International Journal on Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

An Integrated Framework for Checking the Behaviour of fUML Models Using CSP

Islam Abdelhalim · Steve Schneider · Helen Treharne

Received: date / Accepted: date

Abstract Transforming UML models into a formal representation to check certain properties has been addressed many times in the literature. However, the lack of automatic formalization for executable UML models and provision of model checking results as modeller-friendly feedback has inhibited the practical use of such approaches in real life projects. In this paper we address those issues by performing the automatic formalization of the fUML (Foundational subset for executable UML) models into CSP without any interaction with the modeller, who should be isolated from the formal methods domain. The formal analysis provides the modeller with a UML sequence diagram that represents the model checking result in the case where an error has been found in the model. This work also considers the formalization of systems that depend on asynchronous communication between components in order to allow checking of the dynamic concurrent behaviour of systems.

We have designed a comprehensive framework that is implemented as a plugin to MagicDraw (the CASE tool we use) that we call Compass. The framework depends on Epsilon as a model transformation tool that utilizes the Model Driven Engineering (MDE) approach. It also implements an optimization approach to be able to model check concurrent systems using FDR2, and at the same time comply with the fUML inter-object communication mechanism. In order to validate our framework, we have checked a Tokeneer fUML model against deadlock using Compass. The model checking results are reported in this paper showing the advantages of our framework.

Keywords Formalization · fUML · CSP · Deadlock · Model Checking · FDR2 · Tokeneer

I. Abdelhalim · S. Schneider · H. Treharne
Department of Computing, University of Surrey, UK
E-mail: i.abdelhalim@surrey.ac.uk

S. Schneider
E-mail: s.schneider@surrey.ac.uk

H. Treharne
E-mail: h.treharne@surrey.ac.uk

1 Introduction

Formal methods benefits from its mathematically rigorous representation that enables automatic analysis using model checkers and theorem provers. However, not many software engineers (modellers) have the specialist mathematical knowledge to model their industrial size systems formally. On the other hand, semi-formal modelling notations, such as UML (Unified Modeling Language) [25], are easy to use and understand by software engineers, making UML the *de-facto* standard for modelling object oriented systems. The impossibility of automated analysis or checking of the UML models, made it very risky to use UML in modelling safety-critical systems.

Much work has been done to make use of the two domains' advantages (formal and semi-formal modelling) by letting the modeller develop the system model using UML and then automatically transforming it to a formal representation which can be checked against certain properties. Throughout the paper we will refer to this process as "formalization".

By reviewing and analyzing the previous work (refer to Section 9 for more details) we have observed several issues that we consider are the main barriers for the practical use of UML formalization in real life projects. First, the avoidance of having a comprehensive framework that isolates the modeller from dealing with the formal methods, and at the same time integrates with the current case tools. This isolation requires providing the modeller with modeller-friendly debug feedback in case of a problem in the checked model. Second, asynchronous inter-object communication has been addressed rarely in this field, yet in many systems this kind of communication is preferred due to its simplicity and modularity compared to other ways of communication that require tight synchronization between the system's objects (e.g., using a clock). Finally, using UML as a semi-formal language requires tremendous effort to formalize such a huge standard, which has been developed mainly to provide modellers with a multi-view modelling approach. Moreover, formalizing the UML models cannot be a direct process because of its excessive flexibility which increases the gap between it and the corresponding formal model.

The main originality of our work comes from addressing the aforementioned issues. We propose a comprehensive framework that uses fUML (Foundational subset for executable UML) [26] as a semi-formal modelling language. Compared to UML, fUML is a more restricted subset of the UML2 standard that has a well defined structural and behavioural semantics. Our framework also isolates the modeller from the formal methods domain through the whole model checking cycle from the beginning until providing him with a UML sequence diagram (modeller-friendly) that describes a problem scenario (if found). We have implemented this framework as a plugin that integrates with MagicDraw [1], the case tool we use in this work.

We also consider in this work the formalization of the asynchronous communication mechanism between the system objects. We took the well defined specification of the inter-object communication in the fUML standard and formalized it in CSP (Communicating Sequential Processes) [15]. Although the standard was clear in defining this mechanism, it left the event dispatch scheduling (how are signals processed when received?) as a *semantic variation point* to be defined by the fUML execution engine implementor. The formalization of this point allowed us to test different interpretations.

Having the inter-object communication mechanism formalized allowed for checking overall system behaviours. In this paper we will focus on deadlock freedom only as a sample system behaviour to check. We also chose the Tokeneer project [3] as a case study to validate our framework.

This paper extends our previous paper [1] on this area; it introduces the formalization framework that automates the transformation process using Epsilon [18] as an MDE (Model Driven Engineering) framework. We developed a group of Epsilon transformation rules which depend on the available fUML [26] and CSP [33] meta-models. This paper also considers the automatic generation of a sequence diagram that represents the counter-example in case of deadlock.

The rest of this paper is organised as follows. In Section 2, we give a brief background about the fUML standard and CSP. In Section 3, we introduce the Tokeneer project as the used case study in this work. In Section 4, we give an overview of the formalization framework. In Section 5, we describe the Model Formalizer, the most important component in the framework. In Section 6, we describe the role of FDR2 to check the model against deadlock. In Section 7, we describe how the framework automatically provides modeller-friendly feedback. In Section 8, we outline the implementation of the framework as a plugin to MagicDraw. Finally, we discuss related work and conclude in Sections 9 and 10 respectively.

2 Background

2.1 fUML

As defined by OMG, fUML (Foundational Subset for Executable UML) acts as an intermediary between "surface subsets" of UML models and platform executable languages (e.g., Java) [26]. fUML models are executable models, which means they can be used by code-generators to generate full executable code directly from the models, or model-interpreters that rely on a virtual machine to directly read and run the models (e.g., fUML Reference Implementation [20]).

The fUML specification is a subset of the original UML2 specification [25]. This subset was defined by specifying modifications to the original abstract syntax (of UML2) of the class and activity diagrams. These modifications are specified in clause 7 of the standard [26] by merging/excluding some packages in the UML2 specification, as well as adding new constraints.

As defined in the fUML standard, we are listing below some of the modifications to UML2 that are relevant to

[1] MagicDraw is an (award-winning) architecture, software and system modeling case tool. It also supports additional plugins to increase its functionalities

our case study (Tokeneer ID Station) fUML model. All of those modifications are related to the fUML activity diagrams since our goal is to capture the behaviour of our model:

– Central buffer nodes are excluded from fUML because they were judged to be unnecessary for the computational completeness of fUML.
– Variables are excluded from fUML because the passing of data between actions can be achieved using object flows.
– Exception handlers are not included in fUML because exceptions are not included in fUML.
– Opaque actions are excluded from fUML since, being opaque, they cannot be executed.
– Value pins are excluded from fUML because they are redundant with the use of value specifications to specify values.

The operational semantics of fUML is an executable model with methods written in Java, with a mapping to UML activity diagrams. The declarative semantics of fUML is specified in first order logic and based on PSL (Process Specification Language) [12].

Inter-object communication mechanism in fUML

This part gives an overview of the semantics of the inter-object communication in fUML as defined by clause 8 in the standard [26]. Such communication is conducted between active objects only. Active objects in fUML communicate asynchronously via signals. Each active object is associated with an *object activation* which handles the dispatching of asynchronous communications received by its active object. Figure 1 shows the structure related to *object activation*.

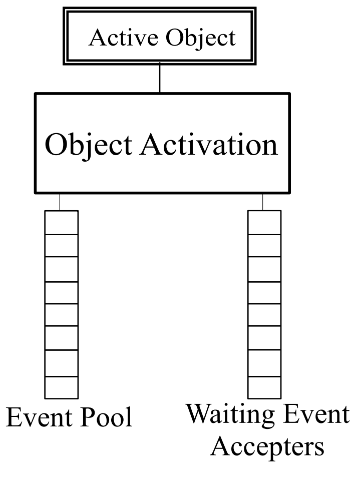


Fig. 1: Object Activation Structure

Object activation maintains two main lists: the first list (*event pool*) holds the incoming signal instances waiting to be dispatched, and the second list (*waiting event accepters*) holds the event accepters that have been registered by the executing classifier behaviour. Event accepters are allowable signals with respect to the current state of the active object.

The fUML standard permits the specifier (tool implementer) to define a suitable dispatching mechanism for signals within the *event pool* (semantic variation point). The default dispatching behaviour, as described in [26], dispatches events on a FIFO (first-in first-out) basis.

2.2 CSP

CSP [15] is a modelling language that allows the description of systems of interacting processes using a few language primitives. Processes execute and interact by means of performing events drawn from a universal set Σ. Some events are of the form $c.v$, where c represents a channel and v represents a value being passed along that channel. Our UML/fUML formalization considers the following subset of the CSP syntax:

$$\begin{aligned} P ::= {} & a \rightarrow P \ \mid \ c?x \rightarrow P(x) \ \mid \ d!v \rightarrow P \\ & \mid \ c!v?x : E \rightarrow P(x) \ \mid \ P_1 \, \Box \, P_2 \\ & \mid \ P_1 \sqcap P_2 \ \mid \ P_1 \underset{A\ B}{\|} P_2 \ \mid \ P_1 \underset{A}{\|} P_2 \ \mid \ P \setminus A \\ & \mid \ \mathit{let}\ N_1 = P_1\ ,\ \dots\ ,\ N_n = P_n\ \mathit{within}\ N_i \\ & \mid \ \mathit{if}\ \ b\ \mathit{then}\ P_1\ \mathit{else}\ P_2 \end{aligned}$$

The CSP process $a \rightarrow P$ initially allows event a to occur and then behave subsequently as P. The input process $c?x \rightarrow P(x)$ will accept a value x along channel c (corresponding to performance of the event $c.x$) and then behave subsequently as $P(x)$. The output process $c!v \rightarrow P$ will output v along channel c (corresponding to performance of the event $c.v$) and then behave as P. Processes interact by synchronising on the events $c.v$. Channels can have any number of message fields, as a combination of input and output values, for example: $c!v?x : E \rightarrow P(x)$. Also x can be constrained to be a value from the set E.

The choice $P_1 \, \Box \, P_2$ offers an external choice between processes P_1 and P_2 whereby the choice is made by the environment. Conversely, $P_1 \sqcap P_2$ offers an internal choice between the two processes.

The parallel combination $P_1 \underset{A\ B}{\parallel} P_2$ executes P_1 and P_2 in parallel. P_1 can perform only events in the set A, P_2 can perform only events in the set B, and they must simultaneously engage in (i.e., synchronise on) events in the intersection of A and B. The interface parallel $P_1 \underset{A}{\parallel} P_2$ requires synchronization only on those events in the common set (interface) A.

The process $P \setminus A$ behaves like P except that the events from A have been internalized. In other words, all these events are removed from the interface of the process and no other process will be able to engage with them. The *let . . . within* statement defines P with local definitions $N_i = P_i$. The conditional choice *if b then* P_1 *else*P_2 behaves as P_1 or P_2 depending on the evaluation of the condition b.

3 Tokeneer: Case study introduction

The Tokeneer project [3] is one of the most interesting pilot projects forming part of the Verified Software Initiative [16], and has been cited by the US National Academies as exemplifying best practice in software engineering [17]. The project was certified to Common Criteria Level 5 and in the areas of specification, design and implementation achieving Levels 6 and 7. The Tokeneer project re-developed one component of a Tokeneer system that was developed by the NSA (National Security Agency) to provide protection to secure information held on a network of workstations situated in a physically secure enclave. A survey of other projects using formal methods has been discussed in [38].

The entire project archive has been released [2] for experimentation by researchers. This includes the project specifications written in Z [4] and an open source implementation. Woodcock and Aydal [37] have conducted several experiments using model-based testing techniques to discover twelve anomalous scenarios which challenged the dependability claims for Tokeneer as a security-critical system. Several of the scenarios highlight the importance of the behaviour of the user because one of the security objectives for Tokeneer is to prevent accidental, unauthorised access to the enclave by a user. The user was not formally modelled in the Z specification [2]. We also note the importance of modelling the user in our analysis.

Our motivation for using the Tokeneer project as a case study was not to re-validate the project but rather to investigate the concurrent behaviour of the various components of the Tokeneer ID station (TIS) subsystem in the context of asynchronous communication.

The correspondence between the Tokeneer formal specifications [2] and our Tokeneer fUML model is not a one-one relationship. Our Tokeneer fUML model contains more implementation details that are abstracted in Tokeneer Z specifications. Therefore, our formal analysis benefits from being able to examine the low level details of asynchronous communication. Such an analysis allows us to investigate potential deadlocks which might occur if the formal specifications were implemented using such communication mechanisms.

3.1 TIS subsystem structure

The components of interest in the TIS subsystem are represented on the class diagram in Figure 2. We do not formalize the class diagram, and its inclusion is just to illustrate the relationship between the system's components.

Door: This is the physical enclave's door that the user opens to access the secure enclave. It has no intelligent behaviour as it is entirely controlled by the door controller component. The two main attributes of this component are: *isOpen* attribute which indicates the status of the door (opened or closed), and the *isLocked* attribute which indicates the status of the door's latch (locked or unlocked).

Door Controller: This component controls the door's latch status (*isLocked*) by setting its value based on the incoming signals from the User Panel. It also manages two timers: the first timer watches if the door is kept closed and unlocked, and the second timer watches if the door is kept opened and locked.

User: This component models the user behaviours toward the system. He is responsible for requesting the enclave entry, and opening the door in case it was successfully unlocked by the User Panel. He is also responsible for closing the door after accessing the enclave. The system may serve more than one user at the same time. However, the results in this paper focus on a single user only.

