Maejo Int. J. Sci. Technol. **2016**, *10*(03), 272-283

Maejo International Journal of Science and Technology

ISSN 1905-7873
Available online at www.mijst.mju.ac.th

Full Paper

Tool support for transforming Unified Modelling Language sequence diagram to coloured Petri nets

Dulani Meedeniya [1, *], Indika Perera [1] and Juliana Bowles [2]

[1] Department of Computer Science and Engineering, University of Moratuwa, Moratuwa, Sri Lanka
[2] School of Computer Science, University of St Andrews, St Andrews, United Kingdom
* Corresponding author, e-mail: dulanim@cse.mrt.ac.lk

Received: 13 October 2015 / Accepted: 12 October 2016 / Published: 19 October 2016

Abstract: Modern software systems are expected to be dependable and the development of such systems requires strong modelling and analysis methods. Model-Driven Development is becoming a mainstream practice in software development to cater for that need. Models help to cope with the large scale and complexity of software systems by specifying the structural and behavioural aspects of the system and providing a means of communication between domain experts, analysts, designers and developers. Consequently, there is an increasing need for being able to combine the benefits of popular design approaches and formal models to contribute to better software products. Sequence Diagram-to-Coloured Petri Net (SD2CPN) is a scenario-based model transformation tool with analysis capabilities. It captures scenarios using Unified Modelling Language sequence diagrams and transforms them into coloured Petri nets that enable reliable analysis of the system models. The model transformations are based on the strongly consistent model-to-model transformation rules that are formally defined previously as part of this research. This paper presents the design, implementation, main features and usage of SD2CPN tool.

Keywords: SD2CPN, coloured Petri net, model-driven development, model transformation, Unified Modelling Language sequence diagram

INTRODUCTION

Model-Driven Development (MDD) is a popular software engineering practice that addresses the complexity issues of a software system [1-3]. The complexity of a software system is a generic norm within large scale systems. Often the domain specific complexities are associated with the solution architecture, making the management of the design architecture a challenge. A common approach to overcoming such complexities is using multiple models, each of which provides a unique viewpoint of the solution architecture. This is based on the divide-and-conquer software design principle and often brings a new challenge of relating and maintaining the consistency of these multiple models. Moreover, these models may require to be transformed into different models in which each has unique views for further analysis. The manual practice of referring to multiple models in a design and transforming them can be a tedious task for the software engineers. Without automated techniques, generating model transformations manually is deemed to be unrealistic and compromises the design quality factors such as model efficiency, cost and risk management.

In order to ensure that software systems behave as intended, formal verification can be used if the design models are of well-defined semantics. Unified Modelling Language (UML) sequence diagram (SD) is a widely used graphical modelling language for capturing inter-object behaviours, although it lacks formal semantics that is required for the formal verification [5, 6]. On the other hand, coloured Petri net (CPN) [7, 8] is a formal model that can represent a system behaviour, both graphically and mathematically, and enables different analyses of a wide range of systems [7, 9].

Model transformations with tool support mediate the discrepancies between different models by making them consistent with each other while supporting model simulation and/or formal verification [4]. Thus, modelling and transformations are considered as key processes in MDD. As a result, there is a growing demand to explore methods and practices of transforming models in a more efficient, complete and consistent manner. The focus of this research concerns the link between MDD and formal methods of model transformations.

Previous work of this research [10-12] has defined a set of formal rules for model transformation from an SD to a CPN with proof for its correctness. However, developers often find it challenging to use formal representations; instead they prefer automated model transformation as part of their integrated development environments and computer-aided software engineering tools they use. Users of such tools can carry out an architectural redesign and evaluation without having expertise in the underlying formal models being used. This paper presents a tool named Sequence Diagram-to-Coloured Petri Net (SD2CPN) that supports the integrated and automated model transformation from an SD to a CPN.

DESIGN MODELS OF SD AND CPN

SDs are a widely used graphical modelling method for capturing inter-object behaviours with a set of *messages* that communicate between the *instances* participating in an interaction over time (see Figure 1(a)). An SD is represented within a solid-outline rectangular frame around the diagram. The name of the diagram following the keyword **sd** is placed inside a pentagon-shaped compartment on the upper-left corner of the frame. An *instance* can correspond to a particular object in the interaction. An instance has a vertical line called *lifeline* that represents its existence at a particular time. The most visible aspect is a sequence of *messages* that are exchanged between the instances, along with their corresponding occurrence on the lifelines. A message with the same source and target lifeline is called a *self-message* [5, 6]. As shown in Figure 1(a), the SD named **N** contains three object instances, viz. **a:A**, **b:B** and **c:C**. The interactions within the diagram start by instance **a** sending a message $\mathbf{m_0}$ to instance **c**.

A UML SD may contain constructs called *interaction-fragments* denoted by a solid-outlined rectangle and add more structure to part of an interaction. An interaction-fragment has one *operator* (e.g. *alt*), one or more *operands* separated using dashed horizontal lines, and zero or more *guard-conditions* [5, 6]. An *operator* is shown in the upper-left corner of the fragment and determines how its operands are executed. The UML standard [6] defines a range of operators and this example shows only the *alt* operator for alternative behaviour. A *guard-condition* is a Boolean expression that determines whether its operand executes or not and it is shown covering the lifeline where the first event occurs. The SD in Figure 1(a) shows **alt** interaction-fragment behaviour with two operands. Here, instance **c** makes a choice based on the guard-condition, which evaluates to true and sends the message $\mathbf{m_1}$ to instance **b** or a self-message $\mathbf{m_2}$ to instance **c**.

