions performed by transitions. Composite states, in contrast to simple states, have substates.

Pseudostates, in turn, serve specific functions depending on their type. Among them are the following types: initial, final, choice, junction, fork, join, entry, exit, deep history, and shallow history.

Finally, concerning transitions, they can include triggers, actions, and guards in their notation. Basically, a trigger enables a transition to be executed from a source state to the target state. After the trigger is fired, if the guard evaluates to true, then the action is performed before activating the target state. The detailed semantics of each constructor can be found in the UML specification [14].

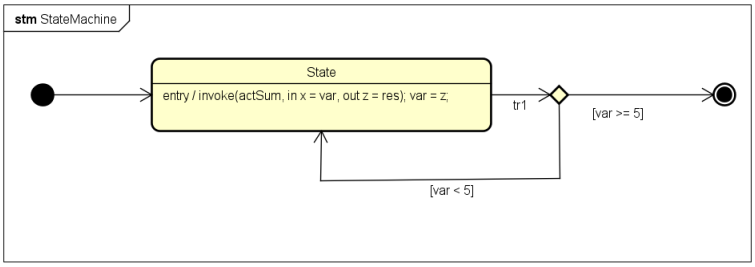

Fig. 2. A state machine diagram with an entry action that invokes an activity diagram

In Figure 2, a state machine diagram is presented featuring a simple state, initial, choice and final pseudostates, and four transitions, two of which include guards.

After the initialization of this diagram, in the first state, it is possible to notice an entry action performing a function call. After concluding the state behavior,

the exit transition can be triggered, leading to the choice pseudostate. If the variable `var` currently has a value less than 5, it will return to `State`. However, if the value of `var` is greater than or equal to 5, the process will proceed to the final pseudostate, and consequently, the diagram will terminate.

The example in Figure 2 is free of deadlocks. To illustrate a deadlock, one can simply change the logical operator in the guard condition from $[var \geq 5]$ to $[var < 5]$. In this case, when reaching the decision node, there will be no outgoing path for the case where `var` is equal to 5, thereby preventing this behavior from advancing.

2.2 CSP

The CSP language (Communicating Sequential Processes) [7], proposed by Hoare in 1985, provides a formal approach for specifying concurrent systems. With formally unambiguous semantics and automated refinement calculation mechanisms, such as those offered by the FDR tool (Failures-Divergences Refinement) [5]. FDR allows checking the absence of deadlocks, livelocks, and nondeterminism of CSP specifications. It uses a Labeled Transition System (LTS) as the internal representation of the CSP model.

The behavior model in CSP is described through processes, the fundamental units of description. These processes, defined in terms of events or other processes, can be composed in parallel in a synchronized or interleaved manner, providing flexibility in modeling concurrent behaviors. By applying CSP to the semantic description of UML diagrams, we are formalizing the semantics of UML. This approach offers more precise and unambiguous models, enabling the use of formal language tools for automated logical reasoning.

The CSP process algebra is a very versatile way to specify systems that consist of interactive components. Each component functions independently and has its own interface designed for interaction with the environment. This formalism provides us with tools to clearly define and analyze the interactions among the different components. In our case, these components are the nodes of the diagrams.

CSP processes are the fundamental unit of behavior and are defined in terms of events and other processes. The function $\alpha(P)$ represents the set of events that a process P can communicate. The basic process, $SKIP$, represents successful termination. A process of the form $a \rightarrow P$ presents the event a to the environment and then behaves as the process P. CSP channels are used to abstract sets of events that share a common prefix. The syntax $c?x$ denotes a channel c receiving a value x, where x is a value of a type which also types channel c. The value for x is determined by the environment. The syntax $c.e$ ($c!e$) represents an expression e communicated through channel c. The difference between . and ! becomes relevant when the channel communicates more than one value simultaneously.

Sequential composition in CSP notation, denoted by $P1\,;P2$, works in a way that is similar to the process $P1$. When $P1$ is successfully completed, control is then passed on to $P2$. Although CSP lacks a specific operator for recursion,

a process name can be used within its definition. For instance, if we define the process P as $a \rightarrow P$, it will first communicate the event a and then behave as P. In parallel composition, the operator $P1 \underset{cs}{\parallel} P2$ synchronizes the events between $P1$ and $P2$ based on the set of events cs, while any events not present in cs occur independently.

3 Formal semantics of state machine and activity diagrams

In this section, we will discuss the formal semantics of state machine diagrams and activity diagrams. The semantics defined here are based on related works [10–13] that established the semantics for these diagrams. These semantics have been studied and outlined to achieve efficient and accurate translations from UML diagrams to CSP_m code, which is a machine-readable version of CSP that is accepted by the FDR tool.

