2009 Eigth IEEE/ACIS International Conference on Computer and Information Science

Model Checking UML Activity Diagrams in FDR+

Dong Xu, Huaikou Miao, Nduwimfura Philbert
School of Computer Engineering and Science
Shanghai University
Shanghai, China
dxu@shu.edu.cn

***Abstract*—The Unified Modeling Language (UML) is the de-facto industrial standard for modeling object-oriented software systems. UML Activity diagrams (ADs) can be used for software modeling and they have undergone significant changes with UML 2.0 specification, and no longer been a special kind of diagrams of UML state machines. ADs, however, are lack of formal semantics like other diagrams in UML official specifications and therefore they cannot be performed formal system behavior analysis. Verify if a UML model meets the required properties by model checking has become a key issue. Many attempts have been made to provide formalization for UML State Diagrams, but little for ADs. This paper formalize the UML ADs and then employs the Hoare's Communicating Sequential Processes to specify its operational and behavior semantics. The presented formalization makes it possible to reason about the behavior of a UML AD in CSP. A software tool for supporting to transfer an AD into CSP_M specification has been implemented. Hence it provides an approach to model checking UML ADs by model-checker FDR. Finally, the formalisms and model checking are explicitly illustrated through a simple but non-trivial example.**

Keywords- Model Checking; UML Activity Diagrams; CSP; FDR

I. INTRODUCTION

The Unified Modeling Language (UML) is the de-facto industrial standard for modeling object-oriented software systems. It is a standard visual modeling language that is designed to specify, visualize, construct and document the artifacts of software systems. UML Activity diagrams (ADs) can be used for software modeling, and activities typically represent a behavior. Hence they are widely used during system modeling stage. ADs, however, have undergone significant changes with UML 2.0 specification, and no longer borrow elements from UML state machines. The Object Management Group (OMG) provides the official definition of UML ADs and an on-going work has been carried out to improve the specification of UML diagrams. Unfortunately there is still lack of formal basis for system analysis in UML ADs [1]. Various approaches have been made to provide formalization for UML State Diagrams in previous work [2, 3, 4], but less work has been carried out to give a formal semantics to the UML ADs. An essential task of modeling business activities is to identify the different types of business transactions and the order in which they are conducted. Much research has been carried out on refining and integrating models for specifying activity processes, with the aim of ensuring correctness and making the task more efficient and manageable. Formal methods are widely regarded as a useful means of increasing software reliability. The formal verification of activity diagrams has been an active research topic ever since UML was proposed.

In this paper, an approach to formalizing the ADs in terms of CSP [5] is presented, which allows model-checking using FDR [7]. A subset of UML ADs is considered, and actions, fork-join pairs, interruptible nodes, activities, final nodes as well as exception handlers, are particularly addressed. Consequently, the UML ADs in CSP specification can serve as a model for the soundness of the operational semantics and be acceptable by FDR for model checking.

The rest of this paper is structured as follows. The language of CSP and UML AD are introduced in section 2. The mappings from AD to CSP are given in section 3. Section 4 illustrates the tool of model checking and demonstrates a simple but non-trivial example. Section 5 briefly discusses the related work. The further work and conclusions are found in Section 6.

II. PRELIMINARIES

A. The langusge of Communicating Sequential Processes

CSP is a formal language for describing concurrent system whose components that are called processes. A process is the basic element of CSP and is defined by events which can be considered as interfaces through which it interacts with other processes and its environment by communication. All events in the interface of a process P form a set written as αP, is called its alphabet. If x is an event and P is a process, $x \rightarrow P$ is a process that describes an object which first engages in the event x and then behaves exactly as described by P. The following paragraphs briefly introduce the CSP operators used in this paper.

The process STOP is the deadlocked process which does nothing. SKIP, on the other hand, is a special process that does nothing but terminates successfully. √ is a special event in CSP that denotes the act of terminating successfully. More than one process may be synchronized to execute some events, and they can be represented by using a parallel operator ‖. For example, P ‖ Q denotes that the processes P and Q are executed in parallel. A set of processes runs concurrently, and it is written as $\| \; s : S \cdot P(s)$.