A CPN [7-9] is a directed, connected, bi-partite graph with two node types called *places* and *transitions* that are connected through directed *arcs* (see Figure 1(b)). A CPN is both state- and action-oriented. It describes the states (places) of the system and the operations (net-transitions) that cause the model to change state. Graphically, places are represented by circles, transitions by rectangles, and arcs by arrows connecting places and net-transitions. Places may contain tokens, which are shown as black dots. Tokens and places are associated with colours that distinguish between object types. Each transition is fired when it acquires the relevant tokens from the linked places. When a transition fires, the acquired tokens are passed onto each output place associated with the transition. Thus, the firing of a transition results in a state change for the tokens. Additionally, a transition or an arc may have an associated guard (a Boolean expression) to represent system interactions such as the execution of a conditional statement. The guard is required to evaluate true, to enable the binding and fire the transition.

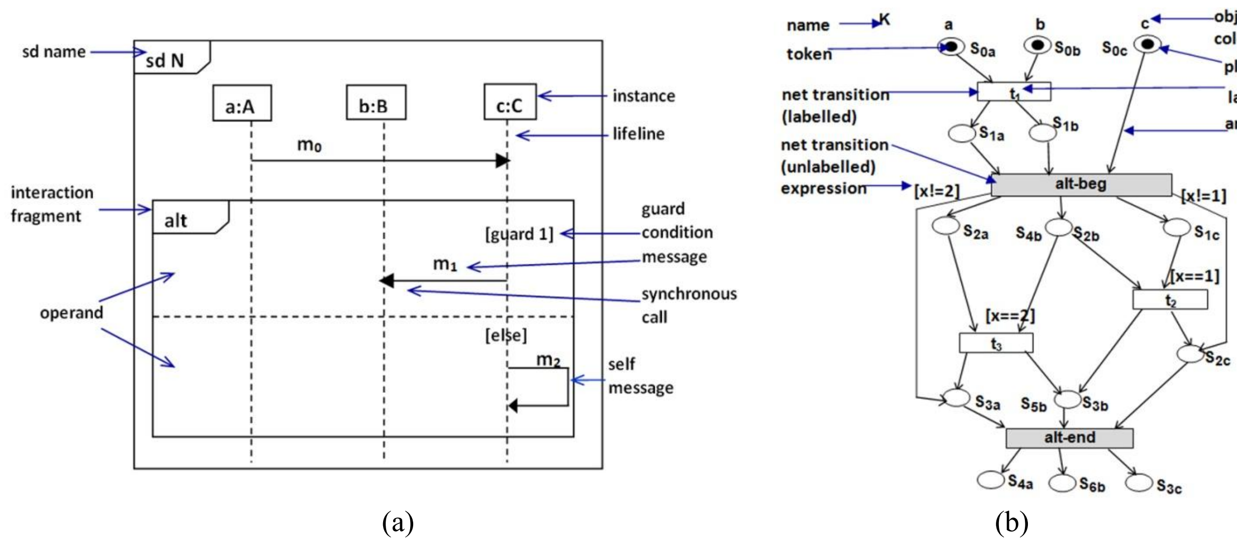


Figure 1. Graphical representation of (a) SD and (b) CPN

Figure 1(b) shows an example of a CPN named **K**, which represents a conditional behaviour. There are three object types (colours): **a**, **b** and **c**, each with one token in their initial places $\mathbf{s0_a}$, $\mathbf{s0_b}$ and $\mathbf{s0_c}$ respectively. CPN **A** contains three labelled net-transitions **t1**, **t2** and **t3**, and unlabelled net-transitions **alt-beg** and **alt-end**, that are used to synchronise the control flow of the model. The transitions **t2** and **t3** are guarded with conditions **[x==1]** and **[x==2]** respectively, and the firing transition is selected based on the condition that evaluates to true.

RELATED WORK

In terms of transforming scenario-based models into state-based models, many approaches were published previously [13-16]. However, some of them considered only the basic SD constructs or event flow of the system and did not consider the handling of the data flow with object-oriented features. Moreover, some presented only a graphical transformation and did not define the formal transformation rules. In some interesting work [15, 16], behavioural trees were used to support both the graphical and formal representations. The evolutionary design and integration relationships were also supported by the models in which the behavioural trees were used.

In our previous work [10-12, 17-19] a formal model-to-model transformation from an SD to a CPN was defined by explicitly stating the connection between the elements of the two models. Also, we have shown the correctness of the transformation by proving that the languages in both models are strongly consistent; thus the transformation is free of implied scenarios. We have contributed to the scalability of the models by supporting partial and incremental analyses of the target model [11]. Further, we have shown the applicability of our transformations for various application domains such as cloud services [12], an elevator system [17] and immersive environments [18].

