Raida Elmansouri
Said Meghzili
Allaoua Chaoui

A UML 2.0 ACTIVITY DIAGRAMS/CSP INTEGRATED APPROACH FOR MODELING AND VERIFICATION OF SOFTWARE SYSTEMS

Abstract *This paper proposes an approach that integrates UML 2.0 Activity Diagrams (UML2-ADs) and the communicating sequential process (CSP) for modeling and verifying software systems. A UML2-AD is used for modeling a software system, while a CSP is used for verification purposes. The proposed approach consists of another way of transforming UML2-AD models to CSP models. It also focuses on checking the correctness of some properties of the transformation itself. These properties are specified using linear temporal Logic (LTL) and verified using the GROOVE model checker. This approach is based on model--driven engineering (MDE). The meta-modeling is realized using the AToMPM tool, while the model transformation and the correctness of its properties are realized using the GROOVE tool. Finally, we illustrated this approach through a case study.*

Keywords model-driven engineering, graph transformation, transformation correctness, model checker, GROOVE, software systems

Citation Computer Science 22(2) 2021: 209–233

1. Introduction

The Unified Modeling Language (UML) [27] is a graphical modeling language that is normalized by OMG. It is used to describe an object-oriented system at various levels of abstraction [27]. A UML 2.0 activity diagram (UML2-AD) [26] is a popular modeling technique for modeling the dynamic behavior of a system; however, it suffers from a lack of mathematical semantics, which implies that UML2-AD cannot be used to verify inconsistencies and bugs.

The problem of software correctness remain significantly more challenging than traditional systems and needs formal specification and verification methods and tools. Software modeling greatly reduces the complexity of system design, while formal methods ensure the accuracy of these systems.

In the present work, our main contribution is a combination of UML2-AD and communicating sequential processes (CSP) [3] to specify and verify the dynamic behavior of systems. This work is in the context of integrating formal methods with informal or semi-formal approaches in the field of software development. This integration makes informal approaches more precise and facilitates the use of formal methods. Model-driven engineering (MDE) [25] has a significant role in the development of software in many domains, such as context-aware systems, ambient systems, and embedded systems. Its role is to decrease the complexity of the software development. However, MDE suffers from lacks of verification tools. To this end, the challenge now is to integrate this approach with formal languages and models that have the ability to ensure the correctness of these approaches.

In this paper, we propose a new way for transforming UML2-AD to CSP expressions using GROOVE [22]. We also verify the preservation of the semantics of a certain structural property of UML2-AD by the transformation using GROOVE and its model checking. The rest of the paper is organized as follows. In Section 2, we discuss related works. In Section 3, we present the concepts and background of our approach. In Section 4, we present the contributions of this paper: we describe our approach that transforms activity diagrams to CSP expressions, and we define some structural properties of the transformation using GROOVE. In Section 5, we illustrate our approach by using an example. The final section concludes the paper and gives some perspectives.

2. Related works

Several researchers have tackled the problem of model transformation correctness over the last few years. In [8], the authors used the negative application conditions (NACs) to show the completeness and correctness of model transformations based on triple graph grammars [24]. In [1], the authors presented a good classification of a formal verification of model transformation properties through a tridimensional approach: the transformation itself, the properties to be preserved, and the formal verification techniques to be used to verify the properties. They defined the transformation correctness as a variety of properties such as semantic correctness, syntax correctness, termination, and confluence. In [15], the authors tackled the model transformation intents and their properties. In [14], the authors highlighted the necessity of methods that make model transformation verified/validated. They discussed the different scenarios of model transformation verification and validation and introduced the principles of a novel test-driven method for verifying/validating model transformations. In [6], the authors presented a survey on the verification of model transformations. In [21], the authors presented another survey of approaches for verifying model transformations. In [4], the authors identified the requirements that should be satisfied by logics used for reasoning about model transformations. They investigated decidable fragments of first-order logic. In [20], the authors presented a full correctness proof of the technique used to determine that formalizations of such transformations in the form of rule systems are guaranteed to preserve functional properties regardless of the models on which they are applied. The correctness is based on a formal proof conducted with the Coq proof assistant. In [10], the authors introduced model transformations in the form of transformation models that connect source and target meta-models. They analyzed transformation models with a UML and OCL tool on the basis of an implementation of relational logic on top of Kodkod. In [12], the authors proposed a declarative language for the specification of visual contracts, enabling the verification of transformations defined with any transformation language. The verification is performed by compiling the contracts into QVT to detect any disconformities of the transformation results with respect to the contracts. In [7], the authors presented a method that translates target OCL constraints to the source meta-model using the transformation definition. So, they ensured that, if a source model satisfies the advanced constraint, the transformed model will satisfy the target constraint. This method has been integrated with the anATLyzer tool. In [18], the authors proposed an approach that transforms BPMN models to Petri net models using the GROOVE tool and its model checker [11]. They proposed how to validate the termination of the transformation and how to verify the preservation of certain semantic properties of the transformation. In [17] and [16], the authors presented an approach based on SCALA (an environment for executing Isabelle/HOL specifications) that allows us to transform UML state machine diagrams (SMD) to colored Petri nets models and proved the correctness of certain structural properties of this transformation. They also illustrated their approach through another case study of transforming BPMN (business process) into Petri nets. Then, they verified that this transformation preserved certain structural properties.

3. Background and concepts

In this section, we recall some basic concepts about the UML activity diagram, CSP, and graph transformation.

3.1. UML activity diagram

A UML activity diagram is one of the UML diagrams that are aimed at capturing the dynamic behavior of a system. An activity diagram is used for modeling the activity flow of a system by taking the sequence and conditions of the flow into account. An activity diagram describes a business process as a series of actions that can be performed by people, software components, or computers [19]. The termination of an action, the availability of data, and the occurrence of some external event can initiate other actions. A UML activity diagram describes different flows such as parallel, branched, concurrent, and single. For more details, see [26].

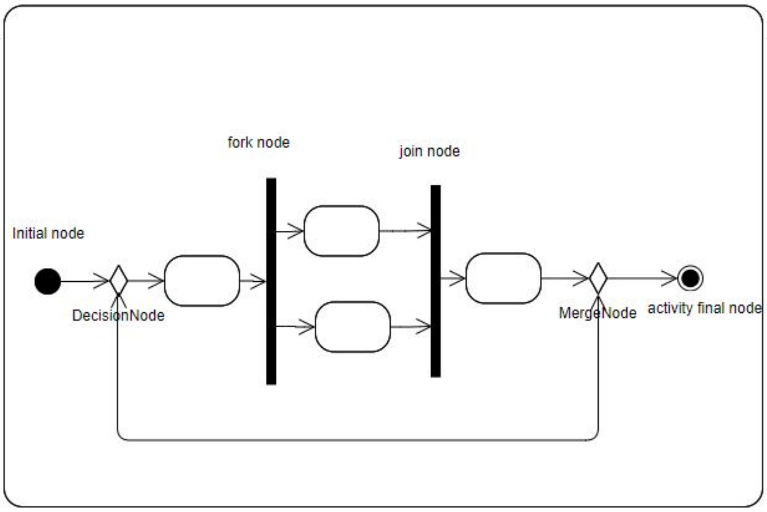


Figure 1. Overview of activity control nodes [26]

In this overview, we can see an initial node, a decision node, a fork node, a join node, a merge node, and an activity final node (see Fig. 1).