Furthermore, in this section, we will explain how this work has developed the integration semantics between state machine diagrams and activity diagrams, which is the main contribution of our paper. Although there are tools that verify diagrams independently, the verification of these diagrams, when integrated, still lacks specific tools.

3.1 Overview

The formal semantics for activity diagrams are constructed on the foundations proposed by previous works [10, 11] and can be seen in Figure 3. Key elements, such as action nodes, control nodes, and objects, are addressed in terms of their interactions and sequential or concurrent executions. The formal representation of these elements in the CSP language aims to ensure fidelity in the transition from the activity model to the formal description of the system's behavior.

Figure 3 illustrates the semantics of an activity diagram in CSP graphically. The roundtangles represent CSP processes and the arrows represent CSP events. The activity is encapsulated within the `MainProcess` process. This process is exclusively triggered by the `startActivity` event and is designed to conclude upon the occurrence of the `endActivity` event. It assumes a central role, comprehensively encompassing the entire activity.

Within `MainProcess`, a process referred to as `Nodes` represents the fundamental units of the activity. Each `Node_n` corresponds to a specific CSP process related to an activity node, executed in parallel synchronizing on the edges as CSP events. This parallelization is crucial for mirroring the concurrent nature of activities and actions transpiring simultaneously.

The synchronization and coordination of these `Node` processes occur concurrently with events denoted as `a_n`, which are related to their edges. Each `Node_n` is synchronized in parallel with these `a_n` events, ensuring a coherent and synchronized execution of the nodes according to their interconnection via edges.

In addition to the `Node` processes, there is another process known as `TokenManager` that controls and supervises them. `TokenManager` acts as a coordinator, ensuring that `Nodes` are executed as expected and that the memory associated with tokens is managed appropriately. It is also responsible for controlling the termination of the whole activity.

To ensure a cohesive and controlled flow of activities throughout the entire process, `Nodes` synchronizes with the `TokenManager` process on specific events such as `update`, `clear`, and `endDiagram`.

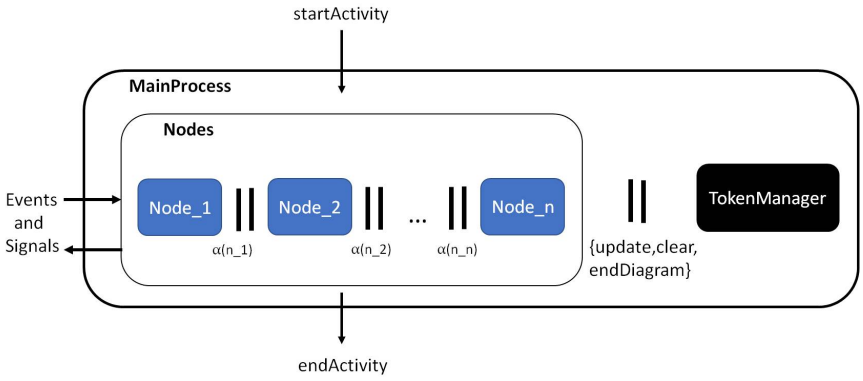

Fig. 3. Activity diagram semantics in CSP [11].

On the other hand, the formal semantics for state machine diagrams considers various elements present in the diagrams, such as states, transitions, pseudostates, and events. The formalization of these elements in terms of the CSP language aims to represent the dynamic behavior of the system unambiguously.

The semantics for state machines in terms of CSP takes into account the reactive and event-driven nature of the model. Events trigger state transitions, and each state may be associated with specific actions. Although our semantics for state machines were inspired by the one provided by the RoboChart language [13, 12], it is an independent implementation with its own design decisions.

In Figure 4, we observe an overview of the semantics for state machines and the interrelation of the CSP-generated processes using the same notion presented in Figure 3. Simple states constitute the foundation of this diagram type, residing within the state machine or even nested within composite states. Notably, these simple states, composite states, transitions, memory, and the controller are all modeled as CSP processes.