User Panel: This component models the behaviour of the panel with which the user interfaces to gain access to the enclave. It is responsible for deciding whether the user is allowed to access the enclave or not.

Alarm: This component holds the status of the alarm (alarming or silent), based on the setting/resetting by the Door Controller component to the *isAlarming* attribute.

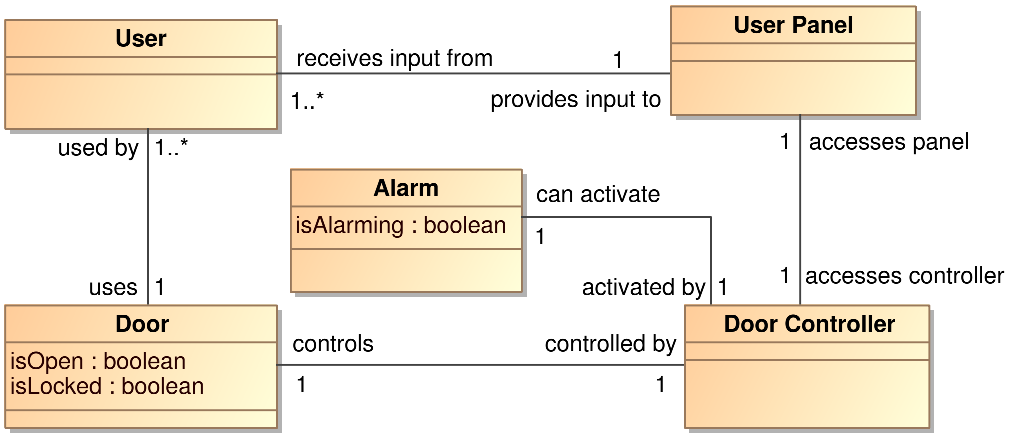


Fig. 2: TIS Class Diagram

3.2 TIS subsystem behaviour

In the Tokeneer fUML model all objects (of the above classes) which have interesting behaviour have associated activity diagrams. The Alarm object is a simple data holder and thus no activity diagram is associated with it. For the purpose of this paper, we choose to focus on a segment of the Door Controller activity (depicted in Figure 3), which includes all the described elements in Section 5.1.

Initially, the Door Controller waits for the *unlockLatchSignal* to be sent by the User Panel when the User requests an entry to the enclave and he is authorized to do so. When receiving this signal, the Door Controller changes the status of the Door's lock to be *Unlocked* by setting the attribute *isLocked* to FALSE. Consequently, the Door Controller sends *unlockingDoorCompleteSignal* to the User Panel to indicate the completion of the Door unlocking. At this point, the Door Controller starts a timer to watch if the User did not open the Door after getting the permission for entry. The two possible scenarios for the timer expiry (*lockTimeoutExceeded* or *lockTimeoutNotExceeded*) are represented as an internal decision. The *lockTimeoutNotExceeded* choice corresponds to the door opening within the allowed time. If the timer timeouts the Door Controller sends the *lockLatchSignal* to itself to change the Door's lock status to *Locked*. Otherwise, the Door Controller will accept the *doorIsOpenSignal* from the Door's object to continue its normal behaviour until sending the *entryAuthorizedSignal* to the User's object.

4 Framework overview

In this work we propose a framework that allows fUML models to be formalized to CSP automatically and checked for deadlock using FDR2. The framework also translates FDR2 output to a modeller friendly format (UML sequence diagram). Figure 4 shows the overall architecture of this framework and the used components.

Initially, the modeller develops the system fUML model using the case tool (MagicDraw). The model should include an fUML activity diagram for each active class in the system to describe its behaviour. Based on a feature in the case tool, the framework exports the fUML model into an XMI (XML Metadata Interchange) [24] format, thus it can be read by any MDE framework for transformation.

At this point, the Model Formalizer reads the fUML model (represented in XMI) and transforms it to a CSP script based on the available fUML [35] and CSP [33] meta-models. The Model Formalizer uses the Epsilon model management framework to perform the model-to-model and model-to-text tasks. The generated CSP script contains a process for each active class in the system, as well as a formalization for the inter-object communication mechanism to allow those processes to communicate with each other asynchronously via signals. The Model Formalizer also generates an Object-to-Class mapping table, which will be used for traceability to relate the modeller-friendly feedback to the original fUML model. In the case of a problem during the formalization process (e.g., an fUML activity diagram without a connected initial node cannot be formalized), the Model Formalizer generates the Formalization Report which reports the error(s) in the fUML model which led to this problem.

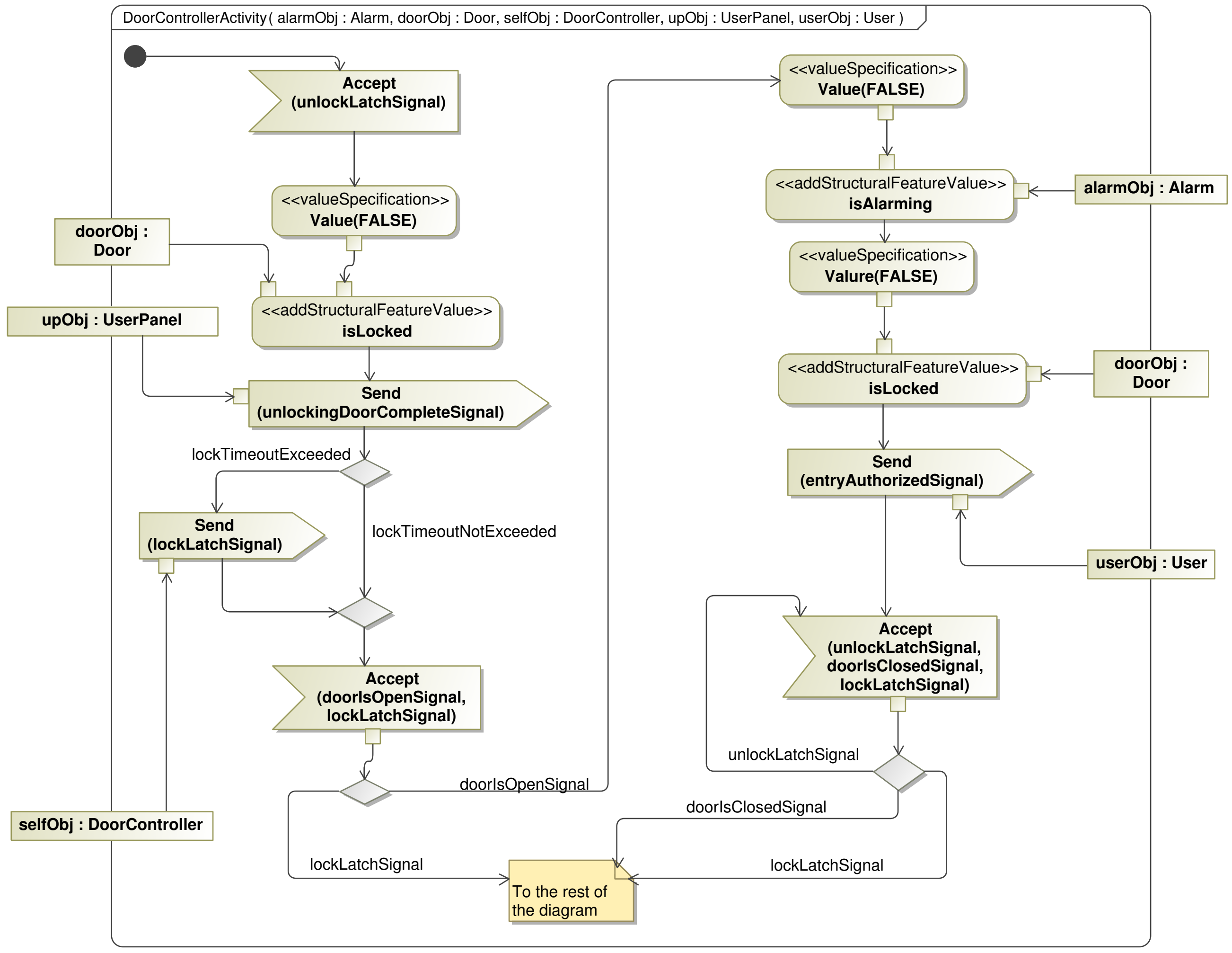

DoorControllerActivity(alarmObj : Alarm, doorObj : Door, selfObj : DoorController, upObj : UserPanel, userObj : User)
Accept (unlockLatchSignal)
<<valueSpecification>> Value(FALSE)
doorObj : Door
upObj : UserPanel
<<addStructuralFeatureValue>> isLocked
Send (unlockingDoorCompleteSignal)
lockTimeoutExceeded
lockTimeoutNotExceeded
Send (lockLatchSignal)
Accept (doorIsOpenSignal, lockLatchSignal)
doorIsOpenSignal
lockLatchSignal
selfObj : DoorController
To the rest of the diagram
<<valueSpecification>> Value(FALSE)
<<addStructuralFeatureValue>> isAlarming
alarmObj : Alarm
<<valueSpecification>> Valure(FALSE)
<<addStructuralFeatureValue>> isLocked
doorObj : Door
Send (entryAuthorizedSignal)
userObj : User
Accept (unlockLatchSignal, doorIsClosedSignal, lockLatchSignal)
unlockLatchSignal
doorIsClosedSignal
lockLatchSignal

Fig. 3: Segment of the Door Controller Activity

Consequently, the framework launches FDR2 to check the generated CSP script for deadlocks. In case of deadlock, FDR2 generates a counter-example which includes the traces (sequence of events) that led to the deadlock. The UML Sequence Diagram Generator reads this counter-example and visualizes it in the form of a UML sequence diagram making use of the information stored in the Object-to-Class mapping table. The generated sequence diagram represents the deadlock scenario in a modeller friendly format which visualizes the objects interactions in a chronological order.

The following sections provide more detail regarding each component included in this framework.

5 The Model Formalizer

The main functionality of the Model Formalizer component is to translate the input fUML model to CSP. The component achieves this translation in three stages:

1. Translating the fUML activity diagrams into CSP processes.
2. Generating CSP processes that represents the inter-object communication mechanism.
3. Combining all the previous CSP processes into one single process that represents the whole system (SYSTEM).

The following sections describe each of those stages and how the Model Formalizer automates the formalization.

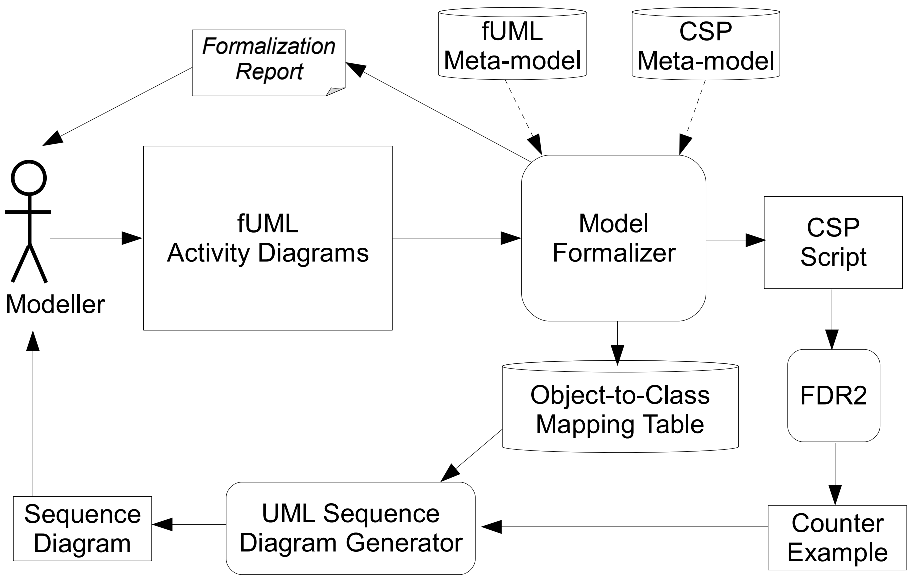


Fig. 4: Framework Architecture

5.1 fUML activity diagrams formalization

We perform the translation from fUML activity diagrams to CSP based on a collection of mapping rules. Table 1 shows the fUML activity diagram's elements and the corresponding CSP representation that reflects the semantic behaviour for each element.

In the mapping rules, *aIH* and *bIH* represent the instance handler of the sender and receiver objects respectively. Instance handlers are used to uniquely identify each object in the system and are included in all the CSP events. The values *rp1* and *rp2* in Rules(3) and (4) represent the registration points where the object (*bIH*) is waiting to accept the signal instances *sig1* and *sig1*, *sig2*, or *sig3* respectively. Each *AcceptEventAction* in an fUML activity diagram (e.g., Figure 3) associated with a unique registration point.

Mapping from UML activity diagrams to CSP has been addressed several times in the literature [41,40]. The novel points of our mapping are as follows:

Rule(1) maps the fUML activity as a parent CSP process with several parameters (*param1*, *param2*, ..). Within this process we define sub-processes, each acts as a different fUML element within this activity. The *within* statement defines the action (sub-process) connected to the initial node (*AC1*). Rule(2) and (3) maps the *SendSignalAction* and *AcceptEventAction* to the CSP parameterized events send and accept respectively. The *registerSignals* event is used to let the object activation fill the *waiting event accepters* list with the allowed signals to be accepted at this point (registration point). The value *rp1* is explicitly included in the event so that each *AcceptEventAction* is uniquely identified. Without those registration points, the model checker will not be able to identify the possible signals to be accepted by the *accept* event. Section 5.2 describes how those events synchronize with the object's buffer process to allow the asynchronous communication between processes (active objects).