3.2. Communicating Sequential Process (CSP)

CSP [13] is a language that was invented by Tony Hoare to specify and reason about concurrent systems. A concurrent system is a set of component processes that interact with each other by communication. These processes are independent entities that interact with the environment through particular interfaces. During an execution, a process performs a sequence of events. A CSP process is completely described and influenced by the ways of its possible communication with its external environment. This is considered to be the basic unit that is used for capturing behavior. For modularity reason, a set of CSP processes are used to describe the behavior of a concurrent system. As with any language, the first step in a CSP process is choosing its alphabet of event communications that are assumed to be instantaneous. STOP is the simplest CSP process, each process is defined by using any equation(s) as follows: P = (behavior). The process names are used to denote system states/modules. The term "communication" comes from the concept of interaction/observation/synchronization and occurs between at least two parts. A sequence of communications gives us a possible behavior as a history (a trace). As defined by Hoare, "a communication is an event that is described by a pair c.v, where c is the name of the channel on which the communication takes place, and v is the value of the message that is passed" [13].

CSP provides some basic operations for the following:

- abstraction;
- choice [ ];
- communication ! ?;
- parallelism |||.

3.3. Graph transformation

The research in the area of graph transformation began in 1970. Since this date, a lot of methods, techniques, and results have been developed and applied in many fields of computer science, such as visual modeling, software engineering, etc. The success of graph transformation in many fields of computer science is due to the fact that the concept of a graph is suitable for describing and explaining complex structures in a direct and intuitive way [23]. Graphs have been proposed for describing the diagrams of the Unified Modeling Language (UML), Entity-Relationship diagrams, Petri nets, etc. A graph transformation is realized by using a graph grammar. Graph grammars have been proposed as a generalization of Chomsky string grammars. The main idea consists of extending the concatenation of strings to a "gluing" of graphs. A graph grammar is a pair: GG = (G0,P), where G0 is called the starting graph, and P is a set of production rules. L(GG), the language generated by the graph grammar GG, is the set of all graphs that can be derived from G0 by applying the rules in P. Several model transformation tools have been proposed in the literature. In the following, we present an overview of the tools that we used in this paper.

3.3.1. GROOVE

GROOVE (GRaphs for Object-Oriented VErification) [9] is a transformation tool graph that uses simple labeled graphs and SPO (single push out) transformation rules. This is based on a formal basis for the transformation and a dynamic semantic. It has the ability to analyze a transformation itself or a graph transformation system using model-based verification. In addition, GROOVE automatically generates the state space of a transformation system, which is stored as a labeled transition system (LTS). In this LTS, a state contains a graph, while a transition is marked with the name of the applied rule. The desired properties of a transformation-like safety and vivacity can be specified in CTL or LTL time logics and verified with the GROOVE model checker. A GROOVE rule as defined in [9] and includes the following:

1. A pattern that must be present in a source graph for a rule to be applicable is represented with the color black.
2. A pattern that must be absent in a graph for a rule to be applicable (optional) is represented with the color red.
3. The elements (nodes and arcs) to be removed from a graph after the application of a rule are represented with the color blue.
4. The elements (nodes and arcs) to add to a graph after the application of a rule are represented with the color green.

For more details, the reader is referred to [9].

3.3.2. AToMPM

AToMPM ("A Tool for Multi-Paradigm Modeling" [2]) is an open-source framework for graph transformation using graph grammars. This allows language developers to define and create new visual domain-specific languages and domain experts to use these languages. It can be used for designing domain-specific modeling environments and managing models. A language is defined by its abstract syntax in a meta-model, its concrete syntax (which defines how each abstract syntax element is visualized), and its semantic definition(s) (either operational [a simulator] or translational [by mapping onto a known semantic domain]). AToMPM supports model transformations to model semantics.

4. Our approach

In this section, we present the transformation of UML 2.0 activity diagrams to CSP processes and its correctness proofs. The objective of this transformation is to formally verify the desired properties of activity diagrams using CSP's process tools and its analytical techniques. The main idea of our approach is depicted in Figure 2; it is achieved automatically into three steps: (1) the transformation of an activity diagram created by AToMPM into its equivalent model in GROOVE using intermediate metaDepth models; (2) the transformation of the obtained activity diagram into CSP processes using GROOVE where a correspondence meta-model is defined from both meta-models (UML AD and CSP); and (3) checking the properties of the transformation itself (expressed as an LTL formula using the GROOVE model checker).

In the following, we present the meta-models, the transformation of the UML2--AD to CSP processes, and the correctness of this transformation itself.

4.1. Meta-models

In order to transform UML2-AD to CSP processes, we use the meta-models of the activity diagram and CSP models that are presented in the task definition [3]. In addition, we define a correspondence meta-model that establishes a relationship between both the source and target models. In the following, we illustrate these steps in detail.

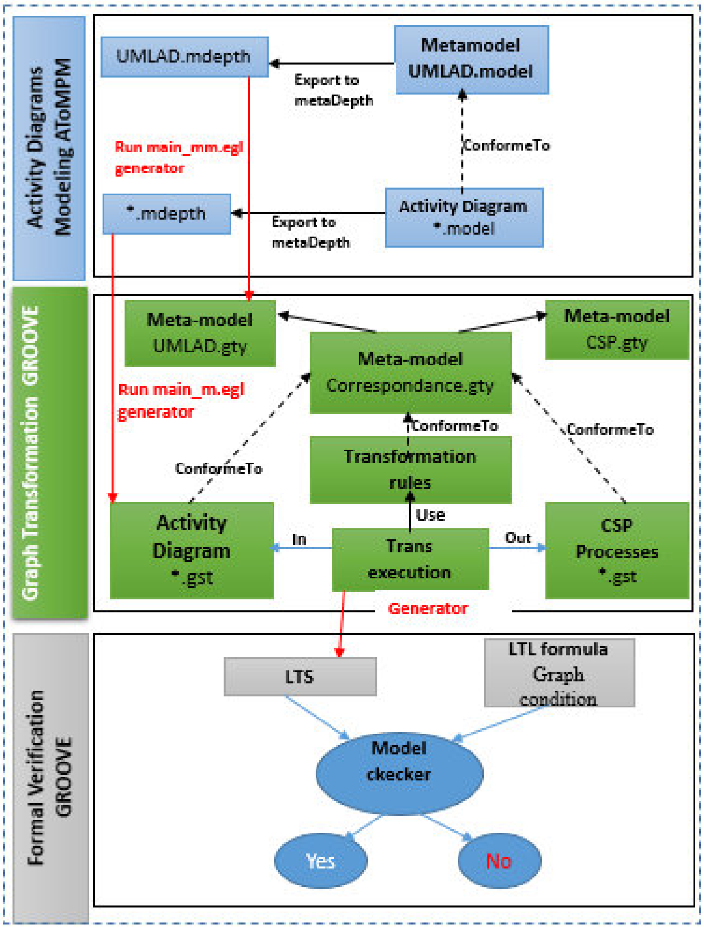


Figure 2. Overview of proposed approach

4.1.1. Meta-modeling activity diagrams

For a UML2-AD meta-model using AToMPM, we propose the meta-model shown in Figure 3. The activity diagrams consist of ActivityNode and one kind of Activityedge connector. An Activitynode has seven different kinds of nodes: Initialnode, Actionnode, Forknode, Joinnode, Mergenode, Decisionnode, and Finalnode. Figure 4 shows the concrete syntax of the UML2-AD meta-model in AToMPM.

Figure 5 shows the meta-model of UML2-AD in GROOVE. This meta-model is generated automatically from the meta-model expressed in AToMPM in Figure 3.

This transformation consists of two steps: the first is performed using MetaDepth, while the second uses the GROOVE plug-in that was developed in [5].