TOOL DESIGN

The main aim of the SD2CPN tool is the transformation of an SD into a behaviourally equivalent CPN which can be analysed using existing formal methods. This is done based on the previously defined formal model transformation rules [10-12]. The tool supports both the graphical and textual representations of the input and output models. The graphical user interface (GUI) of the tool consists of tool palettes and menus, and provides drag and drop capabilities to draw an SD. The textual representation has the same expressiveness as the graphical notations and the text-based grammar is based on Backus-Naur form [20]. This commonly used graphical representation is easy to understand. However, the textual representation can be used to integrate input and output models with the existing SD and CPN modelling tools. For example, an SD modelled by an existing tool can be an input to perform the transformation and the textual output can be used as an input to another tool to analyse the CPN.

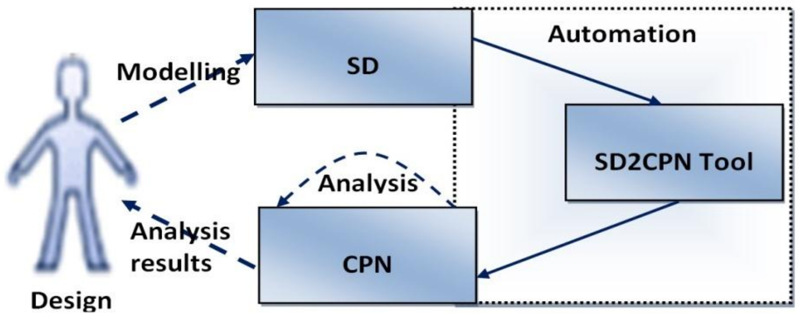


Figure 2. An overview of the designer's interaction with the tool

Figure 2 shows an overview of the interactions between a user and the SD2CPN tool. A user can model an SD and the tool converts it to the corresponding CPN using defined transformation rules [10]. The user's interaction with the SD2CPN tool is based on direct manipulation of either a graphical or a textual representation of a model. In order to support the user in modelling an SD, the GUI includes tool palettes with drag and drop capabilities. Alternatively, the user can also represent an SD textually using a textual notation (see section on Tool Evaluation: A Case Study) with the same expressiveness as that of the more popular graphical notation. Having an SD as the input, the SD2CPN tool generates a formal representation of the SD and synthesises the corresponding CPN for analysing the model. The tool simulates a token passing (the involvement of objects) between transitions and places, and keeps a track of the model's execution flow. Thus, the simulation of the CPN shows the system flow with object-orientation. The synthesised model can be used to validate the SD by reproducing the expected scenarios of the system behaviour.

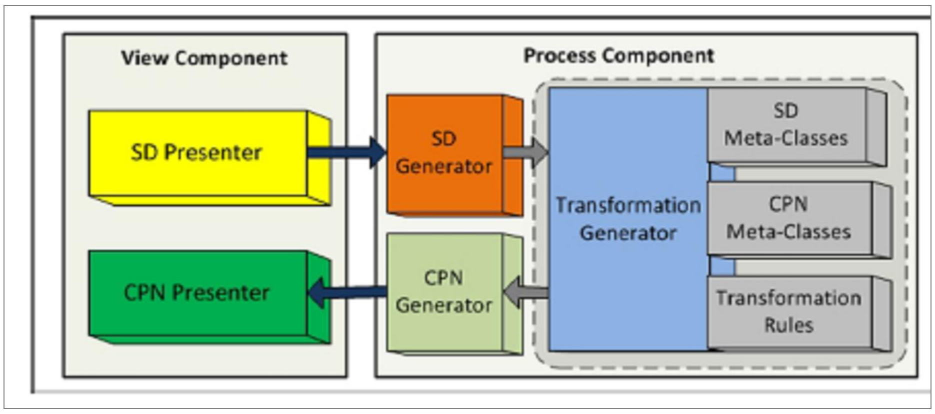


Figure 3. Design architecture of SD2CPN tool

Figure 3 shows the tool design that follows a component-based modular architecture. There are two main modules as follows: the view component that represents the front end of the tool and the process component that constitutes the back end with the transformations. The SD-presenter process is in charge of representing the input SD model either graphically or textually. Then the SD-generator process generates the corresponding formal representation of the SD model [10] and passes it to the transformation-generator process. Here, the transformations are performed by considering the meta-models of SD and CPN [17], and the formal transformation rules [10]. The transformation rules define the mapping of a given element in an SD meta-class to the corresponding element in the CPN meta-class. The CPN-generator process uses the outcome of the transformation process to synthesise the corresponding CPN. Then the CPN-presenter process shows the resulting CPN graphically or textually. The tool is implemented using Java NetBeans environment. NetBeans-visual-library API (application program interface) and Utilities API are used to implement the GUI-based classes in the view-component module. Standard Java programming is used to implement the classes in the process-component module.

TOOL IMPLEMENTATION

This section presents the implementation of the transformations in the SD2CPN tool. Scheme 1 states the execution of the process-component module during the transformation of an SD to a CPN by considering the general elements of the SD. The process starts by getting the input elements of the SD and identifying their instances and associations that correspond to the SD meta-model [17]. As the first step towards transformations, an object of an SD model and the corresponding object of the CPN are created with the same diagram name. While building the formal SD model [10], each SD element is transformed into the corresponding CPN element, leading to a complete CPN model [10].

