Formal Verification of Tokeneer Behaviours Modelled in fUML Using CSP

Islam Abdelhalim, James Sharp, Steve Schneider, and Helen Treharne

Department of Computing, University of Surrey

Abstract. Much research work has been done on formalizing UML diagrams, but less has focused on using this formalization to analyze the dynamic behaviours between formalized components. In this paper we propose using a subset of fUML (Foundational Subset for Executable UML) as a semi-formal language, and formalizing it to the process algebraic specification language CSP, to make use of FDR as a model checker. Our formalization includes modelling the asynchronous communication framework used within fUML. This allows different interpretations of the communications model to be evaluated. To illustrate the approach, we use the modelling of the Tokeneer ID Station specifications into fUML, and formalize them in CSP to check if the model is deadlock free.

1 Introduction

The OMG (Object Management Group) has developed the fUML (Foundational Subset for Executable UML) [1] specification with the purpose of enabling compliant models to be transformed into various executable forms for verification, integration, and deployment. The specification of the execution model incorporates a degree of genericity. This is achieved mainly by defining explicit semantic variation points. A particular execution tool can then realize specific semantics by providing specifications for any of those points. The *semantics of inter-object communications mechanisms* is one of the semantic variation points. The choice of the implementation of such a point may affect the execution model of the system. We use formal model checking to evaluate an implementation of the inter-object communications mechanism and its compatibility with an fUML model. This, in turn, can ensure that faults are detected during the early stages of software development lifecycle, which provide significant savings in cost compared with rectifying errors after the system has been implemented.

We formalize fUML models into the process algebraic specification language CSP [2] with the purpose of checking deadlocks in the models, which could happen if all objects are waiting to accept signals from each other. To achieve checking such a property, we modelled the inter-object communications mechanism into CSP. Then we depended on FDR (Failures-Divergences Refinement) [3] to handle the model checking and report the deadlock scenarios (if found). Potentially, we

can make use of this methodology to allow software engineers, who do not have specialist mathematical knowledge, to formally check their semi-formal (fUML) models.

Much research exists on translating a UML model to a formal model (see Section 8) but many approaches have imposed restrictions on the UML diagrams and notation used. Our previous work on transforming xUML (Executable UML) [4] to $CSP \parallel B$ [5] benefits from the fact that using xUML as a starting point means that the restrictions are those imposed by the language itself. xUML allows models to be executed and tools supporting xUML have meta-models [6]. In this paper we propose to examine the translation between fUML and CSP. fUML benefits from a standardized meta-model. Moreover, the complexity of the queuing mechanism in xUML (each object has a separate signal queue for all the objects in the system including itself which has the highest priority) leads to more complex CSP models, and thus more heavy processes to be analyzed by FDR. Also the syntax of the fUML activity diagrams allowed for modelling the internal choices, which was not possible using xUML. The fUML models yield more abstract formal models than were possible from xUML translation.

In order to validate our approach, we modelled part of the Tokeneer ID Station project [7] into fUML. We then developed a group of mapping rules (maps between the fUML activity diagrams elements and CSP) to manually translate the fUML into CSP, and checked if the formalized model is deadlock free. In this paper we are focusing on a subset of the fUML activity diagram notation, and this has been guided by the Tokeneer case study. Formalizing fUML class diagrams is beyond the scope of this paper, as we want to check the dynamic behaviour of the system. Also the runtime objects creation and deletion are not covered in this work.

Overall the main novelty of this work is the proposed approach to model the asynchronous communication between objects. Also to our knowledge, this is the first attempt to analyze fUML models (by formalizing them into CSP) and addressing the semantic variation points issue in the fUML standard.

We assume the reader of this paper has good background knowledge with the UML 2 standard, CSP, and FDR.

The rest of this paper is organised as follows. In Section 2, we give a background to the fUML standard and the CSP syntax used in this paper. In Section 3, we introduce the Tokeneer case study briefly and include part of the used fUML diagrams. In Section 4, we describe the mapping rules we developed to map between the fUML activity diagram elements and CSP. In Section 5, we describe how we model the asynchronous communication between the fUML active objects in CSP. In Section 6, we describe the CSP model for a selected part of Tokeneer fUML activity diagram. In Section 7, we discuss the deadlock checking for the formalized fUML model and the generated results from FDR. Finally, we discuss related work and conclude in Sections 8 and 9 respectively.