The fUML standard supports the fact that the *AcceptEventAction* handles more than one signal at a time. When the control flow of the activity reaches this action, the object waits for any of the defined signals (*sig1*, *sig2*, or *sig3*) to be received. If any of those signals arrive, the object execution proceeds and the incoming signal instance is passed to the *AcceptEventAction* output pin. For that reason, in Rule(4), we connect the decision node to the action's output pin to branch the flow based on the incoming signal. We use the same concept of Rule(3) followed by an external choice to represent the branching semantics. Rules like (2),(3), and (4) are not presented in [41,40] because the focus there is not on interaction between activity diagrams.

Rule(5) maps the combination of the actions: *valueSpecificationAction* and *addStructuralFeatureValueAction* to two events to allow (for example) the *aIH* instance handler's attribute *isOpen* to be set to FALSE. We represent the decision node as an internal choice (as in Rule(6)) when the incoming edge to the decision node is a control flow. But we represent it as an external choice (as in Rule(4)) when the incoming edge is an object flow. Having the decision nodes in the

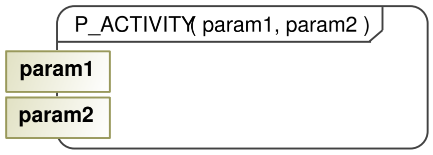
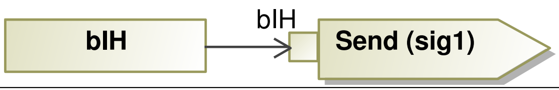
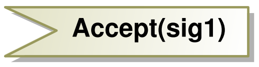
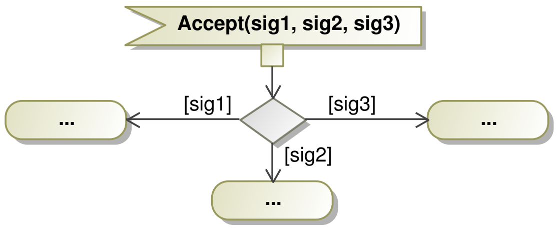
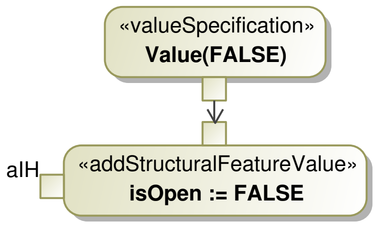
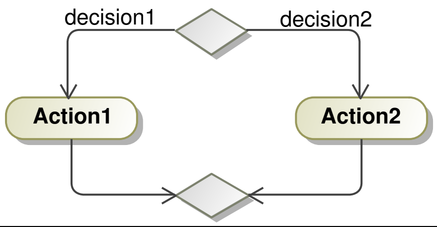
fUML Element	CSP Representation
Rule(1): Activity 	$P\_ACTIVITY(param1, param2) =$ *let* *Activity/Process Body* *within* $AC1$
Rule(2): Send Signal Action 	$AC1 = send!aIH!bIH!sig1 \rightarrow ...$
Rule(3): Accept Event Action 	$AC1 = registerSignals!bIH!rp1 \rightarrow$ $accept!bIH!sig1 \rightarrow ...$
Rule(4): Accept Event Action (*)  Accept(sig1, sig2, sig3) [sig1] [sig3] [sig2]	$AC1 = registerSignals!bIH!rp2 \rightarrow ($ $accept!bIH!sig1 \rightarrow ...$ $\Box$ $accept!bIH!sig2 \rightarrow ...$ $\Box$ $accept!bIH!sig3 \rightarrow ...)$
Rule(5): Add Structural Feature Value Action  «valueSpecification» Value(FALSE) aIH «addStructuralFeatureValue» isOpen := FALSE	$AC1 =$ $valueSpec!aIH?val : \{FALSE\} \rightarrow$ $addStFeatureValue!aIH!isOpen!val$ $\rightarrow ...$
Rule(6): Decision/Merge Nodes 	$DS1 = decision1 \rightarrow AC1$ $\sqcap$ $decision2 \rightarrow AC2$ $AC1 = ... \rightarrow MR1$ $AC2 = ... \rightarrow MR1$ $MR1 = ...$

Table 1: The fUML to CSP Mapping Rules

fUML standard allowed for modelling internal decisions which was not possible using xUML (Executable UML [19]).

The mapping rules scope
It is obvious that the mapping rules do not support all the fUML standard elements, and for the chosen elements not all the properties are considered in the formalization. This part discusses the rationale behind the inclusion and the exclusion for some elements.

The formalization rules include all the fUML elements that have been used in the Tokeneer fUML model (our primary case study) and the chosen properties for each element are sufficient to check deadlock freedom between the communicating active objects. This explains why we have excluded elements such as Activity Final Nodes from the formalization, especially that the dynamic objects creation and destruction is not support in this work. Also, formalizing unnecessary properties will lead to a complicated CSP model that FDR2 will possibly fail to check. For example, the formalization of the *addStructuralFeatureValueAction* considers the assignment of unordered boolean structural features only.

Some of the excluded fUML elements such as Fork and Join nodes are appropriate to use when modelling the concurrent behaviour within the active object. We will show in Section 5.2.2 that modelling the concurrent behaviours is considered in our formalization but only between the active objects which are communicating with each other asynchronously.

As we are constrained with CSP as a formal representation, some aspects in the fUML standard cannot be formalized directly using CSP such as the Join nodes which are used to combine multiple/parallel flows in the activity diagram into one flow. That is mainly because parallel processes in CSP can just synchronize on some events, but their behaviours cannot be combined to act as one process.

The fUML standard includes many intermediate actions such as: Read Structural Feature, Write Structural Feature and Test Identity actions. Our framework is flexible enough to support adding more formalization rules for such actions. However, some actions such as Create/Destroy Objects requires adding additional processes to the CSP model to handle the objects management tasks.

5.1.1 Formalization automation

We use Epsilon[2] as an MDE framework to do the transformation from the source model (fUML) to a CSP script. The transformation is done in two stages; firstly, Model-to-Model transformation from the fUML model to a CSP model using ETL (Epsilon Transformation Language), and secondly, Model-to-Text transformation from the CSP model to a CSP script using EGL (Epsilon Generation Language) [18]. The Model-to-Model transformation includes all the rules shown in Table 1 represented in ETL. Epsilon performs the transformation based on the source/target meta-models. In this work, we use the available UML2 meta-model [3] [35] and the CSP meta-model used in our previous work [33].

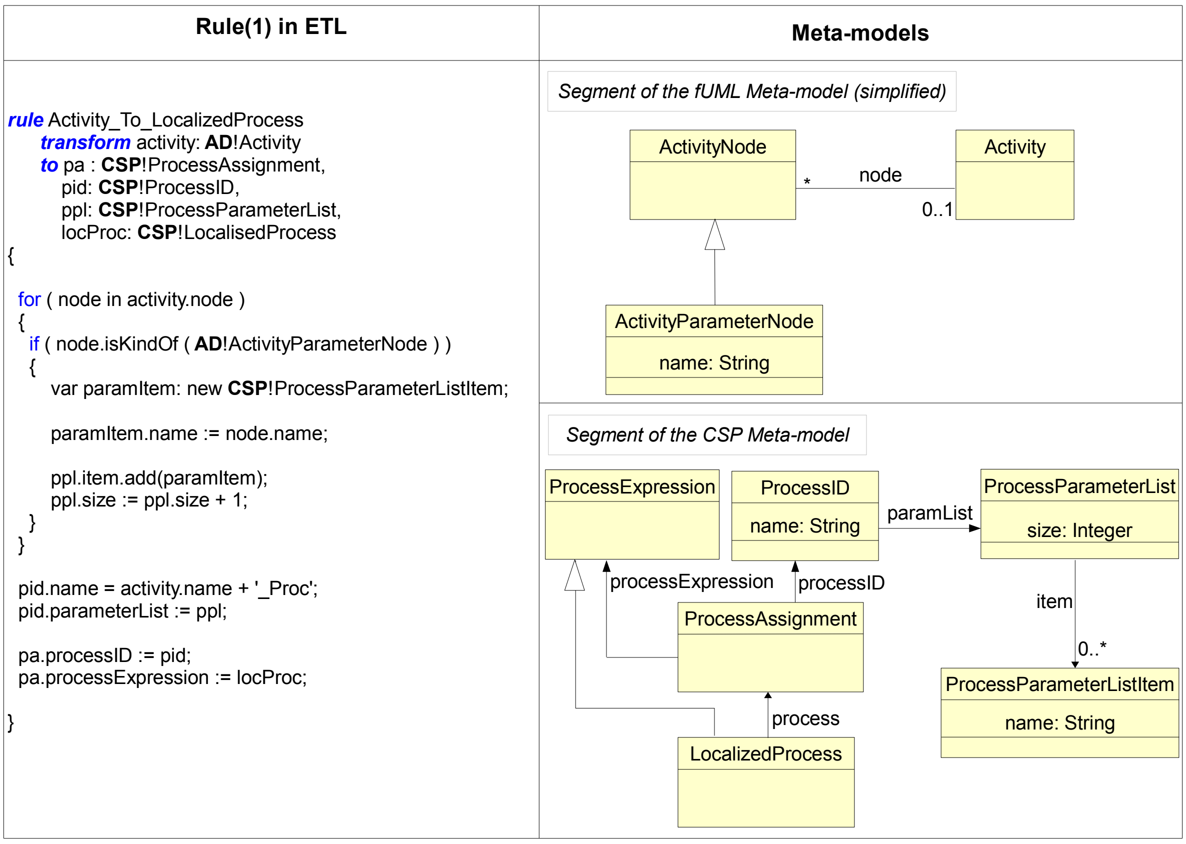
Figure 5 illustrates a sample ETL rule (Rule(1)) and segments of the involved meta-models in this rule. The first meta-model segment (fUML) shows that each *Activity* in the fUML model can have many *ActivityNodes*, and the *ActivityParameterNodes* are a kind of those nodes. This small segment is sufficient to understand Rule(1) ETL representation from the fUML aspect. Similarly, the second meta-model segment (CSP) shows that each *LocalizedProcess* holds mainly a *ProcessAssignment* entity which relates the *ProcessID* (e.g., AC1) with the *ProcessExpression* (the expression after the "=" operator).

The execution of this ETL rule (Activity_To_LocalizedProcess) applies the mapping shown in Rule(1) in Table 1, as it transforms any activity in the fUML source model (*activity*) to a CSP localized process (*locProc*) and all its related elements (*ProcessAssignment*, *ProcessID* and *ProcessParameterList*). The actions and the nodes inside the fUML activity are translated using the other mapping rules.

The fUML and CSP models elements can be accessed using the variables *AD* and *CSP* respectively using the '!' operator. The *for* loop and the nested *if* condition in the rule's body are concerned with the activity parameters nodes (*ActivityParameterNode*) that should be represented as *ProcessParameterListItem*'s in the CSP model. Inside the loop, the rule sets the items' names, adds them to the *ProcessParameterList* (*ppl*)

[2] Epsilon is a family of consistent and interoperable task-specific programming languages which can be used to interact with models to perform common MDE tasks such as code generation, model-to-model transformation, model validation, comparison, merging and refactoring [18]

[3] The unavailability of the fUML meta-model in a format that can be read by Epsilon forced us to use UML2 as a source meta-model. This workaround is technically valid because fUML is a subset of UML2.



```
rule Activity_To_LocalizedProcess
    transform activity: AD!Activity
    to pa : CSP!ProcessAssignment,
       pid: CSP!ProcessID,
       ppl: CSP!ProcessParameterList,
       locProc: CSP!LocalisedProcess
{

 for ( node in activity.node )
 {
  if ( node.isKindOf ( AD!ActivityParameterNode ) )
  {
     var paramItem: new CSP!ProcessParameterListItem;

     paramItem.name := node.name;

     ppl.item.add(paramItem);
     ppl.size := ppl.size + 1;
  }
 }

 pid.name = activity.name + '_Proc';
 pid.parameterList := ppl;

 pa.processID := pid;
 pa.processExpression := locProc;

}
```

Fig. 5: Rule(1) for Transforming State Machines to CSP Localized Processes

and adjusts the *ppl* size. After the loop, the rule sets the CSP *ProcessID* (*pid*) name with the activity name augmented with '_Proc' and then associates the CSP elements with each other. The reader can refer to [18] for more detail about the Epsilon ETL language.

The Model Formalizer uses Epsilon to execute all the ETL rules followed by the EGL script to perform the Model-to-Text transformation which generates a comprehensive CSP script that represents the source fUML model behavioral semantics.

5.1.2 Tokeneer fUML activity diagrams formalization

As mentioned in Section 3, our motivation is not to re-validate the Tokeneer project but to use it as a case study primarily to validate our framework and secondly to study the fUML model behaviour in the context of asynchronous communication as a possible implementation for Tokeneer Z specifications. This section shows a sample output from the Model Formalizer when using Tokeneer fUML as an input model. Figure 6 shows the Door Controller CSP process (*DoorControllerActivity_Proc*) that represents the behavioural semantics of the *DoorControllerActivity* depicted in Figure 3.

As a direct application of Rule(1), the *DoorControllerActivity* is translated to the *DoorControllerActivity_Proc* CSP localized process with the corresponding parameters. *AC2*, *AC8* and *AC10* are generated by Rule(5). Applying Rule(6) on the timer expiry decision node resulted in the internal decision in *DS1*.