Consider the procedure given in Scheme 1. If the element is an instance, it is added as a class 'Instance' of the SD meta-model and transformed into the class 'Colour' in the CPN meta-model. Then a state location associated with the instance is added to the SD model. This object 'StateLocation' is transformed into the object 'Place' with the corresponding colour and marking, which is then added to the CPN model.

```
Procedure GeneralTransformation(SD)
Input    Array of SD (name, instance, transition{send instance a , message label m, receive instance b})
         elements e[];
Output Coloured Petri Net cpn (name, place, net-transition, arc, variable, expression);
Begin
    i = 1,  j = 0, k = 0, p = 0, x = 0;
    Create an SD instance and a CPN instance
        SD sd = new SD();
        CPN cpn = new CPN();
    Assign the SD name
        sd.name = e[0];
    Transform SD name to CPN name
        cpn.name = sd.name;
    foreach(e[i]){
        if(e[i] == instance){
            Add an instance to SD
            sd.instance[j] = e[i];
            Transform SD instance to a colour in the CPN
            cpn.colour[j] = sd.instance[j];
            Add an initial state location to SD
            sd.state[j] = new stateLocation(sd.instance[j]);
            Transform the state location to a place in the CPN with corresponding colour and marking
            cpn.place[j] = sd.state[j](cpn.colour[j], 1);
            j := j+1;
        }
        else if(e[i] == transition){
            Add message label to SD
            sd.messageLabel[k] = e[i].m;
            Transform to the corresponding CPN label
            cpn.label[k] = sd.messageLabel[k];
            Initialise send and receive events of the transition
            sd.event a = e[i].a;
            sd.event b = e[i].b;
            Initialise send and receive state locations
            sd.state $s_1$= new sd.state(e[i].a);
            sd.state $s_2$ = new sd.state(e[i].b):
            Transform state location to the corresponding place in the CPN
            cpn.place[p] = TransformStateToPlace($s_1$, a);   p := p+1;
            cpn.place[p] = TransformStateToPlace($s_2$, b);  p := p+1;
            Add transition to SD
            sd.localTransition[k] = new sd.localTransition(sd.messageLabel[k], a, b, $s_1$, $s_2$);
            Transform transition to the corresponding net-transition in the CPN
            cpn.transition[k] = TransformTransitionToNetTransition(sd.localTransition[k]);
            Add corresponding arcs to CPN given the net-transition and respective places
            cpn.arc[x] = AddArc(cpn.place[p-4], cpn.transition[k]); x := x+1;
            cpn.arc[x] = AddArc(cpn.place[p-3], cpn.transition[k]); x := x+1;
            cpn.arc[x] = AddArc(cpn.transition[k], cpn.place[p-2]); x := x+1;
            cpn.arc[x] = AddArc(cpn.transition[k], cpn.place[p-1]); x := x+1;
        }
    Keep track of each element
    i:=i+1;
    }
    Display generated CPN
    display(cpn);
end
```

Scheme 1. Algorithm for general transformation function in SD2CPN tool

If the element is a local transition with a message label and the two events (corresponding to the sending instance and the receiving instance), the process starts by adding the corresponding object of the class 'messageLabel' and the objects of the associated classes, 'Event' and 'StateLocation' to the SD model (which is an object of the class SD). After that, the object of the LocalTransition is added to the SD object model. Thereafter, each of these objects, i.e. messageLabel, localTransition and stateLocations, are transformed into the objects of Label, NetTransition and Place respectively, and added to the CPN object model. The process retrieves the associated input places of the net-transition and adds the corresponding objects of Arc that links the net-transition and the associated places.

Next, the process updates the status of each object array to keep the flow control and retrieves the next object. When all the SD elements are transformed into the corresponding elements of the CPN, the tool displays the generated CPN by calling the process within the view component.

With respect to the interaction-fragment behaviour, the above algorithm is extended according to the formal transformation rules defined in our previous work [10-12]. Here, when the element is a fragment, the process adds an object of the class 'InteractionFragment' to the SD model [17]. It also adds the associated objects of the class 'Event' to the beginning of the fragment and the objects of the class 'StateLocation' for each operand. The beginning of the fragment is transformed into an object of the class 'NetTransition' (unlabelled) while the created object of the class 'StateLocation' is transformed into the corresponding object of the class 'Place' in the CPN meta-model [17]. By linking with the previous places (the source place of an arc), the objects of the class 'Arc' are created between the net-transition and the associated places. The interactions (local transitions in an SD) within each of the operand are executed according to the general transformation algorithm given above.