+This work is supported by National Natural Science Foundation of China(NSFC) under grant No. 60673115 and Shanghai Leading Academic Discipline Project, Project Number: J50103

978-0-7695-3641-5/09 $25.00 © 2009 IEEE
DOI 10.1109/ICIS.2009.107

B&P is a guarded expression where B is a Boolean guard such that process P will only be executed provided B is true. Choice operation is denoted as | in CSP. If x and y are distinct events, (x → P | y → Q) describes an object which initially engages in either of the events x or y. In general, if G is any set of events and P(x) is an expression defining a process for each different x in G, then (x : G → P(x)) defines a process which first offers a choice of any event y in G, and then behaves like P(y). Also, there are deterministic and nondeterministic choices in CSP. A deterministic choice $P \,\square\, Q$ is a notation for which the environment can control which of P and Q will be selected, provided that this control is exercised on the very first action. In contrast, a nondeterministic choice $P \sqcap Q$ behaves either like P or like Q, but unlike the deterministic choice, the environment cannot influence the way the choice is resolved. A deterministic choice over a set of processes is written as $\square\, s : S \cdot P(s)$ whereby the environment is given a set of processes to choose from. On the other hand, a nondeterministic choice over a set of processes is written as $\sqcap\ s : S \cdot P(s)$.

The sequential composition P;Q, is a process which first behaves as P but continues by behaving as Q immediately after P terminates successfully. Moreover, $P \triangle Q$ is another kind of sequential composition which does not depend on successful termination of P. Instead, the progress of P is just interrupted on occurrence of the first event of Q; and P is never resumed. There is a special kind of interrupt event which is called catastrophic event and is denoted as ς. $P \triangle (\varsigma \to Q)$ is a process which behaves like P up to catastrophe and thereafter like Q. Here Q is perhaps a process which is intended to effect a recovery after catastrophe. In addition, there are many other operators in CSP but not described here. A comprehensive description of the language is found in [5, 6].

B. Overview of UML 2.0 Activity Diagrams

In this section, the UML activity diagrams are firstly introduced and some definitions are then depicted.

Activity diagrams have been added to the UML rather late, and they are directed graphs, consisting of nodes and edges. An activity diagram may have multiple initial states, final states or flow finals. Flow final is the new element introduced in UML 2.0 which indicates that the process is terminated. The edges represent the occurring sequence of activities, objects involving the activity, including control flows, message flows and signal flows. Actions/Activities are denoted with round cornered boxes and joined by process flows or events. In addition, a decision node (branch) is shown as diamonds and can model divergent behavior that is based on a guard condition. Synchronization points called fork nodes are shown by multiple arrows entering or leaving the synchronization bar and may also be defined to illustrate how processing may be concurrently carried out, then synchronized at a join node before further activity is undertaken. Input and output can be shown in AD as parameters. This is done via rectangles which attach to the activities. Partitions allow the user to create views on the AD in which some areas of responsibility may be shown as well as the ones of organizational departments and the like. Moreover, the recent vision of the UML 2.0 has introduced the significant changes and additions into activities. Several new concepts and notations have been introduced, e.g., pins, exceptions, streams, expansion and interruptible activity regions, and so on. They, however, have always been poorly integrated, lacked expressiveness, and did not have an adequate semantics.

In order to conveniently depict some concepts and essential information of an activity diagram, some formal definitions are given as follows.