States or pseudostates within a state machine are intricately linked to transition processes. These transitions are activated through trigger events occurring in CSP channels, forming a crucial synchronization mechanism. Simple states, composite states, transitions, memory, and the controller, all represented as distinct CSP processes, work concurrently. This concurrency allows for the parallel

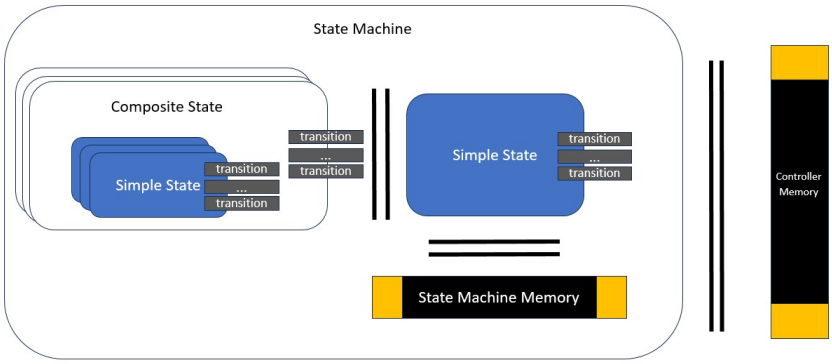

Fig. 4. State machine diagram semantics in CSP.

flow of states, enabling the progression of the system. The synchronization occurs through key events in CSP channels, ensuring coherence in the simulated state transitions.

Moreover, states and transitions operate in parallel concerning the state machine's memory, which functions as a shared resource among the processes. The memory, managed by a controller, facilitates the sending or modification of predefined variables as specified in the diagram.

In essence, the generated semantics employ CSP processes to function either sequentially or in parallel composition. The parallel synchronization of key events is instrumental in achieving an accurate emulation of the modeled behavior of a state machine. As this paper does not focus on this specific semantics, more details can be seen in [13, 12], which are works that inspired our implementation.

3.2 Integrated semantics of State Machines and Activities

Our contribution is to provide a notion for a common usage of state machine and activity diagrams. While individual semantics were established considering the peculiarities of each diagram, integration requires an approach that encompasses communication between both. There are tools that verify diagrams independently, but the lack of support addressing the verification of these diagrams when integrated is a gap that this work seeks to fill.

Figure 5 illustrates how the integration of both semantics is achieved. Our work only tackles the scenario where activities can be called by state machine actions. Then, inside the `State` process, we can have other processes related to its actions (e.g., entry, do, exit actions). For simplification, here we call it `*action`. We do not present a figure for transitions, but it follows the same principle.

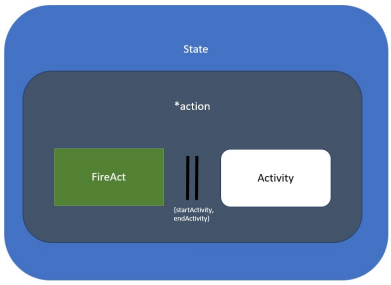

Fig. 5. Semantic integration of state machine and activity

Whenever an action calls an activity, the `*action` process behaves as a parallel composition between the`FireAct` process and the activity diagram process, synchronizing on the events `startActivity` and `endActivity`, which act as channels to initiate and terminate an activity diagram. These events are also the first two events of the `FireAct` process. Essentially, `FireAct` serves as a semaphore that only releases the state machine behavior after the activity diagram process has been completed. Additionally, `startActivity` and `endActivity` transport tokens, where, in the case of `startActivity`, the tokens represent inputs, and `endActivity` is a channel that carries outputs for the activity parameters.

In this way, it is possible to visualize the state machine diagram presented in Figure 2 and how the integration between the two different types of UML diagrams would work. The integration can be observed through the function present in the Entry action of the `State`. This function aims to invoke an activity diagram, passing the necessary inputs for its execution and receiving the outputs upon completion. This function has the following structure:

```
invoke(actName, in i1 = q, ..., out j = o1, ...)
```

It necessarily has the name of the activity being invoked as its first parameter. The subsequent parameters pertain to the structure of inputs and outputs of the selected activity. For each input, the `in` term should be added to indicate that it is an input, followed by the respective activity input parameter name, then `=` and the value to be sent. For each output, the `out` term should be added to indicate that it is an output, followed by the variable that will receive the value, then `=` and the name of the activity output parameter. For instance, `in i1 = q` represents an input for activity parameter `i1` receiving the state machine variable `q`, while `out j = o1` assigns to variable `j` the activity output parameter `o1`.

Thus, the function `invoke(actSum, in x=var, out z=res)` in Figure 2 details the entry action will invoke the activity diagram `actSum`, represented in Figure 1, passing the variable `var` to the input `x`, and receiving `res` in `z` as the output. Afterwards, `z` is assigned to `var` in `var = z`.

Therefore, it can be verified that the diagrams represented in figures 1 and 2, respectively, can be integrated simply through a function that allows efficient translation to the formal CSP model, and consequently allows the verification of properties on models that use both diagrams.