2 Background

2.1 fUML

fUML (Foundational Subset for Executable UML) [1] is an OMG standard acting as an intermediary between "surface subsets" of UML models and platform executable languages. The OMG defines the fUML subset by specifying the modifications to the original abstract syntax (of UML 2) of the class and activity diagrams, which allows fUML models to be transformed to various executable forms. These modifications are specified in clause 7 of the standard by merging/excluding some packages in the UML 2 specification [8], as well as adding new constraints. As defined in the fUML standard, we are listing below some modifications to UML 2 that are relevant to our Tokeneer fUML model; all of them are related to the fUML activity diagrams since our goal is to capture the behaviours 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 using 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).

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 [1]. Such communication is conducted between active objects only. Active objects in fUML communicate asynchronously via signals (kind of classifier). This is achieved by associating an object activation with each object which handles the dispatching of asynchronous communications received by its active object. Figure 1 shows the structure related to the object activation.

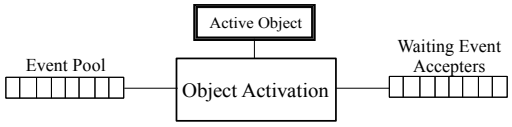


Fig. 1: Object Activation Structure

The 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 dispatches events on a FIFO (first-in first-out) basis.

2.2 CSP

CSP (Communication Sequential Processes) 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.

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 and then behave subsequently as $P(x)$. The output process $c!v \rightarrow P$ will output v along channel c and then behave as P. Channels can have any number of message fields, combination of input and output values. The choice $P \Box Q$ offers an external choice between processes P and Q whereby the choice is made by the environment. The choice $P \sqcap Q$ offers an internal choice between processes P and Q whereby the choice is made by the process. The parallel combination $P \underset{A}{\|}_{B} Q$ executes P and Q in parallel. P can perform only events in A, Q can perform only events in B, and they must simultaneously engage in events in the intersection of A and B. $P \setminus A$ operation describes the case where all participants of all events in A are described in P. All these events are removed from the interface of the process, since no other processes are required to engage in them. $P;\ Q$ initially executes P. When P successfully terminates, then control passes to Q. This composition can be replicated over a sequence of expressions using the form $;\ x : s @ P$.

3 Tokeneer: Case Study Introduction

Tokeneer [7] was a successful research project carried out by Altran Praxis and funded by the National Security Agency (NSA) to develop part of an existing secure system (the Tokeneer System) to investigate SPARK's capabilities to develop high quality and low defect software. The project specifications are available in the formal language Z, and the open source implementation and proofs are available in SPARK Ada.

The entire Tokeneer system is a larger system that provides protection to secure information held on a network of workstations situated in a physically secure enclave. However, the project and this paper focuses on one part of the system, Tokeneer ID Station (TIS) subsystem. Based on the available project specifications [7], we have chosen to model a subset of components of the TIS project into fUML. The selected components include the main dynamic behaviours of the system (our work scope), while other (un-modelled) components act as a data repository. Each component is modelled as a class, and the following class diagram in Figure 2 shows the relationships between those classes. We do not formalize the class diagram, and its inclusion is just to illustrate the relationship between the system's components.

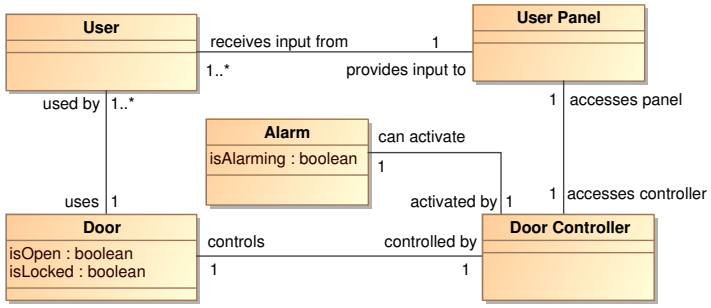


Fig. 2: TIS Class Diagram

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.

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. Due to the space constraints, we just focus on a segment of the Door Controller activity (depicted in Figure 3), which includes all the described elements in Section 4.