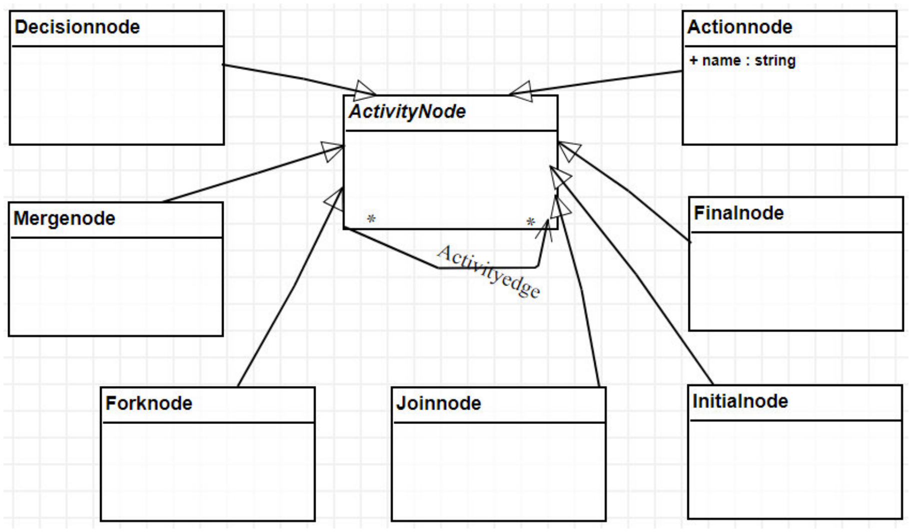


Figure 3. UML2-AD meta-model in AToMPM

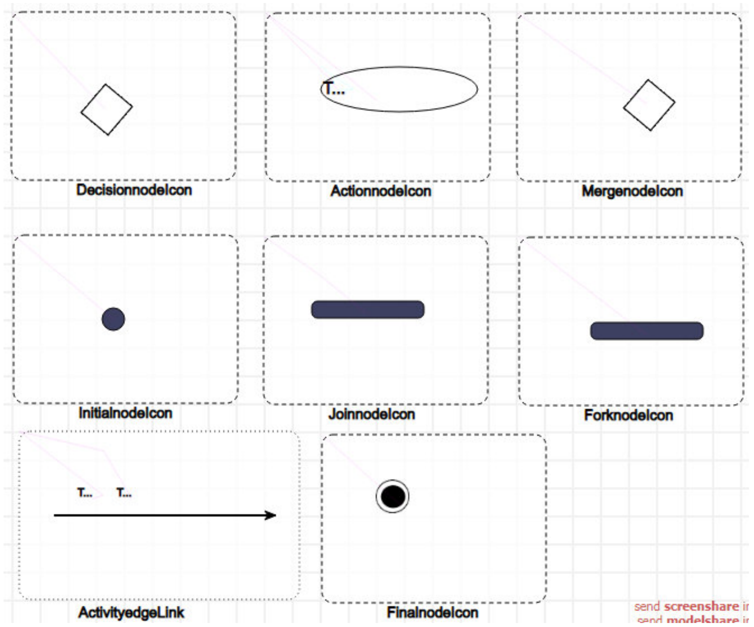


Figure 4. Concrete syntax of UML2-AD meta-model in AToMPM

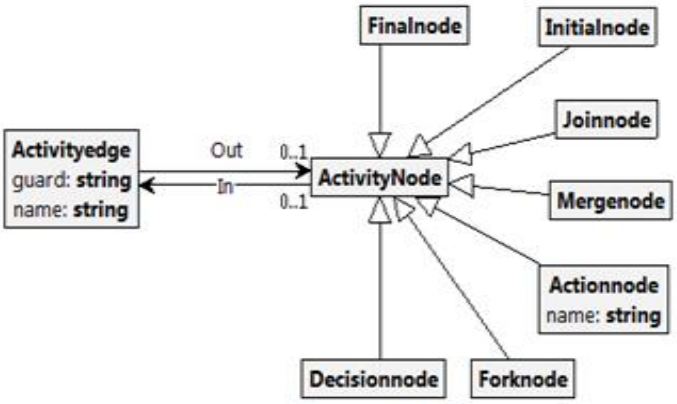


Figure 5. Activity diagram meta-model in GROOVE

4.1.2. Meta-modeling CSP processes

For meta-model CSP processes, we use the meta-model as presented in the task definition [3]. This meta-model is shown in Figure 6. A CspContainer is the root class; it contains a set of ProcessAssignments. A ProcessAssignment contains a process (processIdentifier) on the left-hand side, while it contains a ProcessExpression (process) on the right-hand side. A ProcessExpression can be one of the CSP expressions: a Prefix, Condition, Concurrency, BinaryOperator, or SKIP.

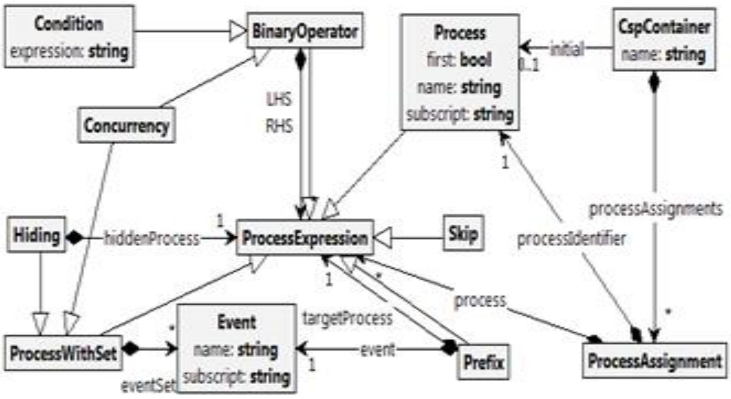


Figure 6. CSP meta-model in GROOVE

4.1.3. Correspondence meta-model

The complete meta-model of this transformation is shown in Figure 7; it defines the relationship between the activity diagrams and CSP processes by additional edges.

These edges are called correspondence edges. For instance, the edge labeled with "to" between the Activityedge and Process nodes.

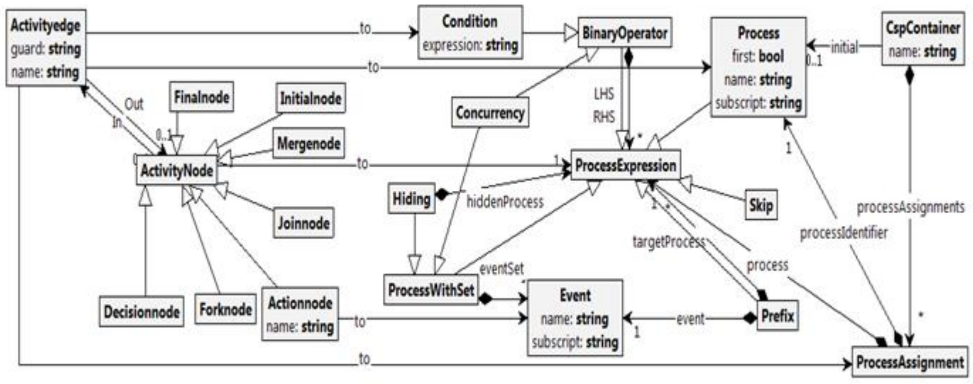


Figure 7. Correspondence meta-model of activity diagram and CSP

4.2. Transforming activity diagram to CSP

To transform activity diagrams into CSP processes, we propose 17 rules. These rules are applied in ascending order according to the priority.

Rule 1: Edge2ProcessAssignment (Priority 17). This rule (Fig. 8) means that each edge of an activity diagram (Activityedge) is translated to a process assignment (ProcessAssignement) of CSP.

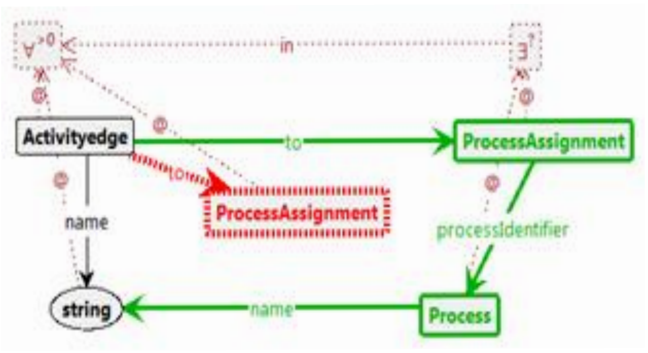


Figure 8. Edge2ProcessAssignment Rule 1

Rule 2: Action2Event (Priority 16). This rule (Fig. 9) means that every action (Actionnode) of an activity diagram is translated to an event (Event) of a prefix expression of CSP.