Next, we show the CSP representation for this scenario. It is described in terms of the `EntryProc` process, where the `entry` event initiates the entry action procedure. During the execution of the `EntryProc`, the value of the variable `var` is collected from the memory through the `get_var` event. Subsequently, the activity diagram is invoked in parallel with `FireAct` process, synchronizing on the `startActivity` event to pass the inputs and `endActivity` event to receive the outputs. At the conclusion of the `FireAct` process, the value of the variable `var` is updated by the output through the `set_var` event. After the completion of the `actSum` and `FireAct` processes, the next process is initiated through sequential composition. In this case, the next process corresponds to the do action specified in the `State_st_State_Do_StateMachine` process.

$$EntryProc(st) = entry.st \rightarrow get\_var?var \rightarrow (actSum(id) \underset{startActivity\_actSum,endActivity\_actSum}{\parallel} FireAct(id)) \,;\, State\_st\_State\_Do\_StateMachine$$

$$FireAct(i) = startActivity.i!var \rightarrow endActivity.i?z \rightarrow set\_var!(z) \rightarrow SKIP$$

It's important to note that we illustrate the scenario for the `Entry` process, but this pattern can be used to any type of action within the state machine. The generic structure allows for the seamless integration of different actions, ensuring a cohesive and adaptable design within the state machine.

4 Tool support

In this section, we address the infrastructure and tooling support for applying the semantics defined in Section 3. We describe its architecture and resources used to implement and validate the semantics of state machines and activities. Additionally, we discuss the features provided by the developed tool.

4.1 Architecture

Our framework has been implemented as a plug-in, BPV (Behavior Property Verifier), for the Astah modeling environment. This plug-in can verify activity and state machine diagrams in isolation, and now, their integration as well. Figure 6 illustrates how the plug-in and its dependencies are organized. Our framework is built upon the UML version of Astah, which allows the use of plug-ins to introduce new features and runs on the JVM (Java Virtual Machine). Thus, we used the Astah API to create a new plug-in and to programmatically read, verify, and analyze state machine and activity diagrams.

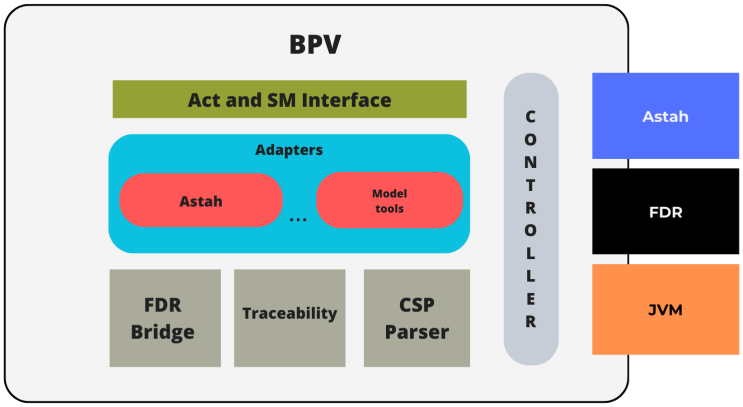

Fig. 6. Plug-in Architecture.

The developed plug-in is divided into five modules. The `Controller` module is responsible for managing, receiving, and returning information between our tool and the modeling environment. The `Adapters` module transforms the tool representation of these diagrams to our internal representation available in the `Act and SM Interface`. The `CSP Parser` module is responsible for translating these representations to CSP_m according to the semantics in Section 3. The `FDR Bridge` module, in turn, is responsible for sending the information translated into CSP to the FDR tool. Finally, the `Traceability` module is responsible for receiving a trace that identifies a counterexample (if any) from the verified diagram. Thus, it can return to the Controller the necessary information to generate a diagram, either a state machine or activity diagram, showing the counterexample to the user in a diagrammatic notation.

4.2 Parser

The automatic translation of state machine diagrams and activity diagrams is facilitated by the plug-in, with translation rules directly embedded in the Java program that has been developed. This translation process is based on the works of Lima et al. [11] for activity diagrams and Miyazawa et al. [12] for state machine diagrams. In addition to adapting these two works for the tool, it was necessary to create a bridge that connects the two and can thus perform an integrated verification of the diagrams.

While this automated process may introduce inconsistencies due to coding errors, we have taken measures to mitigate this by adopting a Test-Driven Development (TDD) approach [2]. In this approach, we initially define test cases

outlining the expected translation behavior before proceeding to implement the parser code.

The creation of elements in the CSP specification based on the input diagram follows the following structure: first, the translation process for state machine diagrams is carried out, and for each invocation of an activity diagram, the translation of the invoked diagram is performed only once.

4.3 FDR integration

Specifying the FDR installation directory is the only requirement for integrating with FDR, which can be done through the plug-in's user interface menu.

```
Tools -> Properties Plugin Configuration -> FDR Location
```

Whenever a deadlock check or nondeterminism is required, the FDR Bridge module dynamically invokes FDR thanks to the provided path, making it unnecessary to include FDR as an embedded plug-in component.