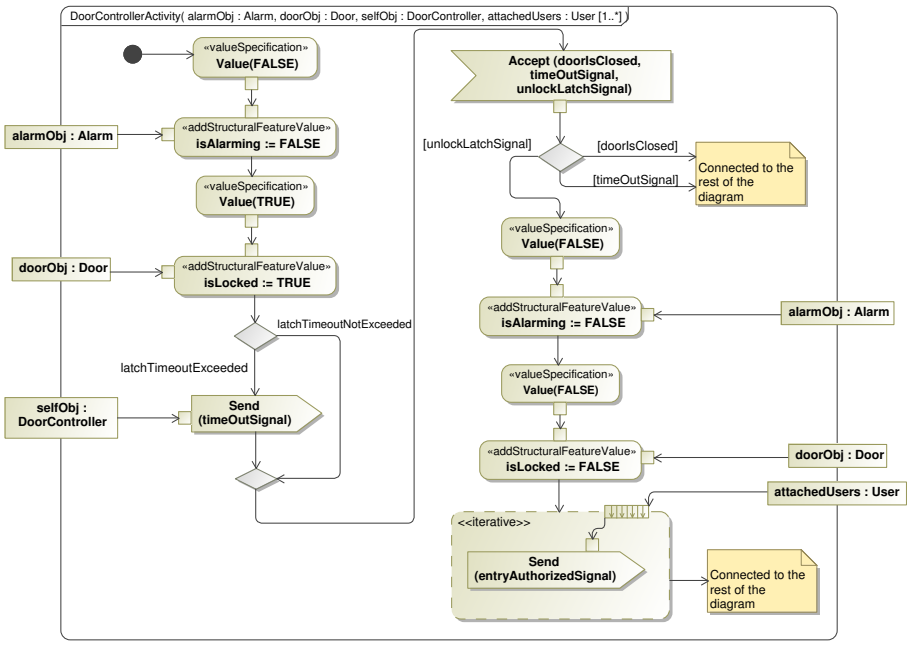


Fig. 3: Segment of the Door Controller Activity

Initially, the Door Controller sets the *isAlarming* variable to FALSE, and *isLocked* to TRUE. At this point the Door's *isOpen* attribute is TRUE, which means the door is open and its latch is locked. For that reason the Door Controller starts a timer to watch this suspicious situation. The activity then represents the two possible scenarios for this timer (*timeOutExceeded*, or *timeOutNotExceeded*). If the timer timeouts the Door Controller sends the *timeOutSignal* to itself to fire the alarm, otherwise it continues with the normal flow. In both cases, the activity waits for one of the following signals to arrive: *closeDoorSignal*, *timeOutSignal*, or *unlockLatchSignal*. If the *unlockLatchSignal* arrives from the User Panel, the Door Controller sets *isLocked* to FALSE, and then sends the *entryAuthorizedSignal* to all Users' objects in the system.

4 Modelling fUML Activity Diagrams into CSP

In this section we outline the used mapping rules to formalize Tokeneer fUML activity diagrams into CSP. Table 1 shows the fUML activity diagram's elements and the corresponding CSP representation that reflects the semantic behaviour for each element. As the automatic transformation (from fUML to CSP) is out of this paper's scope, we describe the mapping informally (mapping rules) instead of formally defining transformation rules.

In the above mapping rules, *aIH* and *bIH* represent the instance handler of the sender and receiver objects respectively. *rp1* and *rp2* represent the points where the object (*bIH*) is waiting to accept the signal instances *sig1* and *sig1*, *sig2*, or *sig3* respectively.

Mapping between UML activity diagrams and CSP has been addressed several times in the literature [9, 10]. The novel points of our mapping are as follows:

Rule(1) maps the fUML activity as a parent CSP process that can accept different parameters (*param1*, *param2*, ..). Within this process we define sub-processes, each act 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 fills the *waiting event accepters* list with the allowed signals to be accepted at this point (registration point). Section 5 describes how those events synchronize with the object's buffer process (*BUF_SYS*) to allow the asynchronous communication between processes (active objects).

The fUML standard supports 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 semantic. Rules like (2),(3), and (4) are not presented in [9, 10] because their focus 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 fUML standard allowed for modelling internal decisions which was not possible using xUML. Rule(7) maps the iterative *ExpansionRegion* as a CSP sequential