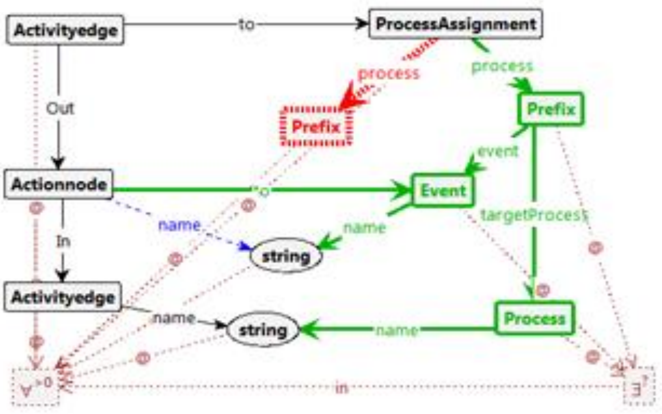


Figure 9. Action2Event Rule 2

Rule 3: MergeNode (Priority 15). This rule (Fig. 10) means that, for each merge node (Mergenode) and for each incoming edge "to," it generates a new process (Process). Also, it deletes all merge nodes.

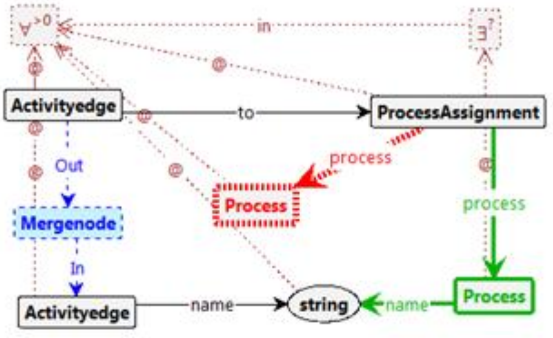


Figure 10. MergeNode Rule 3

Rule 4: FinalNode (Priority 14). This rule (Fig. 11) means that each final node (Finalnode) is translated to a SKIP process for each incoming edge.

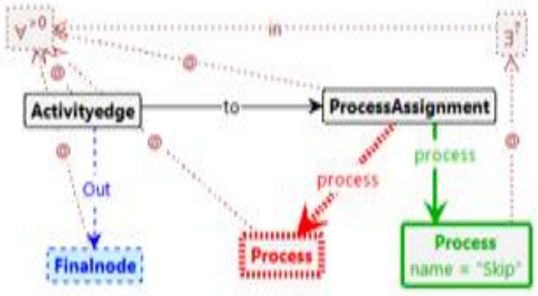


Figure 11. FinalNode Rule 4

Rule 5: InitialNode (Priority 13). This rule (Fig. 12) means that the initial process is marked by variable first = "true." It removes the initial node (Initialnode) of an activity diagram.

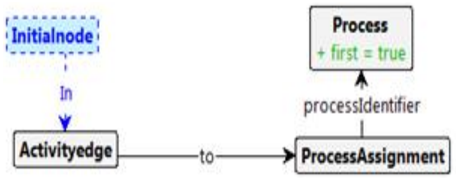


Figure 12. InitialNode Rule 5

Rule 6: JoinNode1 (Priority 12). This rule (Fig. 13) means that the first incoming edge of a join node (Joindenode) is translated to a prefix expression that contains an event (Processjoin) and a target process. This edge is chosen to carry out the continuation process.

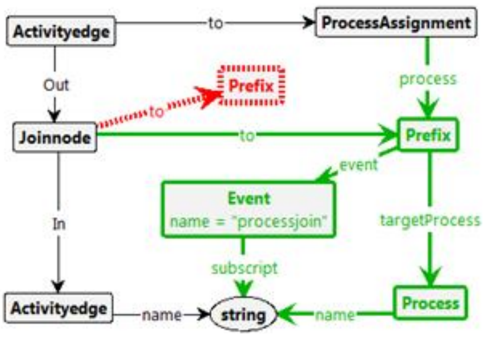


Figure 13. JoinNode1 Rule 6

Rule 7: JoinNode2 (Priority 11). This rule (Fig. 14) means that the other incoming edges of a join node (Joindenode) are translated to a prefix expression that contains an event (Processjoin) and a target process (SKIP).

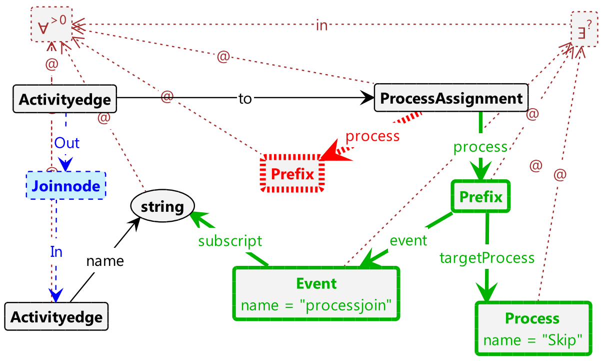


Figure 14. JoinNode2 Rule 7

Rule 8: DecisionNode1 (Priority 10). This rule (Fig. 15) means that the first outgoing edge of a decision node (Decisionnode) is translated to a condition expression of the processAssignment that contains a process (Process) on its left-hand side (LHS).

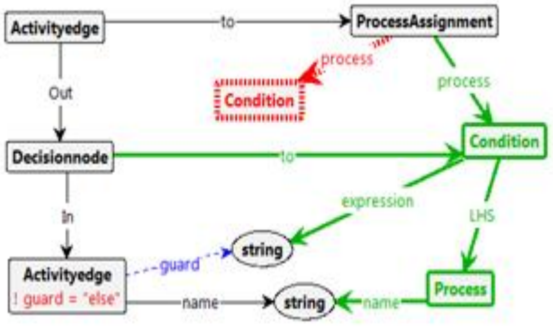


Figure 15. DecisionNode1 Rule 8

Rule 9: DecisionNode2 (Priority 9). This rule (Fig. 16) means that the intermediate outgoing edge of a decision node (Decisionnode) is translated to a condition expression of the condition that contains a process (Process) on its left-hand side (LHS).

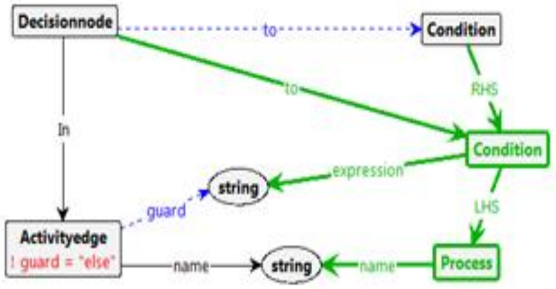


Figure 16. DecisionNode2 Rule 9

Rule 10: DecisionNode3 (Priority 8). This rule (Fig. 17) means that the last outgoing edge (its guard = "else") of a decision node (Decisionnode) is translated to a process (Process) in CSP.

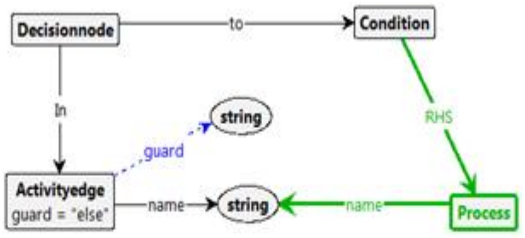


Figure 17. DecisionNode3 Rule 10

Rule 11: ForkNode1 (Priority 7). This rule (Fig. 18) means that the first outgoing edge of a fork node (Forknode) is translated to a concurrency expression of the processAssignments that contains a process (Process) on its left-hand side (LHS).