4.4 Deadlock and nondeterminism analysis

Users can check if a state machine diagram or an activity diagram is deadlock-free or deterministic by clicking on the respective option in the user interface. Once the plug-in is installed, the menu, as shown in Figure 7, is available allowing users to select the type of verification for the current diagram. For deadlock-freedom verification, especially in the case of a state machine diagram integrated with an activity diagram, users should select the root diagram, which is the state machine. Therefore, to access the plug-in UI menu, users need to follow these steps.

```
Tools -> Verification -> State Machine Diagram -> Check Deadlock
```

In this example, the tool performs the following tasks: it generates the corresponding CSP specification for all related diagrams, loads it into FDR, and invokes the deadlock-freedom assertion. If FDR detects a deadlock, it provides an interactive counterexample trace that shows the sequence of events that led to the deadlock.

4.5 Traceability

When the FDR tool encounters a deadlock or nondeterminism, it returns a counterexample trace. This trace is an ordered list of events that shows the path leading up to the point where the flaw occurred. The list is important because it helps to identify and create an interactive and diagrammatic counterexample. In this counterexample, you can navigate forward or backward the elements of the diagram that belong to this trace. The current element is highlighted in red, and when the user reaches the last element it is highlighted in purple. This interactive counterexample helps the user to understand how the issue occurred.

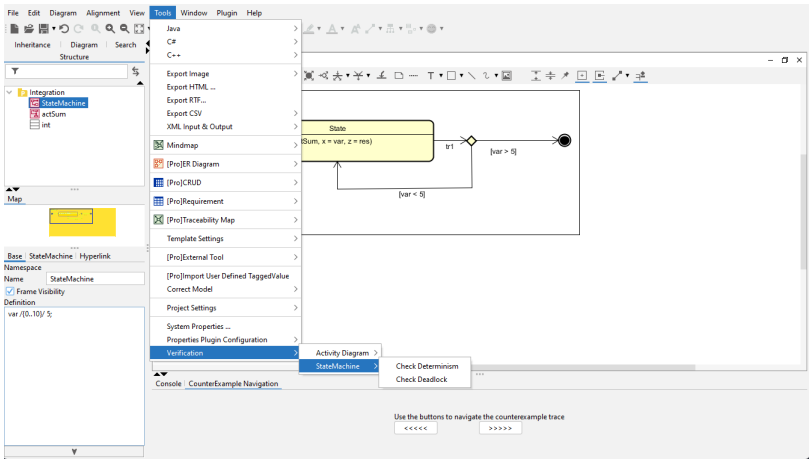

Fig. 7. Steps to use the plug-in in Astah and the interactive navigation buttons

The diagram presented in Figure 9 is the same as the one shown in Figure 2, except for a small difference: the condition `var` $\geq$ `5` has been replaced with `var > 5`. This change causes a deadlock, as explained in Section 2. When we check this property, the dialog shown in Figure 8 alerts the user of this issue. Once detected, the counterexample diagram can be navigated using the forward and backward buttons located in the console, as illustrated at the bottom of Figure 7.

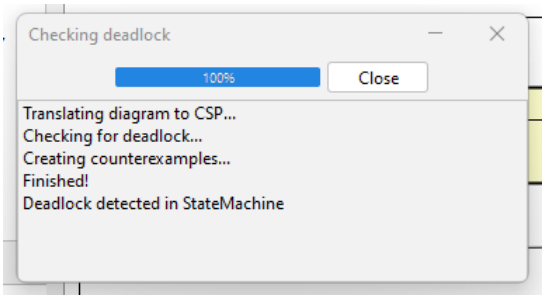

Fig. 8. Dialog with checking deadlock status.

Each step forward in the trace reveals the next node or transition. In this example, the first event in the trace is the state `State`, which is highlighted in red to make it easier to follow, as can be seen in Figure 9. Following this node, the next event in the trace is the transition `tr1`, followed by the choice pseudostate. The deadlock occurs at the choice pseudostate because there is a path for `var > 5` and `var < 5`, but there is no path for when `var` is equal to 5.

Therefore, in Figure 10, the choice pseudostate is painted in purple to indicate the end of the counterexample trace.