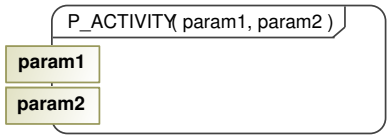
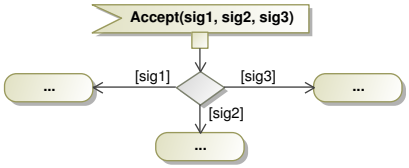
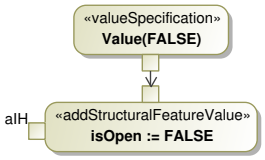
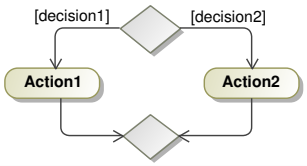
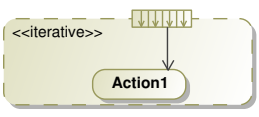
fUML Element	CSP Representation
Rule(1): Activity 	$P\_ACTIVITY(param1, param2) =$ let $\quad Activity/Process\ Body$ $within\ AC1$
Rule(2): Send Signal Action 	$AC1 = send!aIH!bIH!sig1 \rightarrow ...$
Rule(3): Accept Signal Action 	$AC1 = registerSignals!bIH!rp1 \rightarrow$ $\quad accept!bIH!sig1 \rightarrow ...$
Rule(4): Accept Signal Action (*) 	$AC1 = registerSignals!bIH!rp2 \rightarrow ($ $\quad accept!bIH!sig1 \rightarrow ...$ $\quad \square$ $\quad accept!bIH!sig2 \rightarrow ...$ $\quad \square$ $\quad accept!bIH!sig3 \rightarrow ...)$
Rule(5): Add Structural Feature Value Action 	$AC1 =$ $valueSpec!aIH?value : \{FALSE\} \rightarrow$ $addStFeatureValue!aIH!isOpen!value$ $\quad \rightarrow ...$
Rule(6): Decision/Merge Nodes 	$DS1 = decision1 \rightarrow AC1$ $\quad \sqcap$ $\quad decision2 \rightarrow AC2$ $AC1 = ... \rightarrow MR1$ $AC2 = ... \rightarrow MR1$ $MR1 = ...$
Rule(7): Expansion Region 	$AC1 = ER\_IT1(< e1, e2, e3 >)$ $ER\_IT1(<>) = AC2$ $ER\_IT1(seq) =;\ s : seq@Action1!s \rightarrow$ $\quad ER\_IT1(tail(seq))$ $AC2 = ...$

Table 1: fUML to CSP Mapping Rules

composition which repeats the action(s) inside the region (*Action1*) with the number of elements inside the sequence *seq*.

Our CSP representation does not include all the properties of the fUML activity diagram elements, as we just focus on the properties in the Tokeneer fUML model. For example, the formalization of the *addStructuralFeatureValueAction* considers the assignment of unordered boolean structural features only.

5 Modelling the fUML Communication Mechanism in CSP

In Section 2.1 we described how active objects in fUML communicate with each others asynchronously. In this section we describe the formalization of this communication mechanism by modelling its semantics into CSP. The model avoids depending on the sequence data structure or Haskell functions, as they lead to a significant decay in FDR performance during the compilation process. For that reason, this implementation uses the CSP primitives only (parallel composition, prefix, etc.). The idea of this implementation came from Michael Goldsmith [11].

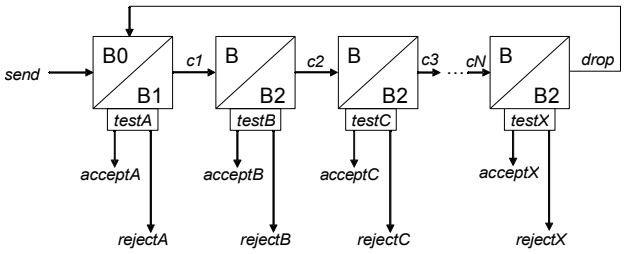


Fig. 4: The Event Pool as a Controlled Buffer

As shown in Figure 4, the idea is built on representing the *event pool* as a buffer with N consecutive nodes. When an object sends a signal to another object (perform 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 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 becomes full, the oldest signal in the buffer (placed in the rightmost node) will be dropped out (*drop* event) and all the signals will be shifted right by one node. Signals are moved down as a parameters to the *c1, c2, . . . , cN* events. We represent each of those nodes as a CSP process with a simple logic illustrated in Figure 5 for the first node ($B0$) and the general node (B).

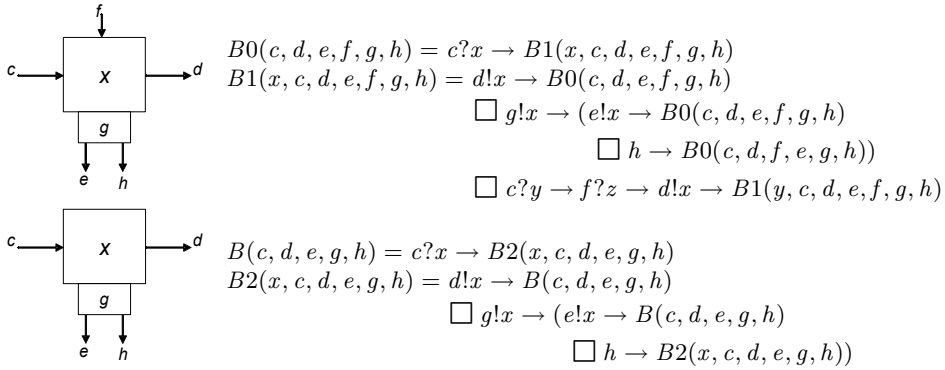


Fig. 5: Buffer's First Node

Figure 6 shows the parallel combination of three nodes forming the *event pool* of the Door Controller instance (*dIH0*) which can hold three signal instances at a time.