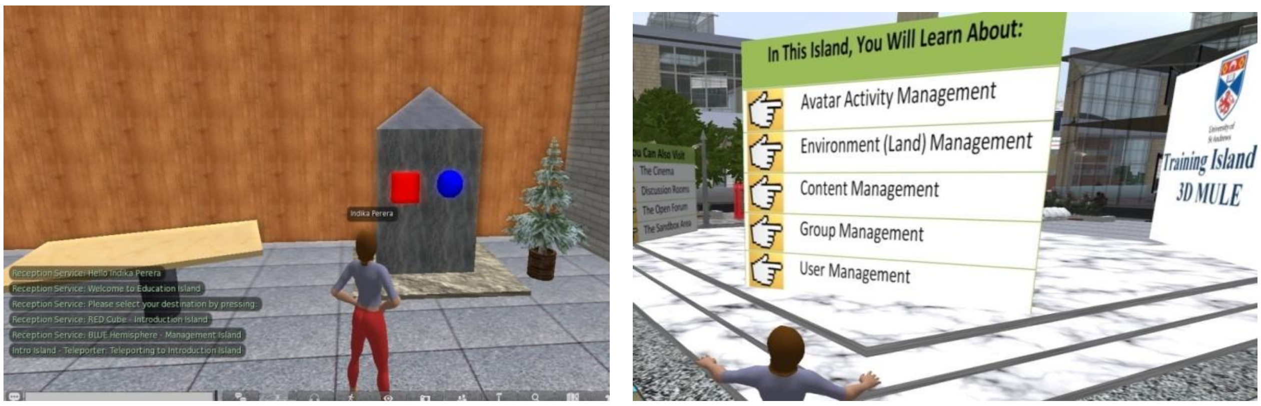
When the process encounters a local transition outside the fragment, the end of the fragment is processed. Here, the objects of the class 'Event', which are associated with the end of the fragment, and the objects of the class 'StateLocation' after the fragment, are added to the SD model and transformed into the corresponding objects of the classes 'NetTransition' and 'Place' in the CPN model respectively. After that, the objects of the class 'Arc' are added to the CPN model. Further, the guard expressions and the associated variables are processed.

TOOL EVALUATION: A CASE STUDY

In order to evaluate the SD2CPN tool, we used an immersive environment development project with educational content manipulation. Three-dimensional multi-user virtual environments (MUVEs), also known as immersive environments or virtual worlds, provide engaging and dynamic user interactions through the means of virtual persona called avatar. The virtual worlds consist of specific features [18], thus useful in many application domains such as education, entertainment, gaming and simulation. However, these features must be carefully considered when programming the dynamic behaviour inside these virtual worlds, so it can be a challenging task. Here, we used an educational environment based on Open Simulator (OpenSim) [21], which is one of the prominent open-source MUVE platforms. With an architecture that supports virtual world development and plugins for additional functions, OpenSim has become a popular choice for academics to start their MUVE-supported teaching or learner support activities [22].

Although the SD2CPN tool could be used for complex interactions, for brevity of analysis, the following basic scenario was selected to indicate a real usage of this tool. This scenario has been implemented through the in-world and content-based scripting using Linden Scripting Language inside the education-island in OpenSim virtual world, as part of the teaching and learner support islands we have developed [22].

When an avatar of a student comes near the reception desk of the education-island (within the range of 2-metre radius) the reception service welcomes the avatar with a personalised greeting message: "Hello <*avatar name*>. Welcome to the education-island." The reception desk is provided with two buttons for initiating a teleporting process to their corresponding islands: introduction-island (push red cube button) and management-island (push blue sphere button). After the welcome message is presented, the avatar is invited to select its preferred destination from the given two buttons. Once the avatar pushes the button for a selected destination, the reception service initiates the teleporting process. Then the teleporter service processes the avatar information and links the avatar with the destination-island simulation. Finally, the avatar is located within the destination island. Figure 4(a) shows an avatar interacting with the reception desk and Figure 4(b) shows the avatar being teleported to the chosen destination island.


Reception Service: Hello Indika Perera
Reception Service: Welcome to Education Island
Reception Service: Please select your destination by pressing:
Reception Service: RED Cube - Introduction Island
Reception Service: BLUE Hemisphere - Management Island
Intro Island - Teleporter: Teleporting to Introduction Island

In This Island, You Will Learn About:
Avatar Activity Management
Environment (Land) Management
Content Management
Group Management
User Management

.

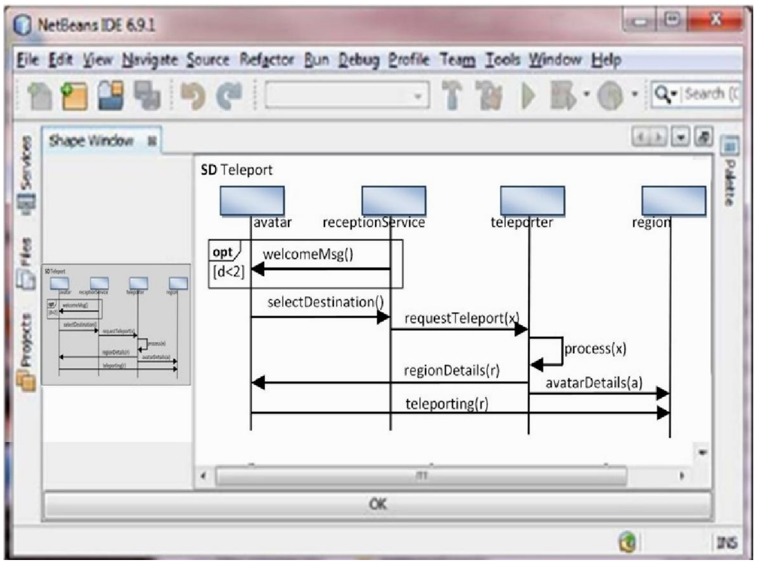
(a) (b)