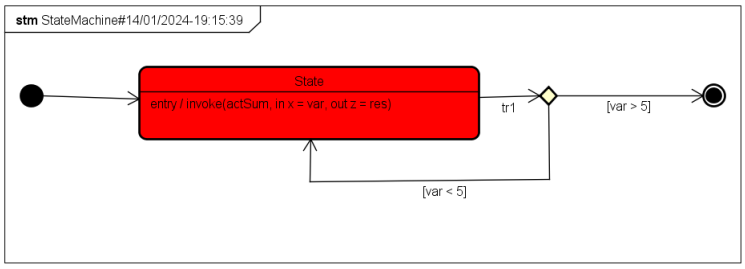

Fig. 9. First step from deadlock counterexample navigation in Astah.

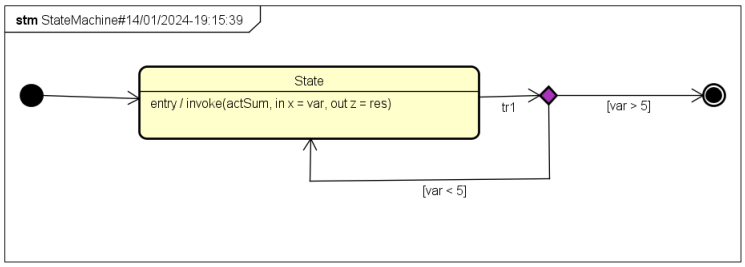

Fig. 10. Last step from deadlock counterexample navigation in Astah.

5 Case study

The diagram model, represented in Figure 11, is designed to represent a common flashlight characterized by a battery that runs out with use. Additionally, the flashlight features a charger for replenishing the battery. The main purpose is to simulate the dynamic behavior of the flashlight in different states, such as on, off, in the recharging process, and SOS mode.

The structure of the model comprises a state machine diagram representing the flashlight and two activity diagrams depicting operations of addition (`actSum`) and subtraction (`actSub`), respectively. The activity diagram `actSum` can be found in Figure 1, while the diagram `actSub` is omitted due to space constraints. However, it follows the same principles as `actSum` but performs the subtraction operation instead. The state machine diagram consists of four simple

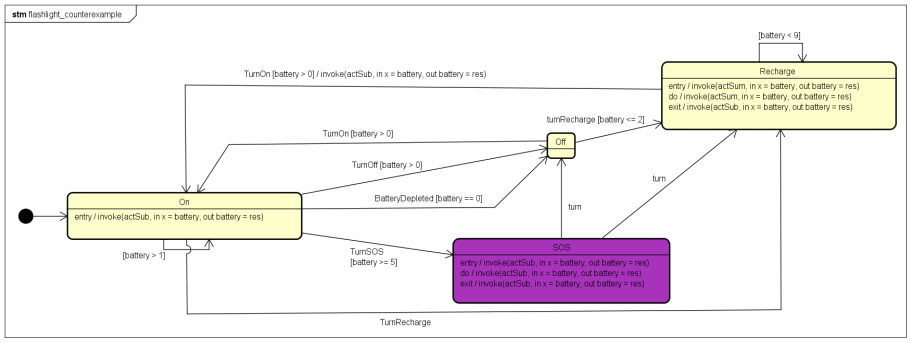

Fig. 11. Flashlight State Machine counterexample for nondeterminism

states (`On`, `Off`, `Recharge`, `SOS`), an initial pseudostate, and 12 transitions, with eight of them having guards.

The states `On`, `SOS`, and `Recharge` contain associated actions. For instance, the `Recharge` state has entry, do, and exit actions. It also has two outgoing transitions, one with a loop in `Recharge` allowing the recharge process to repeat. The entry action executes the `actSum` activity, using the `battery` as input and receiving an incremented value, simulating the recharging process. The do action performs the same operation, interruptible at any moment, and the exit action executes the `actSub` activity, representing a slight battery consumption.

While verifying the model, the tool detects nondeterminism and generates a counterexample diagram. The navigation using interactive buttons led to the `SOS` state, indicated by the highlighted (purple) color. This nondeterminism happens because of the two outgoing transitions with the same trigger `turn`, resulting in distinct situations depending on the path taken. The proposed solution involves changing the trigger's name, replacing `turn` with `TurnOff` and `TurnRecharge`, differentiating the transitions, and eliminating the nondeterminism.

The analysis of the model in FDR took 0.03 seconds to compile the CSP file, which generates an LTS with 2880 states and 3905 transitions, and 0.7 seconds to perform the verification process to check nondeterminism. This scenario shows that our framework can possibly tackle simple to medium complexity models. Nevertheless, further investigation needs to be performed to evaluate how our approach scales and identify possible bottlenecks.

6 Related Work

In this section, we refer to works related to the verification of activity diagrams and state machine diagrams, aiming to analyze and compare properties such as refinement, verification, and integration of diagrams, as well as examining the existence of counterexamples in poorly modeled diagrams, among other aspects.