$$BUF\_dcIH = ((B0(send,\ c1,\ acceptA,\ drop,\ testA,\ rejectA) \underset{\{|c1|\}}{||} B(c1,\ c2,\ acceptB,\ testB,\ rejectB)) \underset{\{|c2,drop|\}}{||} B(c2,\ drop,\ acceptC,\ testC,\ rejectC)) \setminus \{|\ c1,\ c2,\ drop\ |\}$$

Fig. 6: Three Nodes Controlled Buffer

In previous attempts, we have used the default signals dispatching strategy (FIFO) for modelling the inter-object communication. However, this revealed a serious problem when an object receives an unexpected signal (not matched to one of the *waiting event accepters*): the object dismisses it directly because it was removed from its *event pool* for matching and the fUML standard does not allow signals to be returned back to the *event pool*. In many cases the object will need to accept this dismissed signal after some further actions, resulting in a fast invalid deadlock. As the fUML standard allows for overriding the default dispatching strategy, we implemented it by not removing any signal from the *event pool* unless it is registered in its *waiting event accepters* list, to avoid signals dropping. At the same time, signals are dispatched in chronological order (i.e. remove the oldest signal from the *event pool* first) to maintain dispatching signals in the same order they were sent. To meet this logic, we developed a controller process (*BUF_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 7 shows our representation of the buffer controller process (BUF_CTRL_dIH) for the Door Controller instance (dIH). The *getRegisteredSignals* is a mapping function that returns the allowed signal(s) at a certain registration point (rp). Note that *registerSignals* event will synchronize with the corresponding event in the translation of the diagram (Rule(3) and (4)).

$$
\begin{aligned}
BUF\_CTRL\_dcIH(\{\}) = {} & registerSignals!dcIH?rp \rightarrow \\
& BUF\_CTRL\_dcIH(getRegisteredSignals(dcIH, rp))
\end{aligned}
$$

$$
\begin{aligned}
BUF\_CTRL\_dcIH(EA) = {} & testC?x \rightarrow if(member(x, EA))\ then \\
& \quad (acceptC!x \rightarrow BUF\_CTRL\_dcIH(\{\})) \\
& \quad else\ rejectC \rightarrow \\
& testB?x \rightarrow if(member(x, EA))\ then \\
& \quad (acceptB!x \rightarrow BUF\_CTRL\_dcIH(\{\})) \\
& \quad else\ rejectB \rightarrow \\
& testA?x \rightarrow if(member(x, EA))\ then \\
& \quad (acceptA!x \rightarrow BUF\_CTRL\_dcIH(\{\})) \\
& \quad else\ rejectA \rightarrow BUF\_CTRL\_dcIH(EA)
\end{aligned}
$$

Fig. 7: The Buffer Controller Process of the Door Controller Instance

We depend on the *chase* compression function to complete the definition. The main functionality of *chase* is that it forces FDR to follow the *tau's* only during the transition system analysis (i.e. if there is more than possible branch and *tau* is one of them, *chase* will force FDR to follow the *tau* branch and neglect the other branches). If there is more than one *tau* branch available, *chase* will make FDR chooses any one of them arbitrarily. The chase function is not semantics-preserving, in fact it produces a refinement of the original process. However, here it produces exactly what is required.

Figure 8 shows how we made use of the *chase* function by applying it on the parallel combination between the buffer (BUF_dcIH) and the buffer controller (BUF_CTRL_dcIH) of the Door Controller instance after hiding the buffer internal events (*test*, *reject*, *c*, and *drop*) for all nodes (grouped in *aHiddenEvents*). Having those events hidden (*tau's*), FDR will follow them causing signals to be propagated along the nodes whenever a *send* event happens. The set *aSynchEvents* contains the synchronization events: *test*, *reject*, and *accept* for all nodes.

$$
BUF\_SYS\_dcIH = chase((BUF\_dcIH \underset{aSynchEvents}{\Big\|} BUF\_CTRL\_dcIH(\{\})) \setminus aHiddenEvents)
$$

Fig. 8: Buffer System Process of the Door Controller Instance

The described implementation in this section came after several attempts to model the fUML inter-object communication (semantic variation point). The previous attempts were suffering from various problems like: quick incorrect dropping of signals (which leads to an invalid deadlock), heavy CSP scripts that could not be compiled by FDR, and not maintaining the signals sending order. Having the Tokeneer fUML model formalized in CSP allowed for evaluating those attempts before the actual implementation of the system.