Figure 4. (a) An avatar nears reception desk in the education-island; (b) An avatar in a region after being teleported from the education-island

Figure 5(a) shows an SD named Teleport that represents the interactions associated with the teleport functionality of an avatar. The SD contains four instances, namely avatar, receptionService, teleporter and region. When the distance between the avatar and the reception service is less than 2 metres, the appearance of the welcome message is represented by an opt interaction-fragment (optional behaviour) in the SD. Then the remaining interactions occur between the instances. Here, opt interaction-fragment denotes a choice of behaviour, where either the operand happens or it does not. The interactions within the operand are executed only if the guard-condition is evaluated to true. If it is evaluated to false, then the interactions within opt operand are ignored and the remainder of the interactions in the SD are continued with the execution.

The text-based input and output representations are included in the SD2CPN tool to facilitate the integration of transformations with the existing tools. The grammar for these representations is defined in Backus-Naur Form, which describes the syntax of a modelling language [20].

The text-based representation for the teleport scenario, which is an alternative input to the tool, is shown in Figure 5(b). For the SD named Teleport, each instance is specified with an identifier followed by its name, e.g. `a: avatar`. The beginning of opt fragment is represented by `Beg Frag 1: opt`, indicating the fragment identifier and type. The operand and associated condition of the fragment is specified as `Beg Op 1:1 [d<2]` and followed by the instance identifiers `a, s` that are involved in the operand. The textual representation for the first local transition, i.e. transition: s, welcomeMsg, a, specifies that the transition with the message label 'welcomeMsg' is sent from instance s to instance a. Next, the end statement for the operand is specified. After all the interactions are stated, the end of the diagram is indicated by SD end.



(a)

```
SD Teleport begin:
instance a:avatar
instance s:receptionService
instance t:teleporter
instance r:region
Beg Frag 1: opt
Beg Op 1: 1 [d<2]; a, s
transition: s,welcomeMsg,a
End Op: 1:1
End Frag 1;
transition: a,selectDestination,s
transition: s,requestTeleport,t
transition: t,process,t
. . .
. . .
SD end.
```

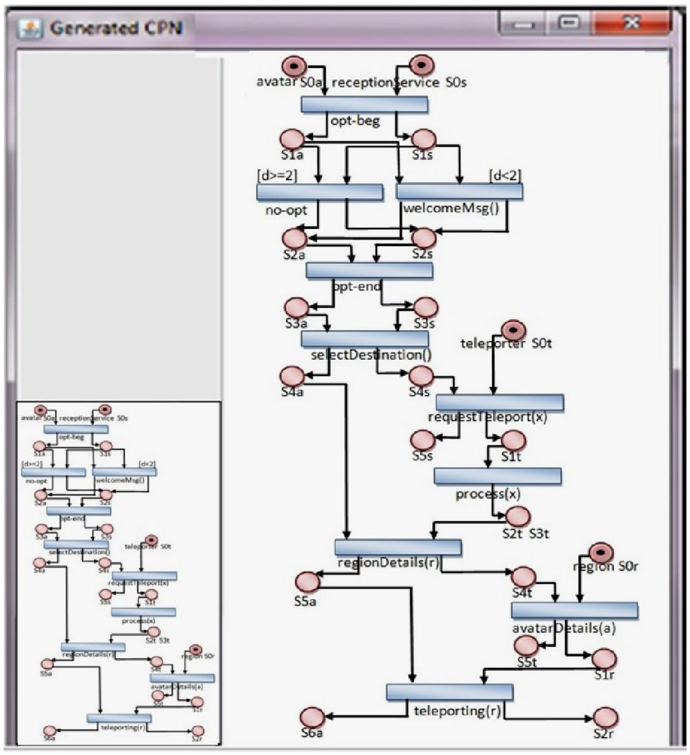
(b)

Figure 5. (a) SD modelled using SD2CPN; (b) Textual representation

Figure 6(a) shows the CPN representation after transforming the SD Teleport. By applying the formal transformations defined in our previous work [10-12, 17, 18], the CPN was generated with the following colours: avatar, receptionService, teleporter and region, with the corresponding identifiers a, s, t, and r respectively. Here, the state locations and the local transitions of the SD are mapped to the places and the net-transitions of the CPN respectively, and the arcs create the links between the places and the net-transitions as expected.

When transforming an SD with the opt interaction-fragment, the corresponding CPN representation contains two net-transitions, namely opt-beg and opt-end, to synchronise the behaviour of the interaction-fragment at the beginning and end respectively [10]. The places and the net-transitions within opt-beg and opt-end describe the same sequence of interaction as in the SD. The guard-condition of the fragment is associated with the first net-transition after opt-beg. Additionally, a new net-transition, namely no-opt, is defined in the CPN. The no-opt net-transition is associated with a guard-condition that is the negation of the disjunction of the condition in the enclosing *opt* interaction-fragment. This new net-transition is linked with the places that correspond to the minimum and maximum state locations within the fragment of an instance [10].