The work by Miyazawa et al. [12] focuses on the modeling and verification of the functional behavior of robotic applications. They propose an approach

to model robotic applications and provide support for the verification of such modeling through model checking. For this purpose, RoboChart employs a translation process from state machines to CSP, utilizing the FDR tool to verify if the model is deadlock-free. In case a deadlock is detected, a counterexample in the form of a CSP trace is provided to the user. This work does not cover activity diagrams and, consequently, their integration with state machine diagrams. However, our semantic approach draws inspiration from Miyazawa et al.'s work, incorporating certain aspects into our methodology. Unlike Miyazawa's work, our methodology provides traceability of counterexamples to the diagram, enhancing user understanding by linking errors back to the visual representation. This is a distinct feature as Miyazawa's work lacks traceability of counterexamples to the diagram, requiring users to possess a deep understanding of CSP to comprehend errors.

Lima et al. [11] concentrate on the UML activity diagram and offer a framework for automatically verifying deadlocks and nondeterminism in it. They have developed a CSP semantics that can replicate an activity diagram and use the FDR tool to automatically check for deadlocks and nondeterministic behavior. In addition, they have created plugins that allow users to verify directly within the modeling environment without switching between different tools. By utilizing traceability, it is also possible to generate counterexample diagrams. This work does not cover state machine diagrams and, consequently, their integration with activity diagrams. It is noteworthy that our work drew inspiration from Lima et al. [11] for both the tool and the semantics of the activity diagram. However, our tool goes beyond by providing interactive counterexamples. In Lima et al.'s tool, the counterexample is presented statically, painted alongside the trace, whereas in our work, the counterexample is fully interactive, allowing users to navigate through the diagram and gain a deeper understanding of the problem.

Kohlmeyer et al. [9] have presented an innovative approach for characterizing software models using UML 2 state machines, activities, and interactions. In their article, they explore various methods of connecting different diagram types and establishing links between state machines and activities. This methodology utilizes Abstract State Machines (ASMs) [3] and selects the most appropriate formalism corresponding to each level of abstraction. The authors have also defined a formal semantics for facilitating communication between these diagrams. The verification tool, called ActiveCharts, uses the generated formal semantics to directly execute UML activities and simulate models specified by state machines. Its purpose is to identify potential errors in the modeled system, providing a more in-depth understanding of it. However, it is important to note that we were unable to access the tool and its manual despite our attempts.

Abdelhalim et al.'s [1] approach involves translating a subset of fUML [15] to CSP to leverage the FDR tool as a template checker. Their work primarily focuses on deadlock checks. Additionally, users are required to manually manipulate the formal result and the tool to conduct the verification process.

Elmansouri et al. [4] introduced a method for verifying UML 2.0 activity diagrams by automatically transforming them into CSP models. However, users

must specify the properties for verification, utilizing the GROOVE [19] model checker. This approach can detect deadlock and livelock scenarios but lacks traceability for errors and counterexamples.

Table 1. Related work.

Work	**Trans.**	**Verif. AD**	**Verif. SMD**	**Integ.**	**Trac.**	**Form.**	**Purpose**
[1]	✓	✓*	X	X	X	CSP	Deadlock
[4]	✓	✓*	X	X	X	CSP	Property Verification
[9]	✓	✓*	✓*	✓	X	ASM	Property Verification
[11]	✓	✓	X	X	✓	CSP	Deadlock and Nondeterminism
[12]	✓	X	✓	X	X	CSP	Deadlock
Our Work	✓	✓	✓	✓	✓	CSP	Deadlock and Nondeterminism

We summarize the comparison between our work and others discussed so far in Table 1. The characteristics considered in this comparison include automated translation, automated verification of activity diagrams, automated verification of state machine diagrams, integration between these two types of verification, traceability, formalism, and purpose. We use the symbol ✓ to indicate that the feature is addressed, while X indicates that it is not addressed.

Regarding automated verification, ✓* means that, although the verification is automated, the user must interact with the formal tool, while ✓ indicates that the verification is fully automated, meaning the user does not need to interact with the formal tool.

7 Conclusion

In the conclusion of this work, we present a comprehensive summary of the contributions and results achieved. Our approach focused on defining formal semantics for the integration of state machine and activity diagrams, exploring the efficient translation of these models into the CSP language. Throughout the development, we emphasized the importance of the automatic verification of these diagrams, highlighting features such as deadlock and nondeterminism.

It is crucial to acknowledge the limitations of this work. First, the framework only works in the Astah [20] platform and has not been fully analyzed for its scalability. Second, only a limited number of case studies were conducted. Finally, the framework does not generate visual counterexamples for activity diagrams yet.