6 Corresponding CSP for Tokeneer fUML Model

Applying the mapping rules of Section 4 to the activity diagram shown in Figure 3 yields the following CSP process in Figure 9.

$$
\begin{array}{l}
DOOR\_CTRL(alarmObj,\ doorObj,\ selfObj,\ attachedUsers) = \\
let \\
\quad AC1 \ = \ valueSpec!selfObj?value : \{FALSE\} \ \rightarrow \\
\qquad\quad addStFeatureValue!alarmObj!isAlarming!value \rightarrow AC2 \\
\quad AC2 \ = \ valueSpec!selfObj?value : \{TRUE\} \rightarrow \\
\qquad\quad addStFeatureValue!doorObj!isLocked!value \rightarrow DS1 \\
\quad DS1 \ = \ latchTimeoutExceeded \rightarrow AC3 \ \sqcap \ latchTimeoutNotExceeded \rightarrow MR1 \\
\quad AC3 \ = \ send!selfObj!selfObj!timeOutSignal \rightarrow MR1 \\
\quad MR1 \ = \ registerSignals!selfObj!rp1 \rightarrow (accept!selfObj!doorIsClosedSignal \rightarrow ... \\
\qquad\qquad\qquad\qquad\qquad\qquad \Box \\
\qquad\qquad\qquad\qquad\qquad\qquad accept!selfObj!timeOutSignal \rightarrow ... \\
\qquad\qquad\qquad\qquad\qquad\qquad \Box \\
\qquad\qquad\qquad\qquad\qquad\qquad accept!selfObj!unlockLatchSignal \rightarrow AC7) \\
\quad AC7 \ = \ valueSpec!selfObj?value : \{FALSE\} \ \rightarrow \\
\qquad\quad addStFeatureValue!alarmObj!isAlarming!value \rightarrow AC8 \\
\quad AC8 \ = \ valueSpec!selfObj?value : \{FALSE\} \rightarrow \\
\qquad\quad addStFeatureValue!doorObj!isLocked!value \rightarrow ER\_IT1(attachedUsers) \\
\quad ER\_IT1(<>) = ... \\
\quad ER\_IT1(users) = ; \ \ u : users@send!selfObj!u!entryAuthorisedSignal \rightarrow \\
\qquad\qquad\qquad\qquad ER\_IT1(tail(users)) \\
within\ AC1
\end{array}
$$

$$
DOOR\_CTRL\_BUF = DOOR\_CTRL(aIH, dIH, dcIH, uIHS) \underset{aAdcIH}{\|} BUF\_SYS\_dcIH \underset{aBdcIH}{}
$$

Fig. 9: The Corresponding CSP Process for the Door Controller Activity Segment

The processes *AC1*, *AC2*, *DS1*, *AC7*, and *AC8* are direct implementation to the mapping rules described in Section 4 and they are not involved in the asynchronous communication between the active objects.

The process *DOOR_CTRL_BUF* represents the parallel combination between the *DOOR_CTRL* process (represents the Door Controller active object) and its object activation represented formally by the *BUF_SYS_dcIH* process. When the *send* event happens at *AC3* it synchronizes with the *send* event in the *BUF_dcIH* to push the *timeOutSignal* inside the Door Controller *event pool* (controlled buffer). After that, the *registerSignals* at *MR1* happens which synchronizes with the *registerSignals* event in the *BUF_CTRL_dcIH* process, which in turns fills the *waiting event accepters* set (*EA*) with the allowed signals at this point (*doorIsOpenSignal*, *timeOutSignal*, and *unLockLatchSignal*). At this point, the *accept* event in *MR1* synchronizes with the *accept* event in the *BUF_CTRL_dcIH* process which will happen only if the accepted signal is member of *EA*. The set *aAdcIH* includes all the alphabets of the Door Controller process (*DOOR_CTRL*). The set *aBdcIH* includes the *accept* and *registerSignals* events for all signals the Door Controller accepts, plus the *send* events for any other object sends a signal to the Door Controller process.