When the process registers (using *registerSignals* event) and accepts (using *accept* event) the *unlockLatchSignal* in *AC1*, this means that the process is ready to accept this signal when it is placed in its object's (*selfObj*) *event pool*. On the other hand, when the *send* event in *AC4* happens, the *unLockingDoorCompleteSignal* will be placed in the User Panel object's (*upObj*) *event pool*. The mechanism that allows for signals sending/accepting is described in more detail in the following sections.

Representing the fUML activity as a localized process (using $let \cdots within$ statement) with a sub-process for each action makes the CSP process more readable and the transformation task easier. This style also allows for

$$
\begin{aligned}
&\mathit{DoorControllerActivity\_Proc}\\
&\quad(\mathit{alarmObj}, \mathit{doorObj}, \mathit{selfObj}, \mathit{upObj}, \mathit{userObj}) =\\
&\mathit{let}\\[1ex]
&AC1 = \mathit{registerSignals}!\mathit{selfObj}!\mathit{rp}1 \rightarrow\\
&\qquad \mathit{accept}!\mathit{selfObj}!\mathit{unlockLatchSignal} \rightarrow AC2\\[1ex]
&AC2 = \mathit{valueSpec}!\mathit{selfObj}?\mathit{val} : \mathit{FALSE} \rightarrow\\
&\qquad \mathit{addStFeatureValue}!\mathit{doorObj}!\mathit{isLocked}!\mathit{val} \rightarrow AC4\\[1ex]
&AC4 = \mathit{send}!\mathit{selfObj}!\mathit{upObj}!\mathit{unLockingDoorCompleteSignal}\\
&\qquad \rightarrow DS1\\[1ex]
&DS1 = (\mathit{lockTimeoutNotExceeded}!\mathit{selfObj} \rightarrow MR1\\
&\qquad \sqcap\\
&\qquad \mathit{lockTimeoutExceeded}!\mathit{selfObj} \rightarrow AC5)\\[1ex]
&AC5 = \mathit{send}!\mathit{selfObj}!\mathit{selfObj}!\mathit{lockLatchSignal} \rightarrow MR1\\[1ex]
&MR1 = AC6\\[1ex]
&AC6 = \mathit{registerSignals}!\mathit{selfObj}!\mathit{rp}2 \rightarrow AC7\\[1ex]
&AC7 = (\mathit{accept}!\mathit{selfObj}!\mathit{lockLatchSignal} \rightarrow ...\\
&\qquad \Box\\
&\qquad \mathit{accept}!\mathit{selfObj}!\mathit{doorIsOpenSignal} \rightarrow AC8)\\[1ex]
&AC8 = \mathit{valueSpec}!\mathit{selfObj}?\mathit{val} : \mathit{FALSE} \rightarrow\\
&\qquad \mathit{addStFeatureValue}!\mathit{alarmObj}!\mathit{isAlarming}!\mathit{val}\\
&\qquad \rightarrow AC10\\[1ex]
&AC10 = \mathit{valueSpec}!\mathit{selfObj}?\mathit{val} : \mathit{FALSE} \rightarrow\\
&\qquad \mathit{addStFeatureValue}!\mathit{doorObj}!\mathit{isLocked}!\mathit{val} \rightarrow AC12\\[1ex]
&AC12 = \mathit{send}!\mathit{selfObj}!\mathit{userObj}!\mathit{entryAuthorizedSignal}\\
&\qquad \rightarrow AC13\\[1ex]
&AC13 = \mathit{registerSignals}!\mathit{selfObj}!\mathit{rp}3 \rightarrow AC14\\[1ex]
&AC14 = (\mathit{accept}!\mathit{selfObj}!\mathit{unlockLatchSignal} \rightarrow AC13\\
&\qquad \Box\\
&\qquad \mathit{accept}!\mathit{selfObj}!\mathit{lockLatchSignal} \rightarrow ...\\
&\qquad \Box\\
&\qquad \mathit{accept}!\mathit{selfObj}!\mathit{doorIsClosedSignal} \rightarrow ...)\\[1ex]
&\mathit{within}\ AC1
\end{aligned}
$$

Fig. 6: The Corresponding CSP Process for the Door Controller Activity Segment (i.e., the translation of Figure 3)

recalling the same action several times without repetition.

5.2 Inter-object communication formalization

In the second stage, the Model Formalizer formalizes the inter-object communication semantics (described in Section 2.1) into CSP. However, having the *events dispatching scheduling* as one of the fUML standard semantic variation points led to different interpretations and thus different performances and results for the model checking using FDR2.

5.2.1 The initial attempts of the events dispatching formalization

We have conducted several attempts to formalize the *events dispatching scheduling* before reaching the current implementation. Although all of the attempts are compatible with the fUML standard, each of them implements the semantic variation point in a different way. The *events dispatching scheduling* is mainly controlled by the representation of the *event pool*. Among those attempts, we outline below two of them:

In the first attempt, we represented the *event pool* as a bag, which means that any signal can be dispatched from the it arbitrarily. The main problem in this representation was that it does not preserve the order of the incoming signals. Also when the bag becomes full, any incoming signal will be dismissed, which will lead to a quick deadlock invalid afterwards. Decreasing the effect of the former problem can be done by increasing the bag's size. However, with this representation, FDR2 failed to compile the CSP script when the bag's size was larger than 4 slots (for the Tokeneer case study), which in practice is too small to keep the system alive

In the second attempt we represented the *event pool* as a queue, which preserves the FIFO (First In First Out) order of the incoming signals. This is the default fUML strategy for dispatching events from the *event pool*. Using the queue solved the problem of the nondeterministic dispatching of signals from the *event pool*, preserving the incoming signals order. However, a new problem was introduced when an object receives an unexpected signal (i.e., not matched to one of the *waiting event accepters*). In this situation, the object dismisses the incoming signal directly because it has been already removed from the *event pool* for matching and the fUML standard does not allow for returning signals back to the *event pool*. In many cases the object may need to accept this dismissed signal after few further actions, which generally leads to an invalid deadlock to the system.

In the following sections we will describe the representation of the *event pool* as a Controlled Buffer in CSP which is the most optimized implementation (compared to the initial attempts) that led to the minimum compilation and checking time.

5.2.2 The event pool list as a Controlled Buffer

In the current implementation, the *event pool* is represented as a Controlled Buffer (described below). The Controlled Buffer with the current implementation benefits from its definition using only the CSP primitives (parallel composition, prefix, etc.) and avoiding using the Haskell functions which can be used to allow functional definitions within process definitions, as they lead to a significant decay in FDR2 performance during the compilation process. In other words, although Haskell functions are allowed by FDR2, they slow down the model checking and avoiding them by pure CSP makes the model checking faster. The current implementation also maintains the signals sending order and provides a scalable solution for the *event pool* size. The idea of this implementation came from Michael Goldsmith [9].

The Controlled Buffer consists of a sequence of nodes, where each node holds one signal at a time. When adding a new signal to the buffer, it is placed in the first empty node on a queue basis. Signals can be removed from any slot of the buffer (not on a queue basis). However, when selecting the signal to be removed, the buffer controller checks the oldest signal first (i.e., the signal that matches the selection criteria and at the same time spent the longest time in the buffer). All the signals located after the removed signal are shifted up when it is removed. When the buffer becomes full, the controller drops the oldest signal in the buffer and shifts all the other signals.

Figure 7 shows the general structure of a Controlled Buffer consisting of N consecutive nodes. When an object sends a signal to another object (performs the *send* event), the signal is placed in the receiver object's buffer (*event pool*) by placing it in the first node ($B0$), then the signal will move down the chain automatically until reaching the rightmost node in the buffer. The same will be repeated for any other incoming signal filling the buffer from right to left. When the buffer is full, the accepting of a new signal will result in the signal in the rightmost node (oldest signal) being dropped out (*drop* event) and all the signals shifted right by one node. Signals are moved down as a parameter to the *c1, c2, . . . , cN* events. According to the fUML standard, the dropped signals cannot be returned back to the *event pool*, and thus will never reach the destination.

As will be outlined below, the receiver object uses the $testY$ event (where Y represents the current node: $A, B, \ldots$) to check if the contained signal is member of the object's *waiting event accepters* list. If so, the signal is removed from the *event pool* via the $acceptY$ event,

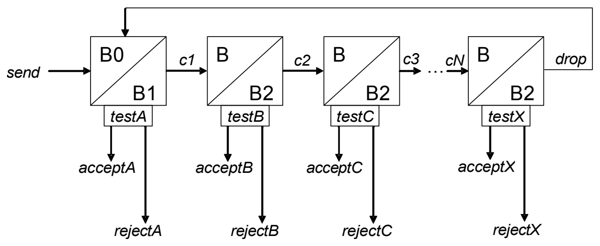


Fig. 7: The Event Pool as a Controlled Buffer

otherwise the $rejectY$ event is enabled to allow checking the next node. We represent each of those nodes as a mutually-recursive CSP process with a simple logic illustrated in Figure 8 for the first node ($B0$) and the general node (B). Notice the example possible values for the processes parameters between square brackets.

The processes *B0* and *B* represent the node when it is empty, while *B1* and *B2* represent them when the node is holding a signal. In *B0* and *B* the only allowed event is c to fill the node with the passed signal in its parameter (x). In *B1* and *B2* the hold signal (x) can either be passed to the next node (d) or tested (g) by the buffer controller for acceptance (e) or rejection (h). If in *B1* the c event happened, the oldest signal will be dropped (f) and then the d event will be allowed to shift the signals to the right.

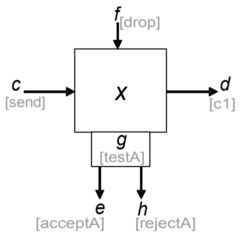
As the buffer consists of sequence of nodes, we combine the previous processes (*B0* and B's) in parallel to form a new process (*CB_NODES*) that represents the Controlled Buffer (*event pool*) but without being controlled yet. Figure 9 shows the CSP representation of a three node buffer which can hold three signal instances at a time. The process *CB_NODES* is defined using one *B0* process and two *B* processes whose parameters are instantiated appropriately. The functionality of *chase* will be described in Section 5.2.4.

$$\begin{array}{l} CB\_NODES = \\ \quad chase(((B0(send,\ c1,\ acceptA,\ drop,\ testA,\ rejectA) \\ \qquad\qquad \underset{\{|c1|\}}{\|}\ B(c1,\ c2,\ acceptB,\ testB,\ rejectB)) \\ \qquad\qquad \underset{\{|c2, drop|\}}{\|}\ B(c2,\ drop,\ acceptC,\ testC,\ rejectC) \\ \qquad)\setminus \{|\ c1,\ c2,\ drop\ |\}\ ) \end{array}$$

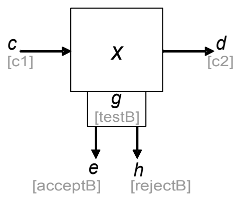
Fig. 9: Three Nodes Controlled Buffer

5.2.3 Controlling the buffer

To maintain dispatching signals in the same order they were sent, we developed a controller process



$$\begin{aligned}
B0(c,d,e,f,g,h) &= c?x \rightarrow B1(x,c,d,e,f,g,h)\\
B1(x,c,d,e,f,g,h) &=\\
&\quad d!x \rightarrow B0(c,d,e,f,g,h)\\
&\quad \square\ g!x \rightarrow (e!x \rightarrow B0(c,d,e,f,g,h)\\
&\qquad\qquad \square\ h \rightarrow B1(x,c,d,e,f,g,h))\\
&\quad \square\ c?y \rightarrow f?z \rightarrow d!x \rightarrow B1(y,c,d,e,f,g,h)
\end{aligned}$$



$$\begin{aligned}
B(c,d,e,g,h) &= c?x \rightarrow B2(x,c,d,e,g,h)\\
B2(x,c,d,e,g,h) &= d!x \rightarrow B(c,d,e,g,h)\\
&\quad \square\ g!x \rightarrow (e!x \rightarrow B(c,d,e,g,h)\\
&\qquad\qquad \square\ h \rightarrow B2(x,c,d,e,g,h))
\end{aligned}$$

Fig. 8: Buffer's First and General Nodes

(CB_CTRL) that checks nodes one by one from the oldest (rightmost) to the newest (leftmost) before removing the signal from the *event pool*, and if the signal exists in the *waiting event accepters* list, the process allows for its acceptance (*accept* event) otherwise the signal is rejected (*reject* event) and the next node is checked. Figure 10 shows our representation of the buffer controller process (CB_CTRL) for a three nodes *event pool*.

$$\begin{aligned}
&CB\_CTRL(\{\}) = registerSignals!aIH?rp \rightarrow\\
&\qquad\qquad CB\_CTRL(getRegisteredSignals(rp))\\[1ex]
&CB\_CTRL(EA) = testC?x \rightarrow if(member(x,EA))\ then\\
&\qquad\qquad (acceptC!x \rightarrow CB\_CTRL(\{\}))\\
&\qquad\qquad else\ rejectC \rightarrow\\
&\qquad testB?x \rightarrow if(member(x,EA))\ then\\
&\qquad\qquad (acceptB!x \rightarrow CB\_CTRL(\{\}))\\
&\qquad\qquad else\ rejectB \rightarrow\\
&\qquad testA?x \rightarrow if(member(x,EA))\ then\\
&\qquad\qquad (acceptA!x \rightarrow CB\_CTRL(\{\}))\\
&\qquad\qquad else\ rejectA \rightarrow\\
&\qquad\qquad send?anySig \rightarrow CB\_CTRL(EA)
\end{aligned}$$

Fig. 10: The Buffer Controller Process for a Three Nodes Event Pool

The *getRegisteredSignals* is a mapping function that returns the allowed signal(s) at a certain registration point (rp). For example, in the Door Controller activity, *getRegisteredSignals(rp2)* returns *lockLatchSignal* and *doorIsOpenSignal*. The *registerSignals* event synchronizes with the corresponding event in the translation of the activity diagram (Rule(3) and (4)) to initiate the signals checking process. The controller process (CB_CTRL) checks ($testY$) the nodes starting from the rightmost node (C) to the leftmost node (A). If the signal is a member of the *waiting event accepters* list (EA), the controller allows for its acceptance ($acceptY$) and flushes all the other signals in EA, otherwise it is rejected ($rejectY$) and the next node is checked. The controller at the end synchronizes with the *send* event to stop the checking until an object sends any signal to *aIH*.