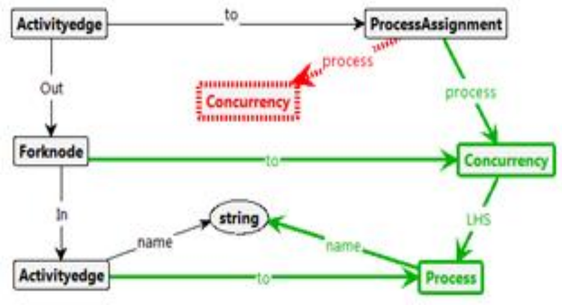


Figure 18. ForkNode1 Rule 11

Rule 12: ForkNode2 (Priority 6). This rule (Fig. 19) means that the intermediate outgoing edge of a fork node (Forknode) is translated to a concurrency expression of the concurrency that contains a process (Process) on its left-hand side (LHS).

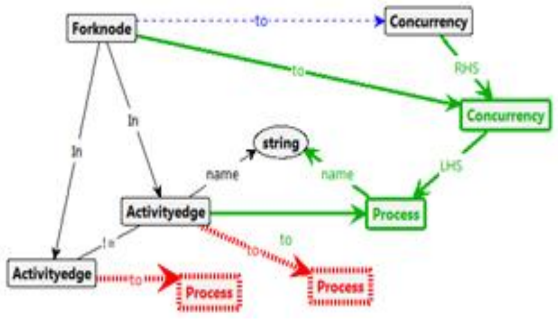


Figure 19. ForkNode2 Rule 12

Rule 13: ForkNode3 (Priority 5). This rule (Fig. 20) means that the last outgoing edge of a fork node (Forknode) is translated to a process (Process) of CSP.

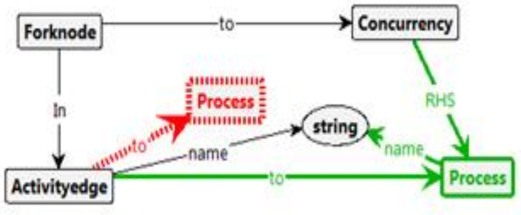


Figure 20. ForkNode3 Rule 13

Rule 14: DelEdges (Priority 4). This rule (Fig. 21) is applied to remove all edges (Activityedges) of a UML activity diagram.

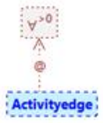


Figure 21. DelEdges Rule 14

Rule 15: DelForks (Priority 3). This rule (Fig. 22) is applied to remove all fork nodes (Forknode) of an activity diagram.

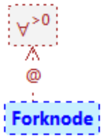


Figure 22. DelForks Rule 15

Rule 16: DelDecisions (Priority 2). This rule (Fig. 23) is applied to remove all decision nodes (Decisionnode) from an activity diagram.

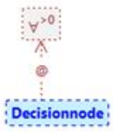


Figure 23. DelDecisions Rule 16

Rule 17: DelActions (Priority 1). This rule (Fig. 24) is applied to remove all actions (Actionnode) from an activity diagram.

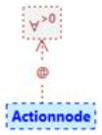


Figure 24. DelActions Rule 17

4.3. Verification of graph transformation

In this part, we tackle the problem of the correctness the transformation itself using GROOVE. This transformation is exogenous since it has different source and target Meta-models. Its intent is translational semantics.

We define two kinds of desired properties for the correctness of the transformation. The first kind is the traceability or the correspondence between the source and target models. The property of this kind is a graph condition that can contain elements of both models. The use of the correspondence meta-model allows us to express this kind of property.

However, the second kind is the preservation of certain semantic properties of the source model in the target model. The property of this kind contains two graph conditions; the first represents a property of the source model, while the second represents a property of the target model. In the following, we illustrate these kinds of properties in detail.

4.3.1. Correspondence properties

To verify the correspondence between an activity diagram and CSP processes, we propose five graph conditions in GROOVE (as follows).

Condition 1: EdgeToProcess. This property (Fig. 25) is valid if for each Activity edge (Activityedge) of an activity diagram model; there is a link to a ProcessAssignment of the CSP model.

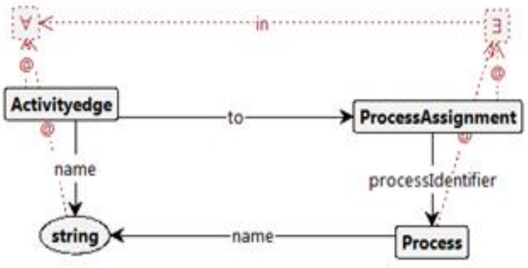


Figure 25. EdgeToProcess Condition 1

Condition 2: ActionToEvent. This property (Fig. 26) is valid if, for each Action of an activity diagram, there is a link to an event of the CSP model.

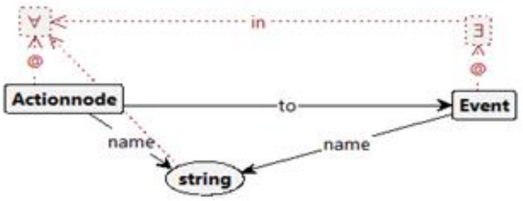


Figure 26. ActionToEvent Condition 2

Condition 3: JoinToPrefix. This property (Fig. 27) is valid if, for each join node of an activity diagram, there is a link to a prefix expression of the CSP model.

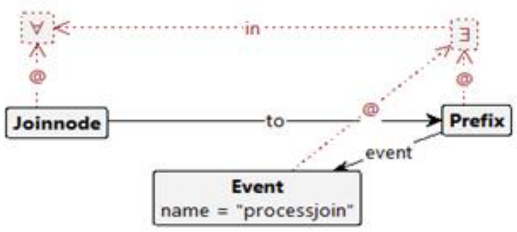


Figure 27. JoinToPrefix Condition 3

Condition 4: DecisionToCondition. This property (Fig. 28) is valid if, for each decision node (Decisionnode) of an activity diagram model, there is a link to a condition expression of the CSP model.

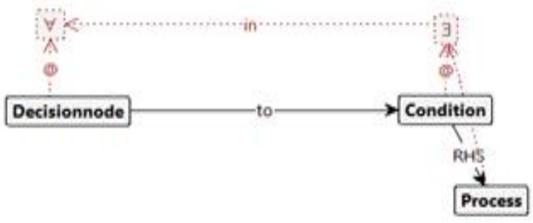


Figure 28. DecisionToCondition Condition 4

Condition 5: ForkToConcurrency. This property (Fig. 29) is valid if, for each fork node (Forknode) of an activity diagram model, there is a link to a concurrency expression of the CSP model.

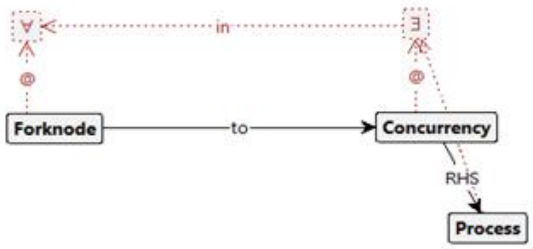


Figure 29. ForkToConcurrency Condition 5

4.3.2. Behavioral properties

In reality, we cannot verify the preservation of the semantics (behavioral properties) of the source model in the target model without defining the semantics of both the source an target models. However, we can test the existence or absence of such a situation or structural property that represents such behavioral properties as deadlock or livelock.

Figure 30 shows a structural deadlock situation of an activity diagram. The deadlock occurs if at least two edges outgoing from the same decision node and incoming to the same join node (synchronization) exist.