7 Deadlock Checking

After formalizing the Tokeneer fUML model into CSP, it becomes a direct process to check the behaviour of the model using FDR. In this paper we focus on checking the deadlock which can happen if all the objects in the system are waiting to accept signals from each others. FDR has the capability of checking such property by determining whether the process (*SYSTEM*) can reach a state in which no further actions are possible, and if a deadlock is found, FDR generates the traces (counter example) that led to this deadlock. Figure 10 shows the main process (*SYSTEM*) which represents the whole system as a parallel combination between all components (for the space we are including here the Door Controller and the User Panel components only).

$$SYSTEM = USER\_PANEL\_BUF \underset{\cup(aAupIH, aBupIH)}{\|} DOOR\_CTRL\_BUF \quad \cup(aAdcIH, aBdcIH)$$

Fig. 10: The *SYSTEM* Process

Tokeneer Deadlock Checking
We managed to check for deadlock in the Tokeneer model for the *SYSTEM* process using FDR. The system includes four interacting processes (Door, Door Controller, User, and User Panel), and each process has its own *event pool* (*BUF_SYS*) with 10 slots. FDR managed to compile the CSP script (more than 600 lines) in less than a second, and the model checking reported several deadlock scenarios (counter examples). Figure 11 shows part of one of those sce-

narios represented visually as a sequence diagram to simplify understanding the problem.

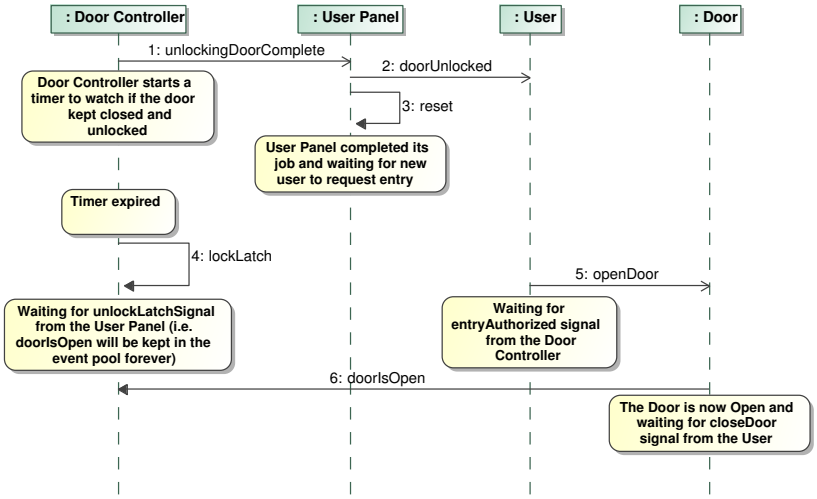


Fig. 11: Sample Scenario Caused a Deadlock

It is obvious that eventually all the objects are waiting for each others causing deadlock. This happens because the user takes a long time (more than the timer period) to open the door after getting the permission to enter from the User Panel. This deadlock was only revealed once we had implemented the asynchronous communication mechanism of Section 5. During experiments with more primitive communication mechanisms, artificial deadlocks arose because of the inability to process signals. We could not execute the model sufficiently in order to reach the state of Figure 11 because signals were being dropped incorrectly.

8 Related Work

There is a significant body of work relating transforming UML diagrams into formal methods. Among these attempts, some of them, e.g. [12–15], focus on formalizing the standard UML diagrams into: Z, Promela, B, or CSP (Constraint Satisfaction Problem). The authors in [12, 13] focus on checking consistency between UML diagrams, whereas the authors in [14, 16] check refinement between UML models. Translation of UML to CSP has been included in [17, 18] which describe how to check the model dynamic behaviours and visualize the formal language into a graphical notation. Our work is more closely compared to [9, 10] and [19] which consider the formalization of activity diagrams into CSP, and Petri nets respectively. The authors in [19] focus on checking deadlocks in the UML models, which is aligned to our work. However, none of them addressed modelling the asynchronous communication between objects formally.

Formally representing the asynchronous communication between objects has been discussed in a limited way in [20, 21, 5] where part of the xUML [4] was formalized, which specify a way of communication different from fUML. On the

other hand, [22] simulated the asynchronous message passing by synchronous communication between processes modelling objects and their message queues.

To our knowledge, our work is the first attempt to formalize the fUML standard, and formally represents the asynchronous communication between its active objects. This formal representation provided a formal way to evaluate different implementations (interpretations) of the signals dispatching mechanism (one of the semantic variation points in the fUML standard).

9 Conclusion and Future Work

An approach to model check fUML activity diagrams by representing them in CSP has been presented in this paper. The approach allowed for modelling the inter-object communication (fUML standard semantic variation point) in different ways until reaching a successful implementation. The approach has successfully demonstrated that deadlocks can be detected automatically. Using this approach, the analysis of Tokeneer fUML model CSP representation using FDR, has detected several deadlock scenarios. This means that the model contains some errors, which need to be resolved in the view of the asynchronous communication between objects.