To allow the CB_CTRL process to control the buffer CB_NODES we combine them in parallel in the new process CB_NODES_CTRL as illustrated in Figure 11. The set *aSynchEvents* contains the synchronization events: *test*, *reject*, and *accept* for all nodes.

$$CB\_NODES\_CTRL = CB\_NODES \underset{aSynchEvents}{\|} CB\_CTRL(\{\})$$

Fig. 11: Controlled Nodes

5.2.4 Moving signals along the Controlled Buffer

The parallel combination in the process CB_NODES_CTRL does not provide a mechanism to force FDR2 to move the signals along the nodes from left to right. For that reason we depend on the *chase* function of FDR2 to complete the definition.

Chase gives priority to internal (tau) transitions over external ones, and chooses one internal transition arbitrarily when there is a choice of several. This reduces the state space of the labelled transition system in FDR2 by removing external transitions competing with internal ones, and selecting one internal transition where there is a choice of them. This results in a refinement of the original process, which can only perform external events once all internal progress have completed.

Thus *chase* is not semantics-preserving in general (and in this case), but it is exactly what is required here so that shuffling the signals along always occurs after an output event before further visible events are possible. For more details about how *chase* works the reader can refer to [42]. For example, using the *chase* function for analyzing the left hand side tree (a tree with some hidden events) in Figure 12 will produce only two possible traces $\langle tau, tau, g\rangle$ or $\langle tau, tau, h\rangle$.

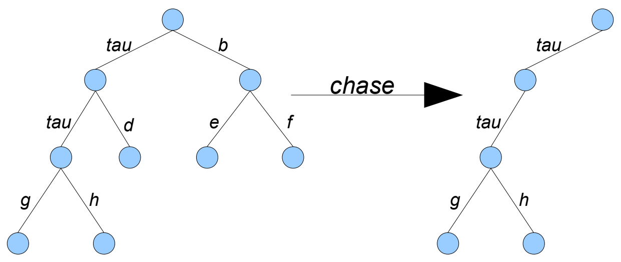


Fig. 12: Application of *Chase* function

Figure 9 and Figure 13 illustrate the application of *chase* to the processes *CB_NODES* and *CB_NODES_CTRL* respectively after hiding the buffer internal events (*test*, *reject*, *c*, and *drop*) for all nodes (grouped in *aHiddenEvents* for the *CB_NODES_CTRL* process). Having those events hidden (*taus*), FDR2 will follow them causing signals to be propagated along the nodes whenever a *send* event happens. The process *CB* is the complete definition of the Controlled Buffer for one instance in the system.

$CB = chase(CB\_NODES\_CTRL \setminus aHiddenEvents)$

Fig. 13: The Complete Definition of the Controlled Buffer

It is important to note that we are not using the *chase* function in the conventional way. *Chase* here *prevents* further external events from occurring following output from the buffer until the signals are propagated internally along the buffer. This is precisely the behaviour required: that the effect of an output is instantaneous. Representing the the buffer as a parallel combination of small processes ($B0 \parallel B \parallel B \parallel \cdots$) rather than a sequence of signals ($CB(\langle sig1, sig2, \cdots, sigN\rangle)$) shows a substantial performance improvement during the model checking compared to the later representation. The non-deterministic design of the *CB* process allowed *chase* to move the signals, because *chase* has no effect on the deterministic processes ($chase(P) = P,\ when\ P\ is\ deterministic$).

5.3 The *SYSTEM* process

This is the third stage of formalizing the fUML model where the Model Formalizer generates the overall system process (*SYSTEM*). This process is a parallel combination between all processes that synchronize on the *send*, *accept*, and *registerSignals* events as depicted in Figure 14. This in turn, for example, will allow object *A* to send signals to object *B* by inserting the signals in its (object B) *event pool* (Controlled Buffer).

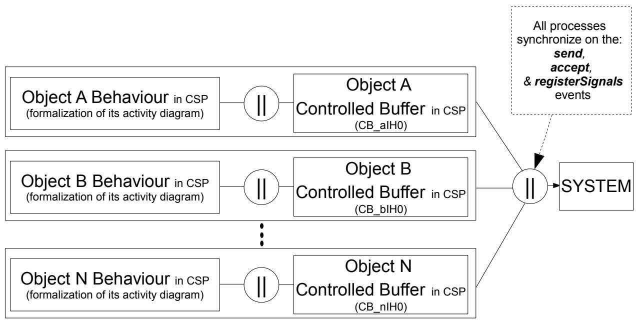

All processes synchronize on the: **send**, **accept**, & **registerSignals** events
Object A Behaviour in CSP (formalization of its activity diagram)
||
Object A Controlled Buffer in CSP (CB_aIH0)
Object B Behaviour in CSP (formalization of its activity diagram)
||
Object B Controlled Buffer in CSP (CB_bIH0)
Object N Behaviour in CSP (formalization of its activity diagram)
||
Object N Controlled Buffer in CSP (CB_nIH0)
||
SYSTEM

Fig. 14: The *SYSTEM*

The *SYSTEM* process can then be used by FDR2 to check the whole system against a specific behaviour such as: deadlock, livelock or determinism.

5.4 The automatic generation of the inter-object communication CSP processes

The Model Formalizer component generates all the processes related to the inter-object communication automatically using Epsilon. A copy of those processes will be generated using an EGL script for each instance in the input fUML model to allow its objects asynchronous communication. For example, the Model Formalizer generates the following processes for the Door Controller instance *dcIH0*: *CB_NODES_dcIH0*, *CB_CTRL_dcIH0*, *CB_NODES_CTRL_dIH0* and *CB_dIH0*.

The Model Formalizer also generates the required sets of alphabets which will be used in those CSP processes such as: *aSynchEvents* and *aHiddenEvents*. Finally, the Model Formalizer generates the SYSTEM process described in the previous section.

6 Deadlock checking using FDR2

After the Model Formalizer completes its function and generates the comprehensive CSP script, the framework initiates FDR2 to perform the model checking. In this paper we will focus on the deadlock checking as one of the possible behaviours that FDR2 can check automatically. FDR2 reports a deadlock when it reaches a state in which no further actions are possible, which means in our model that all the objects in the system (*SYSTEM* process) are waiting to accept signals from each other. In case of deadlock, FDR2 displays a counter-example (sequence of events) that led to this deadlock.

Tokeneer deadlock checking

The Tokeneer CSP model *SYSTEM* process includes four interacting processes (Door, Door Controller, User, and User Panel). Each process has its own *event pool* with 10 slots. When checking *SYSTEM* using FDR2, it managed to compile the CSP script (about 600 lines) and reported a deadlock scenario (counter-example) after exploring 2.5K states in five seconds [4]. The following trace shows part this counter-example:

```
 <...
send.u0.up0.readUserTokenSignal,
accept.up0.readUserTokenSignal,
send.up0.dc0.unlockLatchSignal,
accept.dc0.unlockLatchSignal,
send.dc0.up0.unLockingDoorCompleteSignal,
lockTimeoutExceeded.dc0,
accept.up0.unLockingDoorCompleteSignal,
send.dc0.dc0.lockLatchSignal,
registerSignals.dc0.rp9,
accept.dc0.lockLatchSignal,
valueSpec.dc0.FALSE,
send.up0.u0.doorUnlockedSignal,
addStFeatureValue.dc0.isAlarming.FALSE,
registerSignals.d0.rp20,
registerSignals.u0.rp3,
accept.u0.doorUnlockedSignal,
send.u0.d0.openDoorSignal,
accept.d0.openDoorSignal,
send.up0.up0.resetSignal,
valueSpec.d0.TRUE,
registerSignals.u0.rp4,
addStFeatureValue.d0.isOpen.TRUE,
registerSignals.up0.rp15,
valueSpec.dc0.TRUE,
accept.up0.resetSignal,
registerSignals.up0.rp10,
addStFeatureValue.dc0.isLocked.TRUE,
send.d0.dc0.doorIsOpenSignal,
registerSignals.dc0.rp6,
registerSignals.d0.rp19 >
```

[4] The checking has been done on an Intel Core 2 Duo machine with 2 GB memory

The trace shows the sequence of events generated from the checking of Tokeneer *SYSTEM* process. The *registerSignals* event causes the object to wait until one of the registered signals arrives. As highlighted in the trace, eventually all the system's objects are waiting for each other, causing deadlock. The Door Controller (*dc0*) will never send the *entryAuthorized* signal to the User (*u0*) because it does not make sense for a User to enter when the door is locked. Consequently, the User cannot evolve its behaviour. Also the *unlockLatchSignal* will never be sent from the User Panel (*up0*) to the Door Controller and so the Door Controller cannot evolve its behaviour. This scenario might happen in real life if the user takes a long time (more than the timer period (*lockTimeoutExceeded*)) to open the door after getting permission to enter from the User Panel.

We cannot claim that this deadlock is a breach of the Tokeneer requirements [7] for two reasons: firstly, the entry expiration timer that caused this deadlock was not specified explicitly in the requirements document. However, we added this timer as part of the system implementation to prevent the Door Controller from waiting forever for a User to enter the enclave. Secondly, the requirements do not specify a certain communication mechanism between the system components (objects), leaving that as an implementation issue. We would argue that this deadlock was identified because we modelled concurrent behaviour of all the components within the TIS subsystem.

When we disabled the entry expiration timer (i.e., the door can be kept closed and unlocked forever), the system did not deadlock and FDR2 succeeded in doing a full model check in eight seconds after exploring 9.2K states on the same hardware mentioned in Section 6. The Controlled Buffer with the pre-described implementation in Section 5.2.2 allowed for this fast compilation and model checking compared to the previous implementations of the inter-object communication mechanism.

We also tried to reproduce this deadlock scenario on the Tokeneer simulator [2]; however, this scenario did not happen due to the different implementation decisions that were taken in the SPARK implementation especially for the door unlocking timer.

7 Formalization and model checking feedback

There are two kinds of feedback that can be provided by the framework to the modeller. The first kind is the Formalization Report which is generated by the Model

Formalizer in case of errors during the formalization process. The second kind, is a UML sequence diagram which visualizes the counter-example in case of deadlock.

7.1 The Formalization Report

The formalization rules described in Section 5 include only a subset of the fUML elements, this means that not every fUML diagram can be formalized using the Model Formalizer. The diagrams have to fulfill minimum requirements in order to be formalized. These requirements include the existence of certain elements and the assignment of certain properties. For example, the Model Formalizer cannot formalize an fUML activity diagram that does not include a connected initial node, because this will prevent the Model Formalizer from setting the initial CSP sub-process in the *within* clause of the localized process. Another example is not assigning the name of an edge emerging from a decision node in an fUML activity diagram.

To be able to check the formalizability of each diagram ("is formalizable?"), each transformation rule is divided into two parts. The first part checks for the required elements/assignments, and if met, the second part performs the transformation. Otherwise, a formalization error is reported to the modeller that guides him to the missing items.

7.2 The UML Sequence Diagram Generator

We have shown in Section 6 the output that FDR2 produces in case of deadlock (a counter-example as a sequence of events). This representation may not be accessible to the modeller who developed his model as an fUML model in the beginning. For that reason, we include the UML Sequence Diagram Generator component as part of our framework to transform FDR2 output to a modeller friendly format. This component takes the counter-example generated by FDR2 as an input and generates a UML sequence diagram that represents this counter-example.

The UML Sequence Diagram Generator also makes use of the Object-to-Class mapping table (generated by the Model Formalizer) to relate the behaviour of each object to its class in the fUML model. Figure 15 shows the automatically generated sequence diagram which corresponds to the trace in Section 6.

Table 2: FDR2 Output and the Corresponding *sdx* representation

Events in FDR2 Output	**SDX Representation**
send.dc0.dc0.lockLatchSignal	dc0:dc0.lockLatchSignal
registerSignals.d0.rp19	*1 u0 Expecting: -closeDoorSignal *1

The UML Sequence Diagram Generator depends on an open-source tool called Quick Sequence Diagram Editor [28]. The tool takes an input script (*.sdx file) that specifies the system objects and how they interact with each others. Based on that script, the tool generates an image of that sequence diagram. A subcomponent of the UML Sequence Diagram Generator translates FDR2 output to an *sdx* script based on a group of simple mapping rules. Table 2 shows two samples of FDR2 output and the corresponding *sdx* representation.

To list the corresponding signals of *rp19*, we use the information stored in a mapping table called RP-to-Signals which had been generated by the Model Formalizer during the formalization process. This table maps between each *rp* and the possible accepted signal(s) at this point.

7.2.1 Multiple counter-examples

FDR2 has the option to generate more than one counter-example in case of deadlock. Instead of aborting the model checking once detecting a sequence of events that lead to a deadlock, FDR2 continues the model checking until reaching another sequence. Our framework utilizes this option in FDR2 by allowing the modeller to identify the maximum number of counter-examples to be generated in case of deadlock through a simple GUI (Graphical User Interface) before the model checking as shown in Figure 16. This is made possible by FDR2 batch mode that gave us this level of control through the command line parameters.