Figure 6(b) shows the textual representation of the CPN obtained from the SD2CPN tool. The CPN consists of four colours (a, s, t, r), associated net-transitions, places and arcs. The textual statement place: S0a:1 indicates that the place S0a contains one token. The statement arc: A_S0a To 1:opt-beg specifies that there is an arc where the source element is the place S0a and the target element is the net-transition 1:opt-beg. The first net-transition (unlabelled) is represented by transition: 1: opt-beg, which corresponds to the beginning of the option fragment in the SD. Remaining statements are continued based on the execution order. In order to indicate the end of the CPN the string, CPN end, is presented at the end.

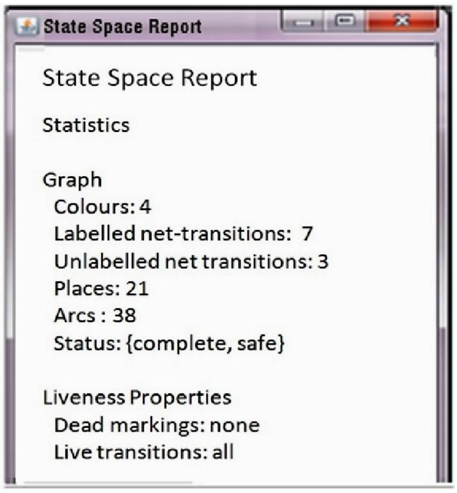


```
CPN Teleport begin:
colour a:avatar
colour s:receptionService
colour t:teleporter
colour r:region
place: S0a :1
place: S0s :1
place: S0t :1
place: S0r :1
arc: A_S0a To 1:opt-beg
arc: A_S0s To 1: opt-beg
transition: 1: opt-beg
arc: A_opt-beg To S1a
arc: A_opt-beg To S1s
place: S1a
place: S1s
arc: A_S1a To 2:welcomeMsg
arc: A_S1s To 2:welcomeMsg
transition: 2:welcomeMsg[(d<2)==T]
arc: A_welcomeMsg To S2a
arc: A_welcomeMsg To S2s
. . .
. . .
CPN end.
```

(a) (b)

Figure 6. (a) CPN generated by SD2CPN; (b) Textual representation of CPN

Since a CPN constitutes a single coherent description of the behaviour specified by the SD, the simulation flow of a CPN represents the communication between each object. Our tool follows the approach given by Jensen and Kristensen [7] to generate the state space report and simulation report. Figures 7(a) and 7(b) show the state space report and simulation report of the teleport scenario respectively. The simulation report lists the occurrences of places, arcs and net-transitions. According to the state space report, the system design model we made does not contain unsafe or intentionally unreachable places; therefore the tool allows the design verification.

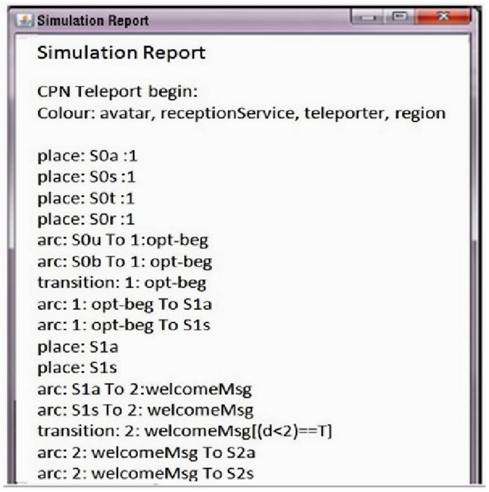

State Space Report

State Space Report

Statistics

Graph
Colours: 4
Labelled net-transitions: 7
Unlabelled net transitions: 3
Places: 21
Arcs : 38
Status: {complete, safe}

Liveness Properties
Dead markings: none
Live transitions: all


```
Simulation Report

Simulation Report

CPN Teleport begin:
Colour: avatar, receptionService, teleporter, region

place: S0a :1
place: S0s :1
place: S0t :1
place: S0r :1
arc: S0u To 1:opt-beg
arc: S0b To 1: opt-beg
transition: 1: opt-beg
arc: 1: opt-beg To S1a
arc: 1: opt-beg To S1s
place: S1a
place: S1s
arc: S1a To 2:welcomeMsg
arc: S1s To 2: welcomeMsg
transition: 2: welcomeMsg[(d<2)==T]
arc: 2: welcomeMsg To S2a
arc: 2: welcomeMsg To S2s
```

(a) (b)

Figure 7. (a) State space report of generated CPN; (b) Simulation report of generated CPN

The SD2CPN tool supports the examination of properties such as reachability and liveness of the modelled system. The reachable CPN places are extracted by referring to the place list given by the simulation report. The liveness property is monitored by considering the net-transitions that are enabled, hence fired. Also, the simulation of the CPN model shows the token passing from one place to another via a net-transition. Thus, we can see the object involvement within the system execution and how it depicts the object-oriented features in the original SD. The tool facilitates keeping track of the actions that are executed and the states that are reached in the system design. These two reports can be used to locate errors or increase the confidence in the correctness of the software system being designed.