Definition 1 An activity diagram is a tuple AD = (N, E, $\mathbb{C}$, R), where

- $N = N_a \cup N_o \cup N_c$ is a finite set of activity nodes of the UML activity diagram, where $N_a = \{a \mid a \text{ is an acton node in ADs}\}$ is a finite set of action nodes and it includes send signal and receive signal nodes; $N_o = \{o \mid o \text{ is an object node in ADs}\}$ is a finite set of object nodes, and it consists of pins, data objects and data store nodes; $N_c = \{c \mid c \text{ is a control node in ADs}\}$ is a finite set of control nodes.
- $E = \{e \mid e \text{ is an edge in ADs}\}$ is a finite set of directed edges, and it includes control flow and object flow of the activity diagrams. So it can be denoted as $E = E_o \bigcup E_c$, where E_c is a finite set of control flows and E_o is a finite set of data flows. E_c and E_o are two disjoint sets, i.e. $E_c \bigcap E_o = \varnothing$. An edge can be associated with a guard, a weight and a name.
- $\mathbb{C}$ stands for graphical elements for containment in ADs, and it formally defined as a tuple $\mathbb{C}$ =(Activities, IR, EH, ER), where Activities are the specifications of parameterized behaviors as the coordinated sequencing of subordinate units whose individual elements are actions; IR is a finite set of interruptible regions; EH is a finite set of exception handlers; ER is a finite set of expansion regions.
- $R \subseteq (N \vee \mathbb{C}) \times E \times (N \vee \mathbb{C})$ is the flow relationship between the nodes or containments and flows.

Definition 2 The set of control node N_c can be partitioned into the following disjoint sets, and it is denoted as follows. $N_c = I \cup D \cup M \cup F \cup J \cup T$, where

- $I = \{i \mid i \text{ is } an \text{ initinal node in ADs}\}$ is a finite set of initial nodes, and an activity may have more than one initial node;
- $D = \{d \mid d \text{ is } a \text{ decision node in ADs}\}$ is a finite set of decisions/branches which are control nodes that choose between outgoing flows;
- $M = \{m \mid m \text{ is a merge node in ADs}\}$ is a finite set of merges;
- $F = \{f \mid f \text{ is a fork node in ADs}\}$ is a finite set of forks that split a flow into multiple concurrent flows;

- $J = \{j \mid j$ is a join node in ADs$\}$ is a finite set of joins;
- $T = \{t \mid t$ is a final node in ADs$\}$ is a finite set of final nodes, and it covers activity final and flow final nodes; So it can be denoted as $T = T_a \cup T_f$, where T_a is a finite set of activity final nodes and T_f is a finite set of flow final nodes. A flow final destroys all tokens that arrive at it. It has no effect on other flows in the activity, whereby an activity final node is a final node that stops all flows in an activity. An activity may have more than one activity final node. The first one reached stops all flows in the activity.

Definition 3 Let S_{in} denote the set of incoming edges for a node in ADs, then $S_{in}(n)$ = {e | e∈E, e is an incoming of n ∧ n∈N}.

Definition 4 Let S_{out} denote the set of outgoing edges for a node in ADs, then $S_{out}(n)$ = {e | e∈E, e is an outgoing of n ∧ n∈N}.

Definition 5 $\forall e \in E, n \in N$, if n is the target of e, then this relationship between e and n is denoted as $Tar(e) = n$. Similarly, if n is the source of e, denote them as $Src(e) = n$.

Definition 6 Let G denote the set of guard conditions in ADs and define a function: $E \xrightarrow{\rho} G$. if there is an edge e associated with a guard condition $g \in G$, then denote them as $\rho(e) = g$, otherwise, $\rho(e) = \varepsilon$.

III. MAPPINGS

A mapping from AD to CSP is defined as a function $\mathcal{H}_{AD} : n \rightarrowtail$ CSP, $n \in N \cup \mathbb{C}$ in the following subsections so as to derive the CSP descriptions of ADs. For simplicity, a new operator χ is defined below.

Definition 7 Given a node $n \in N$, $e \in S_{out}(n)$, then if $\rho(e) = \varepsilon$, $\chi(e) = \mathcal{H}(Tar(e))$ else if $\rho(e)$ is a Boolean expression, $\chi(n) = B \,\&\, \mathcal{H}(Tar(e))$ else $\chi(n) = \rho(e) \rightarrow \mathcal{H}(Tar(e))$

Due to a UML AD begins with a group of initial nodes, therefore $\mathcal{H}_{AD} : \text{AD} \rightarrowtail \text{CSP}$ can be instantiated as $\mathcal{H}_{AD} = \sqcap i : I.\mathcal{H}(i)$, where I is defined in Definition 3.2. Other mappings are presented in the following sections respectively.