The UML Sequence Diagram Generator has the ability to detect if more than one counter-example have been generated by FDR2, and thus generates a corresponding sequence diagram for each counter-example.

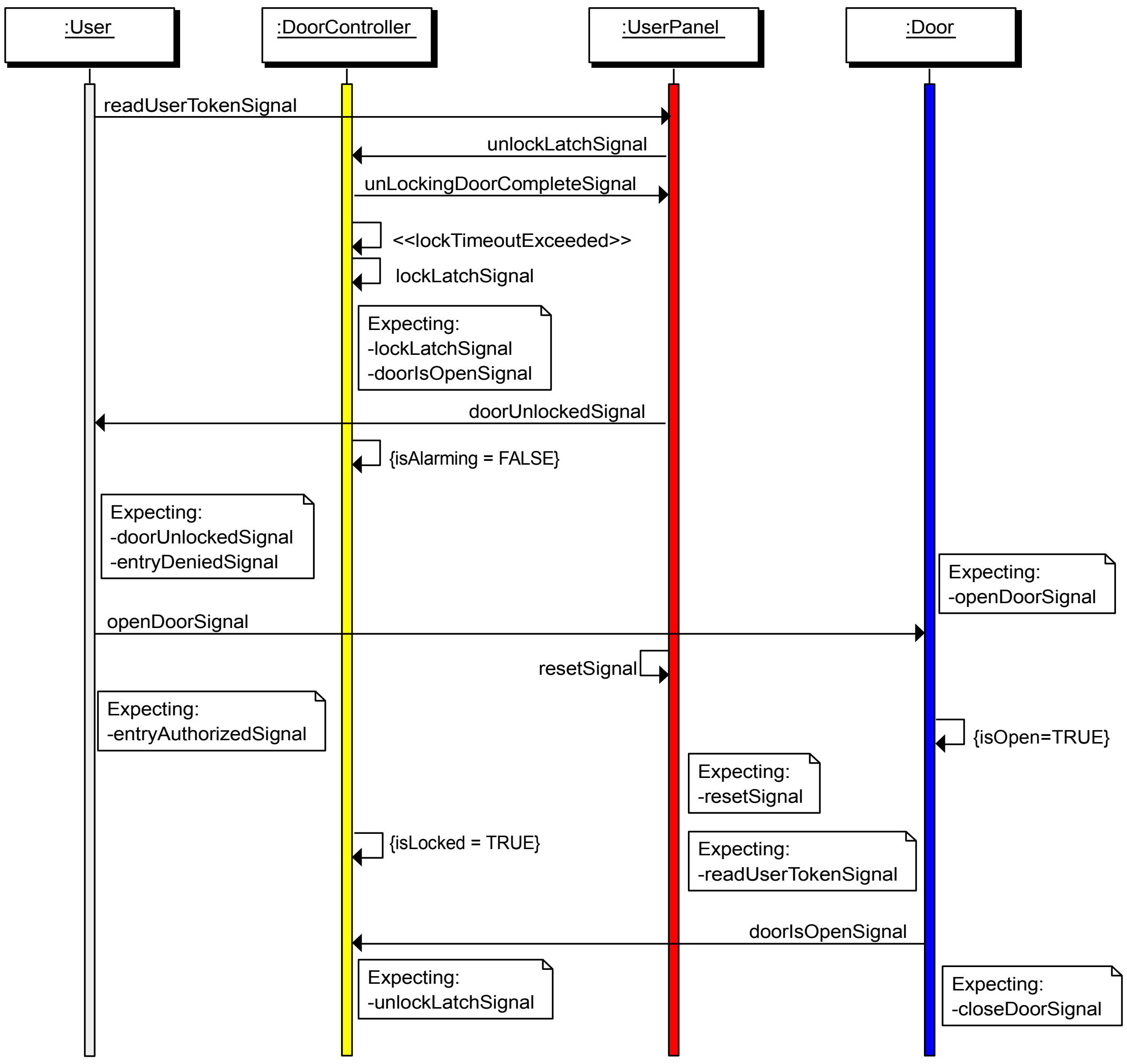


Fig. 15: The Generated UML Sequence Diagram from the FDR2 Counter-example

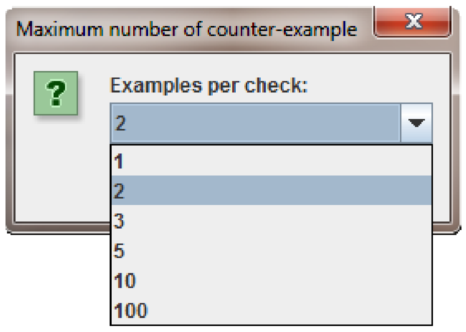


Fig. 16: The Modeller Selects the Counter-examples per Check

7.2.2 Loop detection

Sometimes the generated counter-example includes a repetition of certain pattern(s) (sub-sequence of events) many times, which decreases the readability of the corresponding sequence diagram as it becomes too long to track. To avoid this issue, the UML Sequence Diagram Generator has the ability to detect this repetition automatically using an advanced search algorithm and replace it with one pattern surrounded by a "loop" box.

Figure 17 shows part of a generated sequence diagram. As shown inside the "loop" box, the repetition of sending the signals requestEntry and readUserToken three times, has been detected by the UML Sequence Diagram Generator. Such a scenario can happen due to a bug in the User activity diagram.

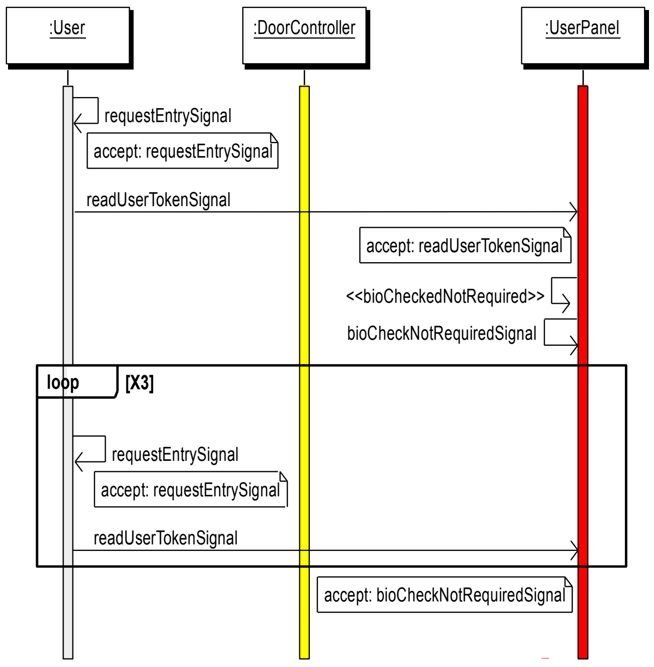

:User
:DoorController
:UserPanel
requestEntrySignal
accept: requestEntrySignal
readUserTokenSignal
accept: readUserTokenSignal
<<bioCheckedNotRequired>>
bioCheckNotRequiredSignal
loop
[X3]
requestEntrySignal
accept: requestEntrySignal
readUserTokenSignal
accept: bioCheckNotRequiredSignal

Fig. 17: Detecting Loops in the Counter-example

8 Framework implementation

We have implemented the framework within MagicDraw as a plugin called "Compass" (Checking Original Models means Perfectly Analyzed Systems). To use Compass, the modeller should first model the system objects' behaviours using fUML activity diagrams. Consequently, he can use the plugin GUI to initiate the deadlock checking. In case of deadlock the plugin generates an UML sequence diagram to the modeller in a separate window. Compass totally isolates the modeller from dealing with the formal representation of the model.

Figure 18 shows a screen shot of MagicDraw/Compass during checking Tokeneer fUML model for deadlock. The screen shows part of the TIS subsystem fUML activity diagrams and the sequence diagram which shows the deadlock scenario.

We would argue that implementing the framework in the form of a plugin to an already existing case tool is more practical than implementing it as a standalone application for several reasons. Compared to a standalone formalization application, a plugin will allow for having a single integrated modelling environment. Also modifying the plugin to work with other case tools is a straightforward task, which means that the plugin can be made available for several case tools. This in turn will allow the modellers who are already using a certain case tool not to change their modelling environment to check the models (or even to re-check legacy models).

9 Related work

Much research work has been done on formalizing semi-formal models to check different properties. Among this work [13] and [39] focused on checking user defined safety specification for an xUML models formalized into mCRL2 [11] and S/R (the input language of COSPAN [14]) respectively. Roscoe *et al.* [29] developed a CSP-M based compiler to formalize Statemate Statecharts [8] into CSP for the purpose of checking several properties such as consistency with application-specific requirements.

Our work is more related to those who focused on checking model-independent system behaviours (i.e., can be checked as part of the toolset) such as deadlock or livelock. In this category, Yong Ng *et al.* [23] used CSP as a formal representation to check deadlock and divergence for the input UML state machines. Thierry-Mieg *et al.* [32] used IPN (Instantiable Petri Nets [22]) to check deadlock and unreachable final states for the input UML activity diagrams. Also Turner *et al.* [34] automatically formalized xUML state machines into $CSP \parallel B$ [30] (an integrated formal language that combines CSP and B) to check deadlock.

Formally representing the asynchronous communication between objects has been discussed in a limited way in [13, 10, 34] where part of the xUML was formalized, which specify a way of communication different from fUML. On the other hand, [39] simulated the asynchronous message passing by synchronous communication between processes modelling objects and their message queues. Our previous work [1] considered also the asynchronous communication mechanism between system objects; however the manual formalization reduced the practicality of the approach.

To perform the formalization automatically, some authors developed their own tools to perform that task. For example, Cabot *et al.* in [6] developed a tool called UMLtoCSP to do the formalization. Also Shah *et al.* in [31] used UMLtoAlloy and Alloy Analyzer to do the formalization and model checking respectively. Another group of authors used MDE tools to do the transformation. Varró *et al.* in [36] summarized a comparison between eleven different MDE tools used to transform from UML activity diagrams into CSP (UML-to-CSP case study [5]), as part of the AGTIVE'07 tool contest. Also Treharne *et al.* in [33] used the Epsilon framework to transform UML state diagrams to CSP||B.

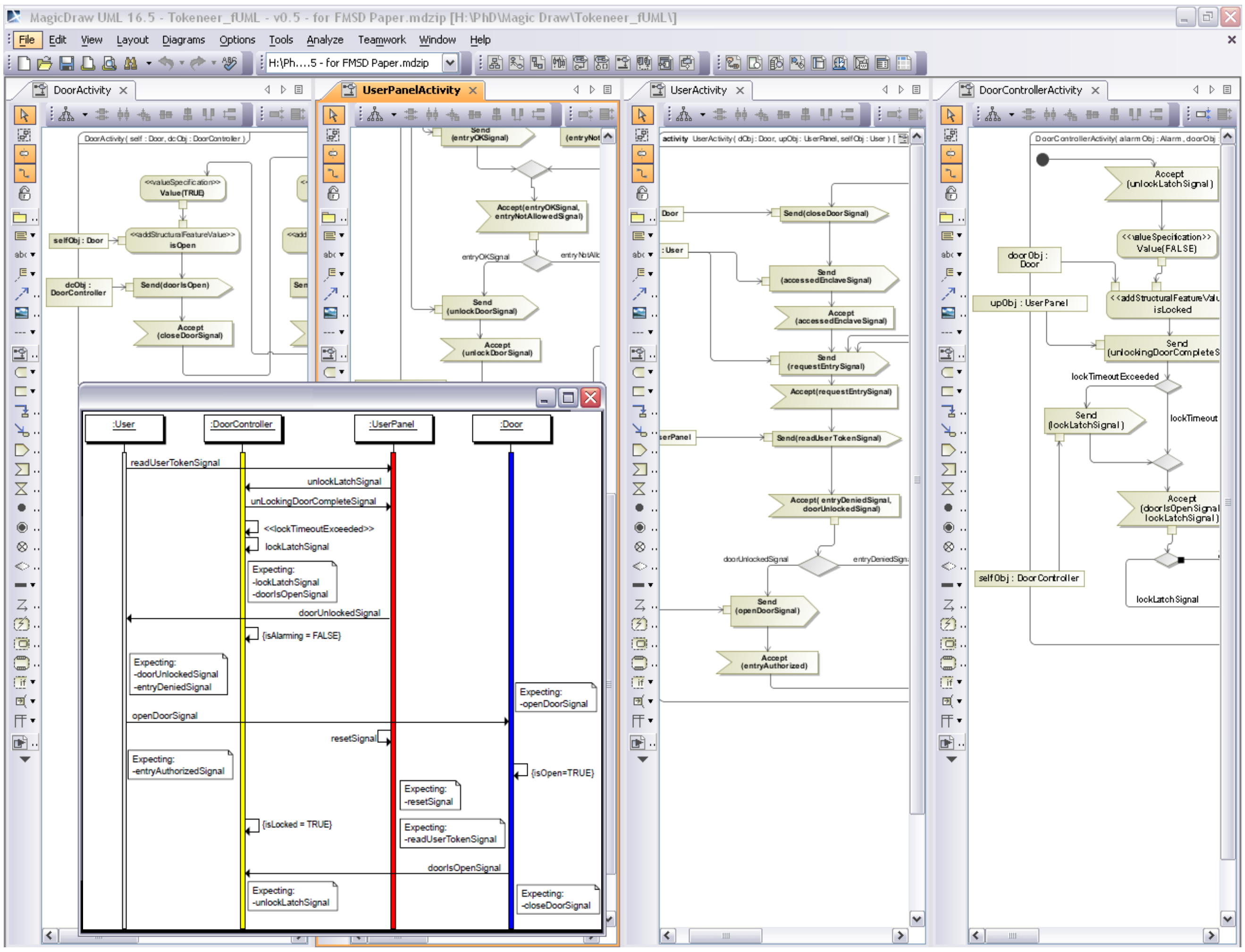

MagicDraw UML 16.5 - Tokeneer_fUML - v0.5 - for FMSD Paper.mdzip [H:\PhD\Magic Draw\Tokeneer_fUML\]
File Edit View Layout Diagrams Options Tools Analyze Teamwork Window Help
DoorActivity
UserPanelActivity
UserActivity
DoorControllerActivity
:User
:DoorController
:UserPanel
:Door
readUserTokenSignal
unlockLatchSignal
unLockingDoorCompleteSignal
<<lockTimeoutExceeded>>
lockLatchSignal
Expecting: -lockLatchSignal -doorIsOpenSignal
doorUnlockedSignal
{isAlarming = FALSE}
Expecting: -doorUnlockedSignal -entryDeniedSignal
openDoorSignal
Expecting: -openDoorSignal
resetSignal
{isOpen=TRUE}
Expecting: -entryAuthorizedSignal
Expecting: -resetSignal
{isLocked = TRUE}
Expecting: -readUserTokenSignal
doorIsOpenSignal
Expecting: -unlockLatchSignal
Expecting: -closeDoorSignal

Fig. 18: Screen shot of MagicDraw Running Compass

Providing modeller friendly feedback to report the model checking results has been addressed only a few times in the literature. The authors in [6,31] proposed presenting the model checking results (e.g., counter-example) as an object diagram that represents a snapshot of the system during the error. Alternatively, Mrugalla *et al.* in [21] presents the counter-example as sequence and timing diagrams. In another approach, the authors in [32,27] proposed compiler-style errors with valuable feedback.