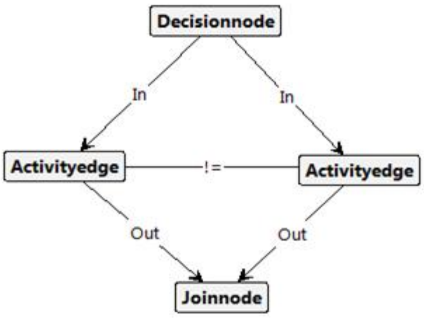


Figure 30. Structural deadlock in activity diagram

Figure 31 shows a structural property in the abstract syntax of the CSP model. This property is semantically equivalent to the property of Figure 30; it contains three process assignments (ProcessAssignment). The incoming process to the decision node is applied to only one of the two outgoing processes of the decision node according to the condition. These two processes must be synchronized by the same event (Processjoin). So, the applied process waits for the other process to engage in the rendezvous. Finally, the deadlock occurs.

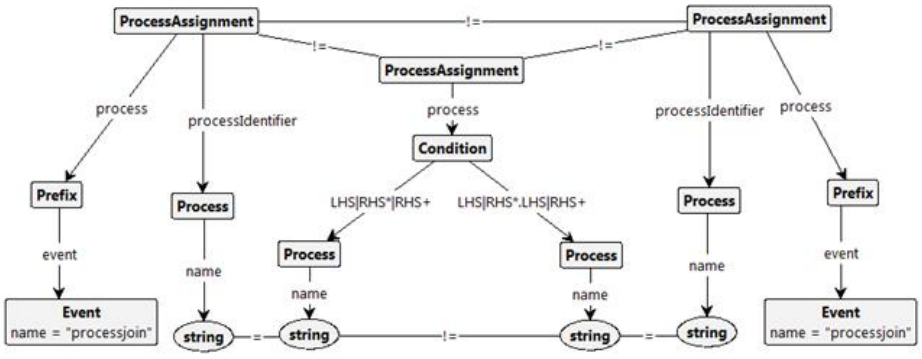


Figure 31. Structural deadlock in CSP processes

5. Case study

To illustrate our approach, we use the example of a UML 2.0 activity diagram that was borrowed from [3]. This example is presented in Figure 32 according to AToMPM syntax.

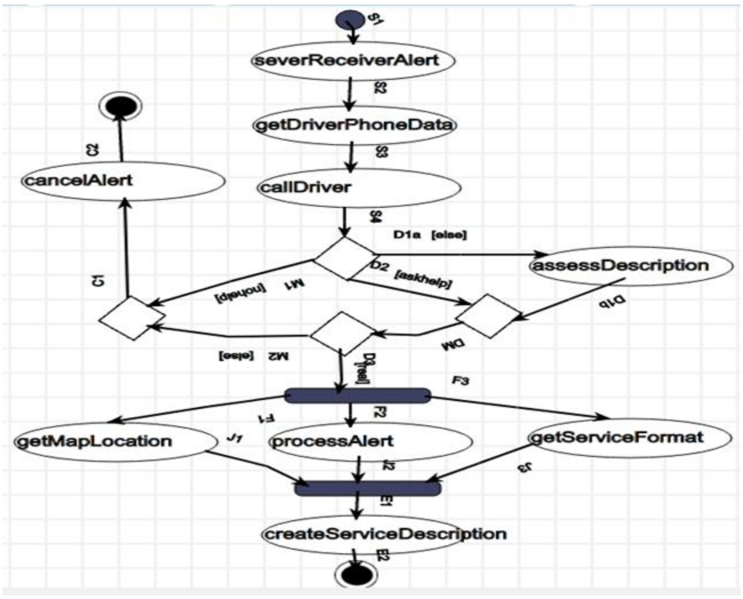


Figure 32. Example of activity diagram created by AToMPM

Its equivalent UML 2.0 activity diagram in GROOVE syntax is shown in Figure 33. The mapping of a model from AToMPM to GROOVE is realized in two sequential steps; the first is performed by using the metaDepth tool, while the second is realized by using the generator from [5]. By the application of our approach on the UML 2.0 activity diagram shown in Figure 33, we have automatically obtained the equivalent CSP processes (in abstract syntax) that are shown in Figure 34.

The labeled transition system (LTS) or graph transformation system of this transformation (from the activity diagram of Figure 33 to the CSP expression of Figure 34) is generated automatically by GROOVE. This LTS is shown in Figure 35. Each state of the LTS represents a stage of transformation and contains its correct properties. Also, each transition represents the applied rule.

In addition, this LTS is the source model of the GROOVE model checker. The properties (graph conditions) that we need for proving their correctness are expressed in LTL and representing the other input of the GROOVE model checker are as follows:

- FG(EdgeToProcess),
- FG(ActionToEvent),
- FG(JoinToPrefix),
- FG(DecisionToCondition),
- FG(ForkToConcurrency).

These properties are expressed in one single property as follows:
FG(EdgeToProcess)& FG(ActionToEvent)& FG(JoinToPrefix)&
FG(DecisionToCondition)& FG(ForkToConcurrency).

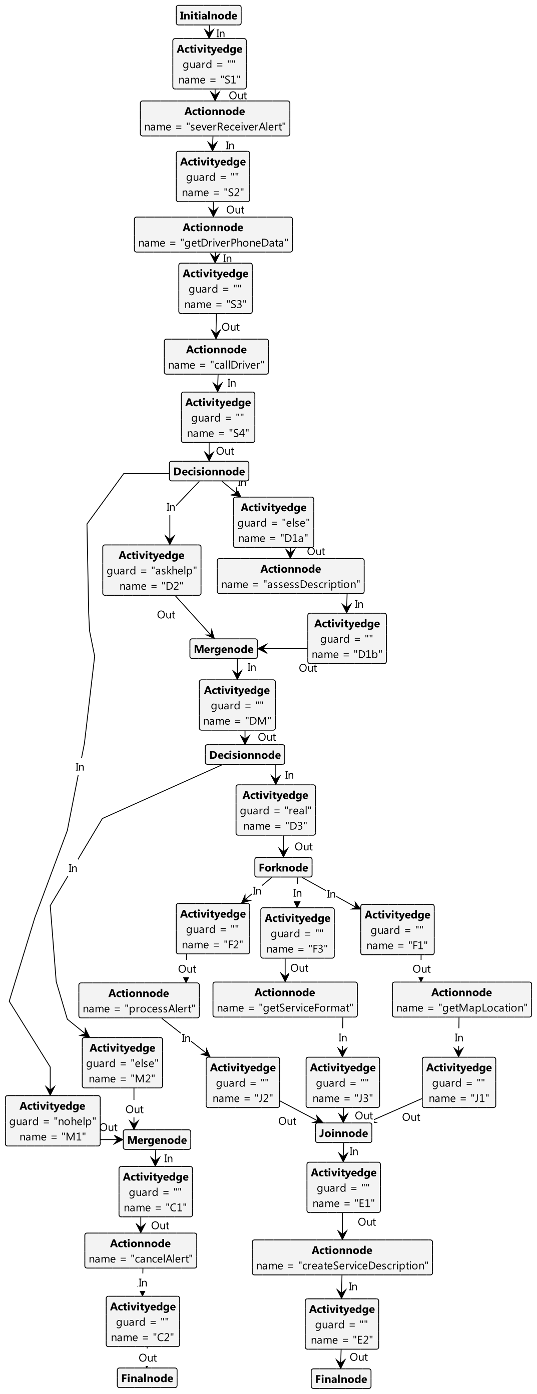

Initialnode
Activityedge guard = "" name = "S1"
Actionnode name = "severReceiverAlert"
Activityedge guard = "" name = "S2"
Actionnode name = "getDriverPhoneData"
Activityedge guard = "" name = "S3"
Actionnode name = "callDriver"
Activityedge guard = "" name = "S4"
Decisionnode
Activityedge guard = "else" name = "D1a"
Activityedge guard = "askhelp" name = "D2"
Actionnode name = "assessDescription"
Activityedge guard = "" name = "D1b"
Mergenode
Activityedge guard = "" name = "DM"
Decisionnode
Activityedge guard = "real" name = "D3"
Forknode
Activityedge guard = "" name = "F2"
Activityedge guard = "" name = "F3"
Activityedge guard = "" name = "F1"
Actionnode name = "processAlert"
Actionnode name = "getServiceFormat"
Actionnode name = "getMapLocation"
Activityedge guard = "else" name = "M2"
Activityedge guard = "" name = "J2"
Activityedge guard = "" name = "J3"
Activityedge guard = "" name = "J1"
Activityedge guard = "nohelp" name = "M1"
Mergenode
Joinnode
Activityedge guard = "" name = "C1"
Activityedge guard = "" name = "E1"
Actionnode name = "cancelAlert"
Actionnode name = "createServiceDescription"
Activityedge guard = "" name = "C2"
Activityedge guard = "" name = "E2"
Finalnode
Finalnode
In
Out