Using the implementation of the communication mechanism described in Section 5, FDR succeeded in compiling the CSP script in less than a second for a 10 slots *event pool* for each object. Also FDR did not report any dropping of signals using this implementation. However, we do expect signals dropping if the model gets more complicated (more events), which means identifying the right *event pool* size is very critical to keep the system alive. Currently we do not have a particular methodology to identify the correct object's *event pool* size.

fUML syntax allowed to model all aspects we were interested within Tokeneer. However, we recommend that the value pin be included in the fUML subset, as its exclusion led to a complicated model full of the valueSpecification action.

Currently the transformation from fUML to CSP is done manually based on the mapping rules. In the future, we are intending to automate this process using one of the MDA approaches and making use of the available fUML and CSP meta-models. When we tried to check one of Tokeneer safety specifications we faced a state explosion problem, because FDR generates exponential increasing states when it does the verification. For that reason we may need to represent the fUML model into another formal language and use theorem provers to check such properties.

Acknowledgments Thanks to Michael Goldsmith and Philip Armstrong for discussion about implementing the buffer in CSP. Thanks also to Ian Wilkie for his helpful information about fUML.

References

1. OMG: Semantics of a foundational subset for executable UML models (fUML) - (Beta 2), http://www.omg.org/spec/fuml/1.0 (November 2009)

2. Schneider, S.: Concurrent and Real-Time Systems: the CSP Approach. Wiley (1999)
3. Formal Systems Oxford: FDR 2.83 manual (2007)
4. Mellor, S.J., Balcer, M.J.: Executable UML, A Foundation for Model-Driven Architecture. Addison-Wesley (2002)
5. 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, Berlin, Springer-Verlag (2008) 364–379
6. Wilkie, I., King, A., Clarke, M., Weaver, C., Raistrick, C., Francis, P.: UML ASL Reference Guide (ASL language level 2.5). Kennedy Carter Ltd. (2003)
7. Cooper, D., Barnes, J.: Tokeneer ID station: System requirements specification. Technical Report S.P1229.41.1, Praxis High Integrity Systems (August 2008)
8. OMG: Unified modeling language (UML) superstructure (version 2.2) (2009)
9. 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, Washington, DC, USA, IEEE Computer Society (2008) 450–453
10. 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, Washington, DC, USA, IEEE Computer Society (2009) 1035–1040
11. Goldsmith, M., Armstrong, P.: Personal communication (February 2010)
12. Amalio, N., Stepney, S., Polack, F.: Formal proof from UML models. In: ICFEM 2004, volume 3308 of LNCS, Springer (2004) 418–433
13. Zhao, X., Long, Q., Qiu, Z.: Model checking dynamic UML consistency. In: ICFEM. (2006) 440–459
14. Ammar, B.B., Bhiri, M.T., Souquières, J.: Incremental development of UML specifications using operation refinements. ISSE **4**(3) (2008) 259–266
15. Cabot, J., Clarisó, R., Riera, D.: Verifying UML/OCL operation contracts. In: IFM '09: Proceedings of the 7th International Conference on Integrated Formal Methods, Berlin, Heidelberg, Springer-Verlag (2009) 40–55
16. Pons, C.: Heuristics on the definition of UML refinement patterns. In: 32nd Int. Conf. on Current Trends in Theory and Practice of Computer Science. SOFSEM. Czech Republic, Springer (2006) 461–470
17. Ng, M.Y., Butler, M.J.: Tool support for visualizing CSP in UML. In: ICFEM '02: Proceedings of the 4th International Conference on Formal Engineering Methods, London, UK, Springer-Verlag (2002) 287–298
18. Ng, M.Y., Butler, M.: Towards formalizing UML state diagrams in CSP. In Cerone, A., Lindsay, P., eds.: 1st IEEE International Conference on Software Engineering and Formal Methods, IEEE Computer Society (2003) 138–147
19. Thierry-Mieg, Y., Hillah, L.M.: UML behavioral consistency checking using instantiable Petri nets. ISSE **4**(3) (2008) 293–300
20. Hansen, H.H., Ketema, J., Luttik, B., Mousavi, M., van de Pol, J.: Towards model checking Executable UML specifications in mCRL2. ISSE (2010) 83–90
21. Graw, G., Herrmann, P.: Transformation and verification of Executable UML models. Electron. Notes Theor. Comput. Sci. **101** (2004) 3–24
22. 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, Washington, DC, USA, IEEE Computer Society (2001) 333