Compared to all the reviewed literature, this work is the first attempt to automatically formalize the fUML activity diagrams, including the formalization of the fUML asynchronous inter-object communication mechanism.

10 Conclusion and future work

In this paper we have presented a framework that helps modellers to check the behaviour of their fUML model automatically. The framework depends on formalizing the fUML model into CSP and then checks it using FDR2 taking into consideration the formalization of the asynchronous inter-object communication mechanism. The comprehensive formalization (for fUML diagrams and communication mechanism) allowed for checking the system against deadlock which may occur if all the system's objects stop working waiting for each other.

In case of deadlock, the framework provides the user with a UML sequence diagram that describes that deadlock scenario in terms of the fUML model, not the formal CSP model, to isolate the modeller from the formal domain.

We have developed an implementation of this framework as a MagicDraw plugin called Compass. Compass made use of the Epsilon MDE framework to translate the fUML model into a CSP script in two stages (Model-to-Model then Model-to-Text).

Validating the framework's functionality and applicability was achieved by applying it on a non-trivial case study (Tokeneer ID Station). Using the implementation of the communication mechanism described in Section 5.2, FDR2 succeeded in compiling the generated CSP script and detected the deadlock scenario in five seconds for a 10 slots *event pool* for each object. The detected deadlock scenario was due to an implementation decision added to Tokeneer fUML model (i.e., not a breach in the Tokeneer specification).

Currently, the framework supports having only one instance for each class. Such a constraint will be addressed in our future work to support multiple instances for each class in the system. Also we will modify the framework to include safety and security specifications checking.

Acknowledgements Thanks to Michael Goldsmith, Philip Armstrong and Bill Roscoe for discussion about implementing the Controlled Buffer in CSP. Thanks also to James Sharp for his help in developing the buffer. Finally, we want to thank Jim Woodcock for his discussion about the Tokeneer project.

Appendix A: The ETL transformation rules

This appendix includes a simplified version (just showing the main logic) of the used ETL rules in the framework and the associated meta-models for each rule. We have developed a group of Epsilon operations to allow for more compact ETL rules. The following outline those operations:

A.1 Operations

– ***getCSP_Process***
  Takes an activity diagram element reference as an input and returns the corresponding CSP ProcessID for that element.

– ***createSendEvent***
  Creates a CSP Event entity (*send*) for the SendSignal action and returns it. It also creates the corresponding CSP EventParameter's and associates them with the Event entity.

– ***createRegisterSignalsEvent***
  Creates a CSP Event entity (*registerSignals*) for the AcceptEvent action and returns it. It also creates the corresponding CSP EventParameter's and associates them with the Event entity.

– ***createAcceptEvent***
  Creates a CSP Event entity (*accept*) for the AcceptEvent action and returns it. It also creates the corresponding CSP EventParameter's and associates them with the Event entity.

– ***createValueSpecificationEvent***
  Creates a CSP Event entity (*valueSpec*) for the ValueSpecification action and returns it. It also creates the corresponding CSP EventParameter's and associates them with the Event entity.

– ***createAddStructuralFeatureValueEvent***
  Creates a CSP Event entity (*addStFeatureValue*) for the AddStructuralFeatureValue action and returns it. It also creates the corresponding CSP EventParameter's and associates them with the Event entity.

– ***createInternalChoiceEvent***
  Creates a CSP Event entity for a given internal choice branch and returns it.

– ***addToLocalizedProcess***
  Adds the created subprocess (ProcessAssignment) to the corresponding localized process.

– ***getTargetNode***
  Returns the target node (connected to the other side of the edge) given the edge reference.

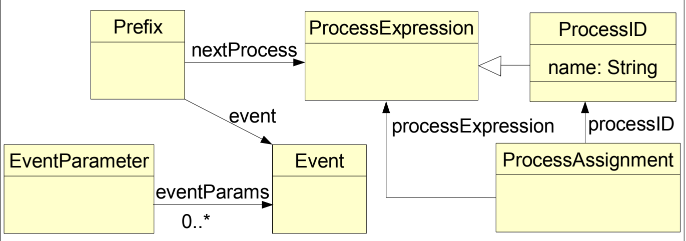
A.2 Rules

Rule(2) in ETL

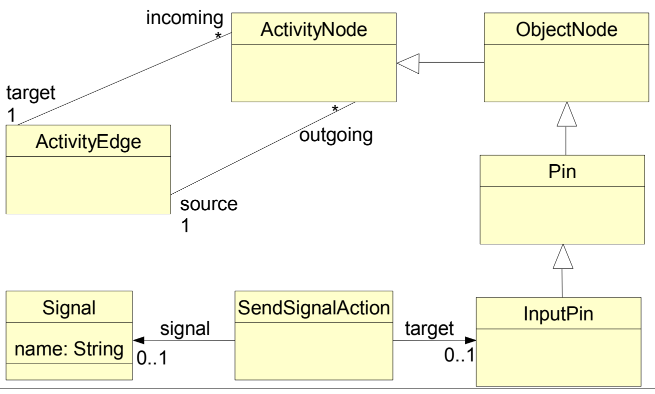
```
rule SendSignalAction_To_SubProcess
     transform action: AD!SendSignalAction
     to pa: CSP!ProcessAssignment,
        pid: CSP!ProcessID,
        prefix: CSP!Prefix
{

 pa.processID := getCSP_Process( action );

 var targetObj : String :=
                    action.target.incoming.source.name;

 var signalName: String := action.signal.name;

prefix.event := createSendEvent( targetObj,
                                        signalName);

prefix.nextProcess :=  getCSP_Process( getTargetNode
                                   (action.outgoing ) );
 pa.processExpression := prefix;

 addToLocalizedProcesst (action, pa);

}
```

Meta-models

Segment of the fUML Meta-model (simplified)



Segment of the CSP Meta-model (simplified)

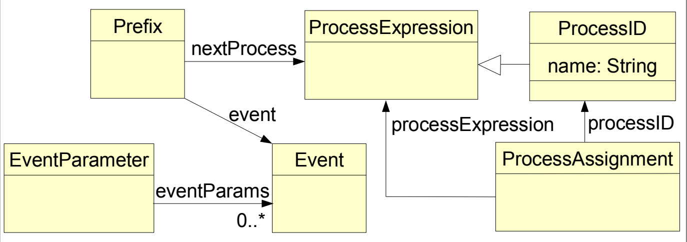


Rule(3 & 4) in ETL

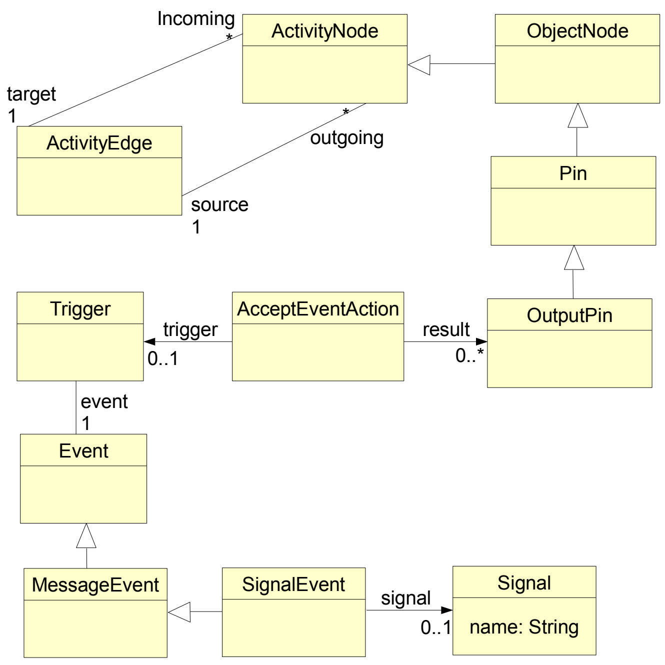
```
rule AcceptEventAction_To_SubProcess
     transform action: AD!AcceptEventAction
     to pa : CSP!ProcessAssignment,
        prefix: CSP!Prefix
{

 pa.processID := getCSP_Process (action);

 prefix.event := createRegisterSignalsEvent();

 if (action.trigger.size() = 1)  - - Rule(3)
 {

   var next_prefix: new CSP!Prefix;

   next_prefix.event := createAcceptEvent (
                              action.trigger.event.signal.name );

   next_prefix.nextProcess := getCSP_Process(
                              getTargetNode( action.outgoing) );

   prefix.nextProcess := next_prefix;

 }
 else - - First part of Rule(4)
 {
    prefix.nextProcess := getCSP_Process (
                     getTargetNode( action.result.outgoing ) );

 }

 pa.processExpression := prefix;

 addToLocalizedProcess(action, pa);

}
```

Meta-models

Segment of the fUML Meta-model (simplified)



Segment of the CSP Meta-model (simplified)

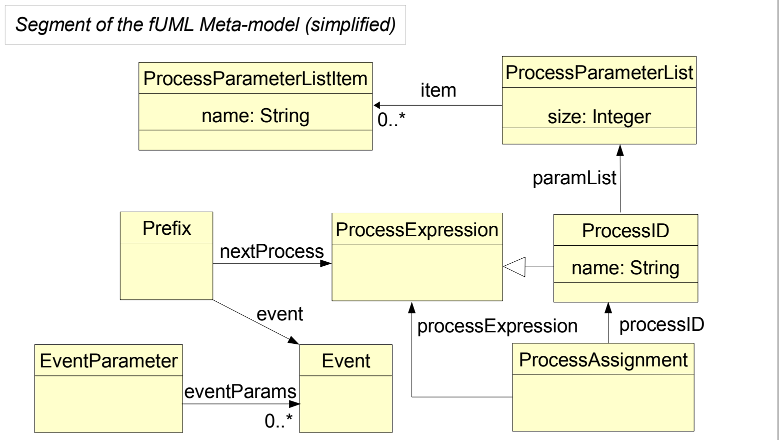


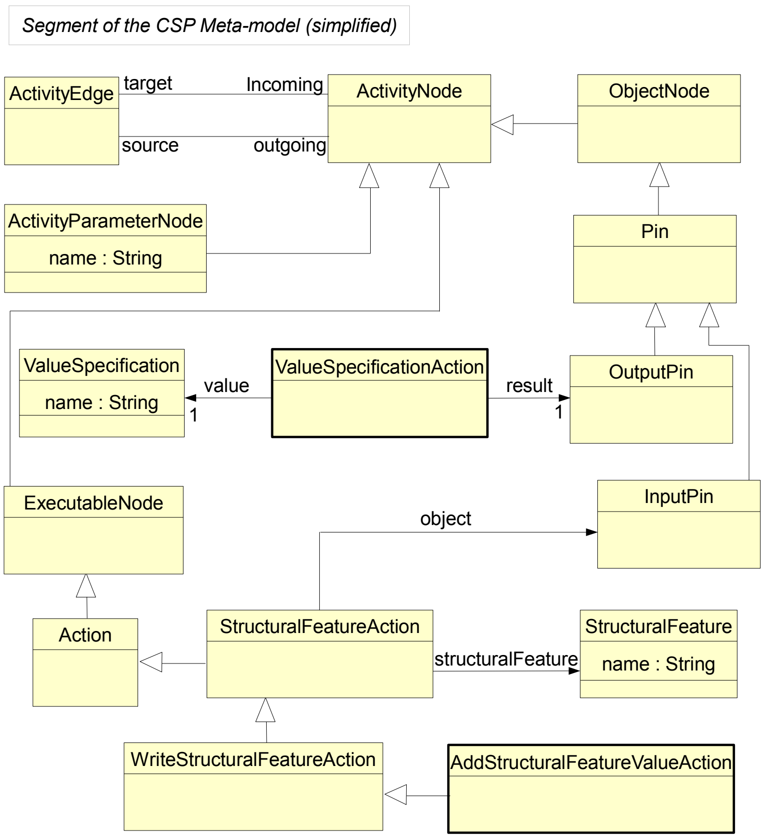
Note: *this rule has been presented in Table 1 as one rule for simplification. However, in the actual ETL code this rule are two rules (one for ValueSpecification action and one for the AddStructuralFeatureValue action) each one creates a parametrized sub-process.*