Figure 33. Activity diagram source model in GROOVE

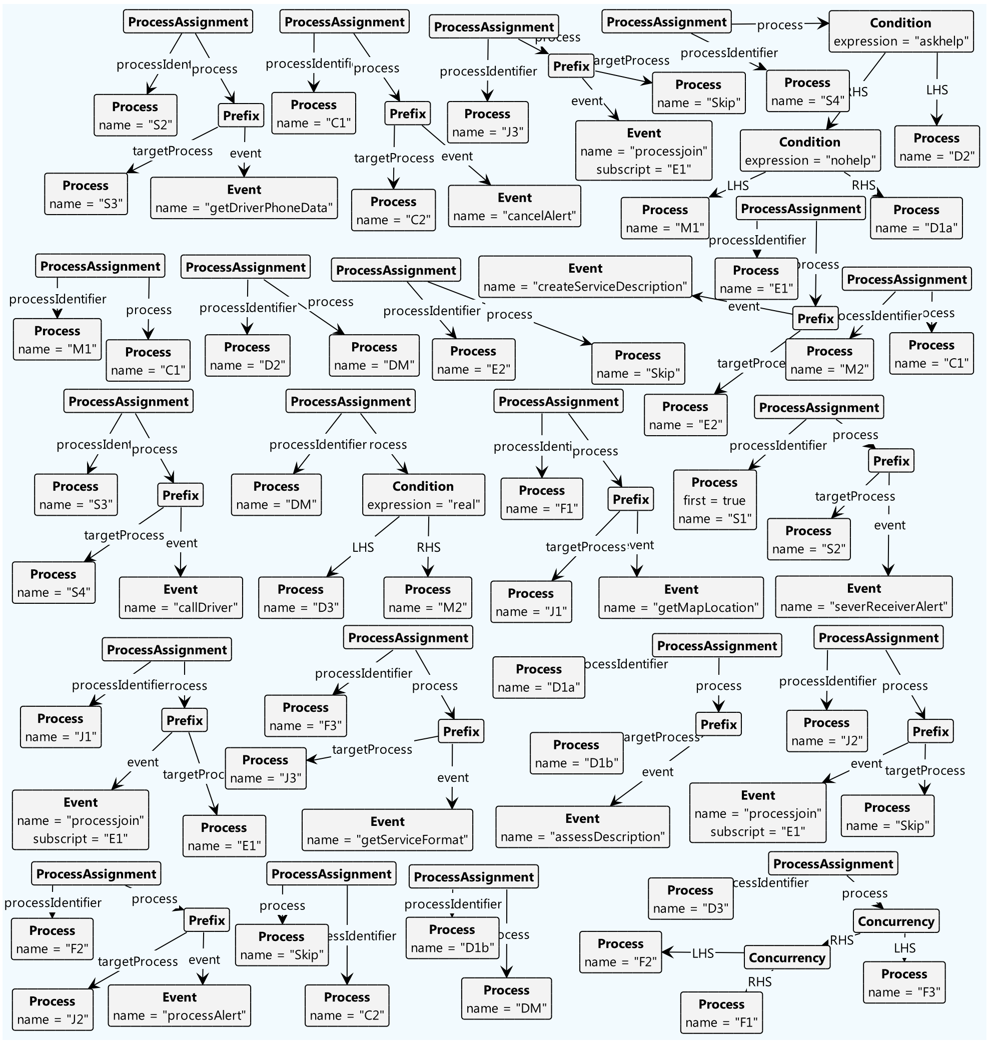


Figure 34. CSP abstract syntax equivalent to activity diagram in GROOVE

We need the LTS that was generated during the transformation shown in Figure 35, and the logical property that is expressed on the LTL logic shown in Figure 36 will also be checked.

Then, the model-checking response that this property holds for this system is the answer (as shown in Figure 37). This means that the path-invariant property is valid.

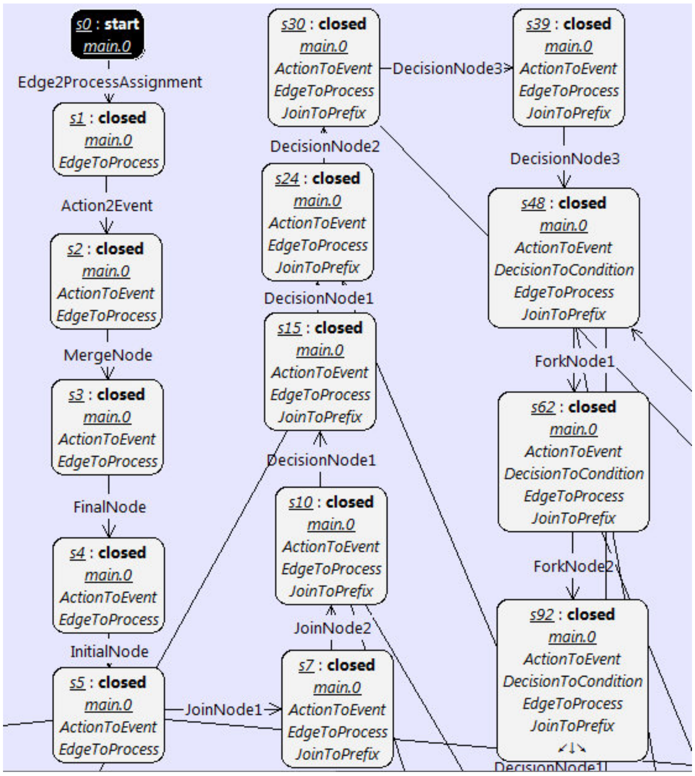
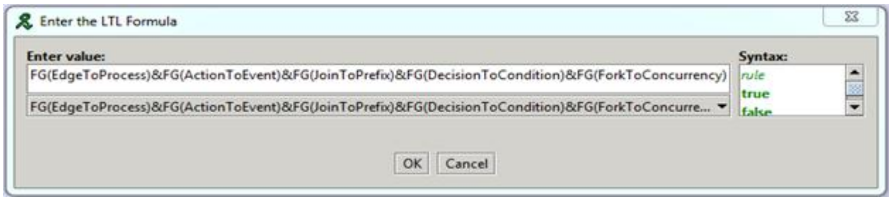


Figure 35. LTS of transforming example

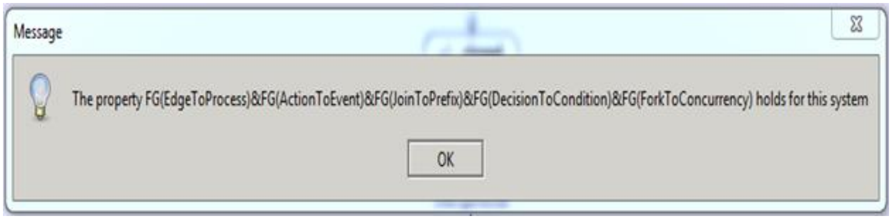


Enter value:

FG(EdgeToProcess)&FG(ActionToEvent)&FG(JoinToPrefix)&FG(DecisionToCondition)&FG(ForkToConcurrency)

FG(EdgeToProcess)&FG(ActionToEvent)&FG(JoinToPrefix)&FG(DecisionToCondition)&FG(ForkToConcurre...