For future work, we suggest implementing new plugins to expand the framework's applicability to other modeling environments. Further research should also focus on scalability and conducting more extensive case studies, including

examples from the TMT [17] from OpenMBEE[16]. This work only focuses on state machines being able to call activities in their actions, which is a common way to combine these diagrams. However, another aspect that requires further investigation is activities firing/consuming events that can be handled/triggered by state machines. Finally, addressing the current limitation of visual counterexamples for activity diagrams is necessary to improve the verification process's usability and completeness.

References

1. Abdelhalim, I., et al.: Formal verification of Tokeneer behaviours modelled in fUML using CSP—. In: Proceedings of the 12th international conference on Formal engineering methods and software engineering. pp. 371–387. ICFEM'10, Springer-Verlag, Berlin, Heidelberg (2010), http://dl.acm.org/citation.cfm?id=1939864.1939895
2. Beck: Test Driven Development: By Example. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
3. Börger, E., Stärk, R.: Abstract State Machines. A Method for High-Level System Design and Analysis (01 2003). https://doi.org/10.1007/978-3-642-18216-7
4. Elmansouri, R., Meghzili, S., Chaoui, A.: A uml 2.0 activity diagrams/csp integrated approach for modeling and verification of software systems. Computer Science **22** (04 2021). https://doi.org/10.7494/csci.2021.22.2.3478
5. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.: Fdr3 — a modern refinement checker for csp. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 8413, pp. 187–201. Springer Berlin Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_13
6. Haskins, B., Stecklein, J., Dick, B., Moroney, G., Lovell, R., Dabney, J.: 8.4.2 error cost escalation through the project life cycle. INCOSE International Symposium **14**, 1723–1737 (06 2004). https://doi.org/10.1002/j.2334-5837.2004.tb00608.x
7. Hoare, C.A.R.: Communicating and Sequential Processes. Prentice Hall (1985)
8. Khendek, F., Bourduas, S., Vincent, D.: Stepwise design with message sequence charts. In: Kim, M., Chin, B., Kang, S., Lee, D. (eds.) Formal Techniques for Networked and Distributed Systems. pp. 19–34. Springer US, Boston, MA (2001)
9. Kohlmeyer, J., Guttmann, W.: Unifying the semantics of uml 2 state, activity and interaction diagrams. In: Pnueli, A., Virbitskaite, I., Voronkov, A. (eds.) Perspectives of Systems Informatics. pp. 206–217. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
10. Lima, L., Didier, A., Cornélio, M.: A formal semantics for sysml activity diagrams. In: Iyoda, J., de Moura, L. (eds.) Formal Methods: Foundations and Applications. pp. 179–194. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
11. Lima, L., Tavares, A., Nogueira, S.C.: A framework for verifying deadlock and nondeterminism in uml activity diagrams based on csp. Science of Computer Programming **197**, 102497 (2020). https://doi.org/https://doi.org/10.1016/j.scico.2020.102497, https://www.sciencedirect.com/science/article/pii/S0167642320301064
12. Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A., Timmis, J., Woodcock, J.: Robochart: modelling and verification of the functional behaviour of

robotic applications. Software & Systems Modeling **18**(5), 3097–3149 (Oct 2019). https://doi.org/10.1007/s10270-018-00710-z, https://doi.org/10.1007/s10270-018-00710-z
13. Miyazawa, A., Ribeiro, P., Ye, K., Cavalcanti, A., Li, W., Woodcock, J., Timmis, J.: Robochart reference manual. Tech. rep., University of York (2017), https://robostar.cs.york.ac.uk/publications/techreports/reports/robochart-reference.pdf
14. Object Management Group: OMG Unified Modeling Language (OMG UML), superstructure, version 2.4.1. Tech. rep., OMG (2011)
15. Object Management Group: Semantics of a Foundational Subset for Executable UML Models (FUML). Tech. rep., Object Management Group (2013), OMG Document Number: formal/2013-08-06
16. OpenMBEE: Open Model Based Engineering Environment). https://www.openmbee.org/, accessed on: 10-12-2023
17. OpenMBEE: TMT-SysML-Model. https://github.com/Open-MBEE/TMT-SysML-Model, accessed on: 16-01-2023
18. Reggio, G., Leotta, M., Ricca, F., Clerissi, D.: What are the used uml diagrams? a preliminary survey. In: EESSMOD@MoDELS. vol. 1078, pp. 3–12 (10 2013)
19. Rensink, A.: The groove simulator: A tool for state space generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) Applications of Graph Transformations with Industrial Relevance. pp. 479–485. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
20. Vision, C.: Astah (2019), http://astah.net/