Rule(5) in ETL

```
rule ValueSpecificationAction_To_SubProcess
    transform action: AD!ValueSpecificationAction
    to pa : CSP!ProcessAssignment,
        prefix: CSP!Prefix,
        next_pid: CSP!ProcessID,
        next_ppl: CSP!ProcessParameterList,
        next_process_var: CSP!ProcessParameterListItem

 {
  pa.processID := getCSP_Process(action);

  prefix.event := createValueSpecificationEvent(
                                       action.value.name);

  next_pid := getCSP_Process(
                  getTargetNode( action.result.outgoing ) );

  next_process_var.name = 'val';

  next_ppl.item.add(next_process_var);
  next_ppl.firstItem := next_process_var;
  next_ppl.size := 1;

  next_pid.parameterList := next_ppl;

  prefix.nextProcess := next_pid;

  pa.processExpression := prefix;

  addToLocalizedProcess (action, pa);
}


rule AddStructuralFeatureValueAction_To_SubProcess
  transform action: AD!AddStructuralFeatureValueAction
  to pa : CSP!ProcessAssignment,
      prefix: CSP!Prefix,
      pid: CSP!ProcessID,
      ppl: CSP!ProcessParameterList,
      process_var: CSP!ProcessParameterListItem
{
  pid := getCSP_Process (action);

  process_var.name = 'val';

  ppl.item.add(process_var);
  ppl.firstItem := process_var;
  ppl.size := 1;

  pid.parameterList := ppl;
  pa.processID := pid;

  prefix.event := createAddStructuralFeatureValueEvent(
                          action.object.incoming.source.name,
                          action.structuralFeature.name);

  prefix.nextProcess := getCSP_Process (
                              getTargetNode(action.outgoing));

  pa.processExpression := prefix;

  addToLocalizedProcess (action, pa);
}
```

Meta-models





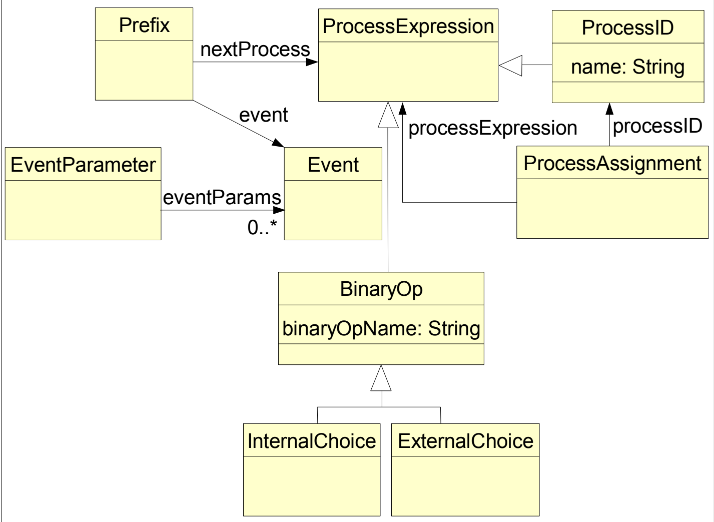
***Note:** This ETL script handles Rule (6) and continue the implementation of Rule (4).*

Rule (6 & 4) in ETL

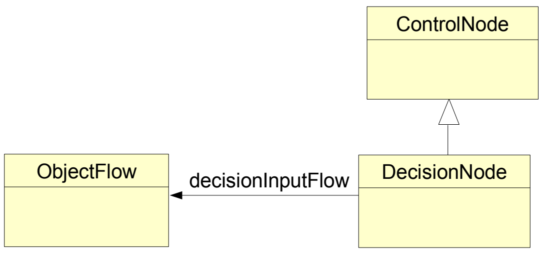
```
rule DecisionNode_To_SubProcess
      transform node: AD!DecisionNode
      to pa : CSP!ProcessAssignment
{
  pa.processID := getCSP_Process (node);

  if (node.incoming.isKindOf(AD!ObjectFlow))
  {
      var extChoice : new CSP!ExternalChoice;
      var acceptedSignals : Sequence;

      extChoice.multiOpName := '[]';

      for (edge in node.outgoing)
      {
        var prefix: new CSP!Prefix;

        prefix.event := createAcceptEvent ( edge.name );
        prefix.nextProcess := getCSP_Process ( edge.target );

        extChoice.expressions.add( prefix );

        acceptedSignals.add(edge.name);
      }
     pa.processExpression := extChoice;
  }
  else - - The incoming edge is Control Flow
  {
      if (node.incoming.size() = 1) - - Decision  & Control Flow
      {
         var intChoice : new CSP!InternalChoice;

         intChoice.multiOpName := '|~|';

         for (edge in node.outgoing)
         {
           var prefix: new CSP!Prefix;

           prefix.event := createInternalChoiceEvent(edge.name);

           prefix.nextProcess := getCSP_Process ( edge.target );

           intChoice.expressions.add( prefix );
         }
         pa.processExpression := intChoice;
     }
     else - - Merge node
     {
         pa.processExpression := getCSP_Process (
                                    getTargetNode( node.outgoing) );
     }
  }

   addToLocalizedProcess (node, pa);

}
```

Meta-models

Segment of the fUML Meta-model (simplified)



Segment of the CSP Meta-model (simplified)



References

1. Abdelhalim, I., Sharp, J., Schneider, S.A., Treharne, H.: Formal Verification of Tokeneer Behaviours Modelled in fUML Using CSP. In: J.S. Dong, H. Zhu (eds.) Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings, *Lecture Notes in Computer Science*, vol. 6447, pp. 371–387. Springer (2010)
2. Altran Praxis: The Tokeneer Project. http://www.adacore.com/tokeneer (cited August 2009)
3. Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., Everett, B.: Engineering the tokeneer enclave protection software. In: 1st IEEE International Symposium on Secure Software Engineering (2006)
4. Barnes, J., Cooper, D.: Tokeneer ID station: Formal Specification. Tech. Rep. S.P1229.41.2, Altran Praxis (2008)
5. Bisztray, D., Ehrig, K., Heckel, R.: Case Study: UML to CSP Transformation. In Applications of Graph Transformation with Industrial Relevance (2007)
6. Cabot, J., Clarisó, R., Riera, D.: Verifying UML/OCL operation contracts. In: IFM '09: Proceedings of the 7th International Conference on Integrated Formal Methods, pp. 40–55. Springer-Verlag, Berlin, Heidelberg (2009)
7. Cooper, D., Barnes, J.: Tokeneer ID station: System Requirements Specification. Tech. Rep. S.P1229.41.1, Altran Praxis (2008)
8. David, H., Amnon, N.: The STATEMATE semantics of statecharts. ACM Trans. Softw. Eng. Methodol. **5**(4), 293–333 (1996). URL http://dx.doi.org/10.1145/235321.235322
9. Goldsmith, M., Armstrong, P.: Personal communication (2010)
10. Graw, G., Herrmann, P.: Transformation and verification of Executable UML models. Electron. Notes Theor. Comput. Sci. **101**, 3–24 (2004)
11. Groote, J.F., Mathijssen, A., Reniers, M., Usenko, Y., van Weerdenburg, M.: The formal specification language mCRL2. In: E. Brinksma, D. Harel, A. Mader, P. Stevens, R. Wieringa (eds.) Methods for Modelling Software Systems (MMOSS), no. 06351 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany (2007)
12. Gruninger, M., Menzel, C.: Process Specification Language: Principles and Applications. AI Magazine **24(3)**, 63–74 (2003)
13. Hansen, H.H., Ketema, J., Luttik, B., Mousavi, M., van de Pol, J.: Towards model checking Executable UML specifications in mCRL2. ISSE pp. 83–90 (2010)
14. Hardin, R.H., Har'El, Z., Kurshan, R.P.: COSPAN **1102** (1996)
15. Hoare, C.: Communicating Sequential Processes. Prentice Hall International Series in Computing Science (1985)
16. Hoare, C., Misra, J., Leavens, G.T., Shankar, N.: The verified software initiative: A manifesto. ACM Comput. Surv. **41**(4), 1–8 (2009)
17. Johnson, D.: Cost effective software engineering for security. In: J. Misra, T. Nipkow, E. Sekerinski (eds.) FM 2006: Formal Methods, *Lecture Notes in Computer Science*, vol. 4085, pp. 607–611. Springer Berlin / Heidelberg (2006)
18. Kolovos, D., Rose, L., Paige, R.: The Epsilon Book. URL http://www.eclipse.org/gmt/epsilon/doc/book/ (last viewed 4th of October 2011)
19. Mellor, S.J., Balcer, M.J.: Executable UML, A Foundation for Model-Driven Architecture. Addison-Wesley (2002)
20. ModelDriven.Org: fUML Reference Implementation. http://portal.modeldriven.org (last viewed 4th of October 2011)
21. Mrugalla, C., Robbe, O., Schinz, I., Toben, T., Westphal, B.: Formal Verification of a Sensor Voting and Monitoring UML Model. In: S.H. Houmb, J. Jürjens, R. France (eds.) Proceedings of the 4th International Workshop on Critical Systems Development Using Modeling Languages (CSDUML 2005). Technische Universität München, Fredrikstad, Norway (2005)
22. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4), 541–580 (1989)
23. Ng, M.Y., Butler, M.: Towards formalizing UML state diagrams in CSP. In: A. Cerone, P. Lindsay (eds.) 1st IEEE International Conference on Software Engineering and Formal Methods, pp. 138–147. IEEE Computer Society (2003)
24. OMG: XML Metadata Interchange (XMI) (Version 2.1.1)
25. OMG: Unified modeling language (UML) superstructure (version 2.3) (2010)
26. OMG: Semantics of a foundational subset for executable UML models (fUML) - Version 1.0 (2011)
27. Planas, E., Cabot, J., Gómez, C.: Verifying action semantics specifications in UML behavioral models. In: CAiSE '09: Proceedings of the 21st International Conference on Advanced Information Systems Engineering, pp. 125–140. Springer-Verlag, Berlin, Heidelberg (2009)
28. Quick Sequence Diagram Editor - v3.1: http://sdedit.sourceforge.net/ (last viewed 4th of October 2011)
29. Roscoe, A.W., Wu, Z.: Verifying statemate statecharts using CSP and FDR. In: ICFEM, pp. 324–341 (2006)
30. Schneider, S., Treharne, H.: CSP theorems for communicating B machines. Formal Asp. Comput. **17**(4), 390–422 (2005)
31. Shah, S.M.A., Anastasakis, K., Bordbar, B.: From UML to Alloy and back again. In: MoDeVVa '09: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation, pp. 1–10. ACM, New York, NY, USA (2009)
32. Thierry-Mieg, Y., Hillah, L.M.: UML behavioral consistency checking using instantiable Petri nets. ISSE **4**(3), 293–300 (2008)
33. Treharne, H., Turner, E., Paige, R.F., Kolovos, D.S.: Automatic Generation of Integrated Formal Models Corresponding to UML System Models. In: TOOLS (47), pp. 357–367 (2009)
34. Turner, E., Treharne, H., Schneider, S., Evans, N.: Automatic generation of CSP||B skeletons from xUML models. In: Proceedings of the 5th international colloquium on Theoretical Aspects of Computing, pp. 364–379. Springer-Verlag, Berlin (2008)
35. UML2 Project: http://www.eclipse.org/modeling/mdt/?project=uml2 (last viewed 4th of October 2011)
36. Varró, D., Asztalos, M., Bisztray, D., Boronat, A., Dang, D.H., Geiß, R., Greenyer, J., Van Gorp, P., Kniemeyer, O., Narayanan, A., Rencis, E., Weinell, E.: Transformation of UML Models to CSP: A Case Study for Graph Transformation Tools. Applications of Graph Transformations with Industrial Relevance pp. 540–565 (2008)

37. Woodcock, J., Aydal, E.G.: A Token Experiment. Festschrifts in Computer Science, the BCS FAC Series, Festschrift for Tony Hoare (2009)
38. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal methods: Practice and experience. ACM Comput. Surv. **41**(4), 1–36 (2009)
39. Xie, F., Levin, V., Browne, J.C.: Model checking for an executable subset of UML. In: ASE '01: Proceedings of the 16th IEEE International Conference on Automated Software Engineering, p. 333. IEEE Computer Society (2001)
40. Xu, D., Miao, H., Philbert, N.: Model checking UML activity diagrams in FDR. In: ICIS '09: Proceedings of the 2009 Eigth IEEE/ACIS International Conference on Computer and Information Science, pp. 1035–1040. IEEE Computer Society, Washington, DC, USA (2009)
41. Xu, D., Philbert, N., Liu, Z., Liu, W.: Towards formalizing UML activity diagrams in CSP. In: ISCSCT '08: Proceedings of the 2008 International Symposium on Computer Science and Computational Technology, pp. 450–453. IEEE Computer Society, Washington, DC, USA (2008)
42. Zakiuddin, I., Moffat, N., O'Halloran, C., P.Ryan: Chasing events to certify a critical system. Tech. rep., UK DERA (1998)