CONCLUSIONS

We have discussed the need for tools that support MDD as a growing area of interest for software architecture and design. The SD2CPN tool presented in this article harnesses the benefits of a graphical modelling language (i.e. UML) and a formalism (i.e. CPN), to bridge two popular yet uniquely distinct software design methodologies. The focused type of UML diagrams for the tool is SD; however, the modular architecture of the tool makes it possible to extend model transformations to other UML diagram types with minimum modifications.

The SD2CPN tool supports a forward transformation through automation; it can be extended to incorporate back annotating of the analysis results to the source SD model, thereby providing a complete automation cycle for model transformation. Further, the platform-independent implementation of the SD2CPN tool can be wrapped with a plug-in layer for the existing modelling and analysis tools by using the extensibility features given in the textual notations.

REFERENCES

1. A. Kleppe, J. Warmer and W. Bast, “MDA Explained: The Model Driven Architecture: Practice and Promise”, 1st Edn., Addison-Wesley Longman Publishing Co., Inc. Boston, **2003,** pp.12-22.
2. T. Stahl, M. Völter, J. Bettin, A. Haase and S. Helsen, “Model Driven Software Development: Technology, Engineering, Management”, 1st Edn., John Wiley and Sons, New York, **2006,** pp. 13-21.
3. S. Sendall and W. Kozaczynski, “Model transformation: The heart and soul of model-driven software development”, *IEEE Softw.*, **2003**, *20*, 42-45.
4. M. B. Kuznetsov, "UML model transformation and its application to MDA technology", *Program. Comput. Softw.*, **2007**, *33*, 44-53.
5. J. Arlow and I. Neustadt, “UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design”, 2nd Edn., Addison-Wesley Professional, Boston, **2005**, pp.273-282.
6. Object Management Group, “OMG unified modelling language: Superstructure”, **2011,** http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF, pp.473-533 (Accessed: June 2015).
7. K. Jensen and L. M. Kristensen, "Coloured Petri Nets: Modelling and Validation of Concurrent Systems", 1st Edn., Springer, Berlin, **2009**, pp.313-361.
8. K. Jensen, "An introduction to the theoretical aspects of coloured Petri-nets", in “A Decade of Concurrency Reflections and Perspectives” (Ed. J. W. de Bakker, W. P. de Roever and G. Rozenberg), Springer-Verlag, London, **1993**, Ch.6.

9. G. Gelen and M. Uzam, "Novel analysis of Petri-net-based controllers by means of TCT implementation tool of supervisory control theory", *Maejo Int. J. Sci. Technol.*, **2010**, *4*, 360-396.
10. J. Bowles and D. Meedeniya, "Formal transformation from sequence diagrams to coloured Petri nets", Proceedings of 17th Asia Pacific Software Engineering Conference, **2010**, Sydney, Australia, pp.216-225.
11. J. Bowles and D. Meedeniya, "Strongly consistent transformation of partial scenarios", *ACM SIGSOFT Softw. Eng. Notes*, **2012**, *37*, 1-8.
12. J. Bowles and D. Meedeniya, "Parametric transformations for flexible analysis", Proceedings of 19th Asia Pacific Software Engineering Conference, **2012**, Hong Kong, pp.634-643.
13. M. A. Ameedeen, B. Bordbar and R. Anane, "Model interoperability via model driven development", *J. Comput. Syst. Sci.*, **2011**, *77*, 332-347.
14. O. R. Ribeiro and J. M. Fernandes, "Some rules to transform sequence diagrams into coloured Petri nets", Proceedings of 7th Workshop and Tutorial on Practical Use of Coloured Petri nets and the CPN tools, **2006**, Aarhus, Denmark, pp.237-256.
15. L. Wen, D. Tuffley and R. G. Dromey, "Formalizing the transition from requirements' change to design change using an evolutionary traceability model", *Innovat. Syst. Softw. Eng.*, **2014**, *10*, 181-202.
16. K. Ahmed, T. Myers, L. Wen and A. Sattar, "Detecting requirements defects utilising a mathematical framework for behavior engineering", *Int. J. Soft Comput. Softw. Eng.*, **2013**, *3*, 187-198.
17. D. Meedeniya, J. Bowles and I. Perera, "SD2CPN: A model transformation tool for software design models", Proceedings of 18th International Computer Science and Engineering Conference, **2014**, Khon Kaen, Thailand, pp.461-466.
18. D. A. Meedeniya and I. Perera, "Model based software design: Tool support for scripting in immersive environments", Proceedings of IEEE 8th International Conference on Industrial and Information Systems, **2013**, Peradeniya, Sri Lanka, pp.248-253.
19. D. A. Meedeniya, I. Perera and J. Bowles, "Transformation and composition of software design models for Model Driven Development ", Proceedings of IEEE 10th International Conference on Industrial and Information Systems, **2015**, Peradeniya, Sri Lanka, pp.31-36.
20. M. A. Reniers, "Message sequence chart: Syntax and semantics", *PhD Thesis*, **1999**, Eindhoven University of Technology, Netherlands.
21. The Open Simulator Project, "Open Simulator", **2007**, http://opensimulator.org/wiki/Main_Page (Accessed: June 2015).
22. C. Allison, A. Miller, T. Sturgeon, I. Perera and J. McCaffery, "The third dimension in open learning", Proceedings of 41st Frontiers in Education Conference, **2011**, Rapid City (SD), USA, pp.1-6.