Figure 36. Checked property



The property FG(EdgeToProcess)&FG(ActionToEvent)&FG(JoinToPrefix)&FG(DecisionToCondition)&FG(ForkToConcurrency) holds for this system

Figure 37. Obtained result

6. Conclusions

In this paper, we have proposed an approach integrating UML 2.0 activity diagram (UML2-AD) and communicating sequential process (CSP) for modeling and verifying software systems. We used the UML2-AD for modeling a software system and the CSP for verification purposes. The approach consists of another way of transforming UML2-AD models to communicating sequential process (CSP) models. It also focuses on checking the correctness of some properties of the transformation itself. The main objective of this transformation is the verification of dynamic behavior of systems such as safety and vivacity properties by using CSP techniques and tools. We have also defined some structural properties of the transformation in order to verify their correctness at execution time.

To implement this approach, we have combined AToMPM and GROOVE graph transformation tools. AToMPM was used for the modeling, while GROOVE was used for the transformation and verification of the transformation itself. The desired properties are specified in LTL and verified using the GROOVE model checker.

In future work, we plan to complete the transformation of UML2-AD to CSP expressions by transforming other elements such as object flows. We also plan to verify our transformation approach using a theorem prover and check other essential properties such as behavior correctness.

References

[1] Amrani M., Combemale B., Lúcio L., Selim G., Dingel J., Le Traon Y., Vangheluwe H., Cordy J.R.: Formal Verification Techniques for Model Transformations: A Tridimensional Classification, *The Journal of Object Technology*, vol. 14(3), pp. 1–43, 2015. doi: 10.5381/jot.2015.14.3.a1.

[2] AToMPM project: 2019. http://www-ens.iro.umontreal.ca/~syriani/atompm/atompm.htm.

[3] Bisztray D., Ehrig K., Heckel R.: Case study: UML to CSP Transformation, 2007. AGTIVE 2007 Tool Contest Overview. https://www.informatik.uni-marburg.de/~swt/agtive-contest/UML-to-CSP.pdf.

[4] Brenas J.H., Echahed R., Strecker M.: Ensuring Correctness of Model Transformations While Remaining Decidable. In: *International Colloquium on Theoretical Aspects of Computing*, pp. 315–332, Springer, 2016.

[5] Busser de J.: Model checking AToMPM transformation systems with GROOVE, Technical report, University of Antwerp, 2015.

[6] Calegari D., Szasz N.: Verification of Model Transformations: A Survey of the State-of-the-Art, *Electronic Notes in Theoretical Computer Science*, vol. 292, pp. 5–25, 2013.

[7] Cuadrado J.S., Guerra E., de Lara J., Clarisó R., Cabot J.: Translating Target to Source Constraints in Model-to-Model Transformations. In: *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pp. 12–22, IEEE, 2017. doi: 10.1109/MODELS.2017.12.

[8] Ehrig H., Hermann F., Sartorius C.: Completeness and Correctness of Model Transformations Based on Triple Graph Grammars With Negative Application Conditions, *Electronic Communications of the EASST*, vol. 18, 2009.

[9] Ghamarian A.H., Mol de M., Rensink A., Zambon E., Zimakova M.: Modelling and analysis using GROOVE, *International Journal on Software Tools for Technology Transfer*, vol. 14(1), pp. 15–40, 2012. doi: 10.1007/s10009-011-0186-x.

[10] Gogolla M., Hamann L., Hilken F.: Checking Transformation Model Properties with a UML and OCL Model Validator. In: *Proceedings of 3rd International Workshop on Verification of Model Transformation (VOLT'2014)*, pp. 16–25, 2014.

[11] GROOVE Home Page: 2019. http://GROOVE.cs.utwente.nl/.

[12] Guerra E., Lara de J., Wimmer M., Kappel G., Kusel A., Retschitzegger W., Schönböck J., Schwinger W.: Automated verification of model transformations based on visual contracts, *Automated Software Engineering*, vol. 20, pp. 5–46, 2013. doi: 10.1007/s10515-012-0102-y.

[13] Hoare C.A.R.: Communicating sequential processes, *Communications of the ACM*, vol. 21(8), pp. 666–677, 1978. doi: 10.1145/359576.359585.

[14] Lengyel L., Charaf H.: Test-driven verification/validation of model transformations, *Frontiers of Information Technology & Electronic Engineering*, vol. 16(2), pp. 85–97, 2015.

[15] Lúcio L., Amrani M., Dingel J., Lambers L., Salay R., Selim G.M., Syriani E., Wimmer M.: Model transformation intents and their properties, *Software & Systems Modeling*, vol. 15(3), pp. 647–684, 2016.

[16] Meghzili S., Chaoui A., Strecker M., Kerkouche E.: Transformation and validation of BPMN models to Petri nets models using GROOVE. In: *2016 International Conference on Advanced Aspects of Software Engineering (ICAASE)*, pp. 22–29, IEEE, 2016.

[17] Meghzili S., Chaoui A., Strecker M., Kerkouche E.: On the Verification of UML State Machine Diagrams to Colored Petri Nets Transformation Using Isabelle/HOL. In: *2017 IEEE International Conference on Information Reuse and Integration (IRI)*, pp. 419–426, IEEE, 2017.

[18] Meghzili S., Chaoui A., Strecker M., Kerkouche E.: Verification of Model Transformations Using Isabelle/HOL and Scala, *Information Systems Frontiers*, vol. 21(1), pp. 45–65, 2019.

[19] Penker M.: *Business Modeling With UML: Business Patterns at Work*, John Wiley & Sons, 2000.

[20] Putter de S., Wijs A.: A formal verification technique for behavioural model-to--model transformations, *Formal Aspects of Computing*, vol. 30(1), pp. 3–43, 2018. doi: 10.1007/s00165-017-0437-z.

[21] Rahim L.A., Whittle J.: A survey of approaches for verifying model transformations, *Software & Systems Modeling*, vol. 14(2), pp. 1003–1028, 2015.

[22] Rensink A.: The GROOVE Simulator: A Tool for State Space Generation. In: *International Workshop on Applications of Graph Transformations with Industrial Relevance*, pp. 479–485, Springer, 2003. doi: 10.1007/978-3-540-25959-6_40.

[23] Rozenberg G.: *Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1: Foundations*, World Scientific, 1997. doi: 10.1142/3303.

[24] Schürr A.: Specification of Graph Translators With Triple Graph Grammars. In: *International Workshop on Graph-Theoretic Concepts in Computer Science*, pp. 151–163, Springer, 1994.

[25] Silva da A.R.: Model-driven engineering: A survey supported by the unified conceptual model, *Computer Languages, Systems & Structures*, vol. 43, pp. 139–155, 2015. doi: 10.1016/j.cl.2015.06.001.

[26] UML diagrams: Process Shopping Order. UML Activity Diagram Example: 2015. https://www.uml-diagrams.org/shopping-process-order-uml-activity-diagram-example.html?context=activity-examples.

[27] Unified Modeling Language Specification, version 2.0: 2005. https://www.omg.org/spec/UML/2.0/About-UML/.

Affiliations

Raida Elmansouri
University Constantine 2-Abdelhamid Mehri, MISC Laboratory, Department of Computer Science and Its Applications, Faculty of Ntic, Constantine, Algeria,
raida.elmansouri@univ-constantine2.dz

Said Meghzili
University Constantine 2-Abdelhamid Mehri, MISC Laboratory, Department of Computer Science and Its Applications, Faculty of Ntic, Constantine, Algeria,
said.meghzili@univ-constantine2.dz

Allaoua Chaoui
University Constantine 2-Abdelhamid Mehri, MISC Laboratory, Department of Computer Science and Its Applications, Faculty of Ntic, Constantine, Algeria,
allaoua.chaoui@univ-constantine2.dz

Received: 24.10.2019
Revised: 11.01.2021
Accepted: 15.02.2021