A. Initial node

An initial node is a control node which is a starting point for executing an activity or structured node. If n is an initial node, then $|S_{out}(n)| = 1$ and $|S_{in}(n)| = 0$. A mapping from initial to CSP is given as below.

Criterion 1 (Initial Node) Given an initial node n, i.e. $n \in I$, if $e \in S_{out}(n) \wedge e \in E_c$, then $\mathcal{H}(n) = \chi(e)$.

B. Action node

An action is the fundamental unit of behavior specification and it represents a single step within an activity, that is, one that is not further decomposed within the activity. An action may have sets of incoming and outgoing activity edges that specify control flow and data flow from and to other nodes. The following criterion is to define a mapping from an action node to CSP specifications.

Criterion 2 (Action Node) Given an action node n, i.e., $n \in N_a$, $e \in S_{out}(n)$, then if $|S_{out}(n)| = 1$, $\mathcal{H}(n) = n \rightarrow \chi(e)$, else $\mathcal{H}(n) = n \rightarrow (e : S_{out}(n) \rightarrow \chi(e))$.

C. Decision node

A decision node is a control node that chooses between outgoing flows. It has one incoming edge and multiple outgoing activity edges. Most commonly, guards of the outgoing edges are evaluated to determine which edge should be traversed. Choice operator is therefore employed in the criterion below.

Criterion 3 (Decision Node) Given a decision node n, i.e., $n \in D$, $e \in S_{out}(n)$, so $|S_{out}(n)| > 1 \wedge |S_{in}(n)| = 1$, then $\mathcal{H}(n) = n \rightarrow (e : S_{out}(n) \rightarrow \chi(e))$.

D. Merge node

A merge node is a control node that brings together multiple alternate flows. It has multiple incoming edges and a single outgoing edge. The formalization for a merge node in CSP is therefore:

Criterion 4 (Merge Node) Given a merge node $n \in M$, $e \in S_{out}(n)$, **so** $|S_{out}(n)| = 1$, **then** $\mathcal{H}(n) = \chi(e)$.

E. Fork node

A fork node is a control node that splits a flow into multiple concurrent flows. It has one incoming edge and multiple outgoing edges, i.e.:

Criterion 5 (Fork Node) Given a fork node $n \in F$, $e \in S_{out}(n)$, **so** $|S_{out}(n)| > 1$, **then** $\mathcal{H}(n) = \| e : S_{out}(n) \cdot \chi(e)$.

F. Join node

A join node that has multiple incoming edges and one outgoing edge is a control node that synchronizes multiple flows. The fork-join pairs in the UML ADs represent concurrent flows and they can have more complicated structures in some complicated UML ADs. For instance, they may contain nested fork-join pairs, branches, as well as loops between the fork and the join. There are various generalized categories of fork-joins in [9]. For brevity, only simple fork-join (SFJ) is discussed here and a corresponding revised definition is given as follows.

Definition 7 A simple fork-join is a 4-tuple SFJ = *(N, E, p, j)* where N and E are defined in definition 3.1; $p \in P$ and $j \in J$ are the unique fork node and the unique join node in the SFJ, respectively; There are some limitations to SFJ: $\forall n \in N, |S_{out}(n)| = |S_{in}(n)| = 1$, which means that each node in

SFJ has one incoming and one outgoing exactly; $|S_{out}(p)| = |S_{in}(j)|$, which means that the number of incoming edges of j equals to the number of outgoing edges of p.

Note that the simple fork-join here may not cover all possible concurrent structures in ADs, it however would have encompassed most usage in practice, i.e.:

Criterion 6 (Join Node) Given a join node *j* in a SFJ, **then** $\mathcal{H}(j) = \chi(e)$.

G. Final node

A final node is an abstract control node at which a flow in an activity stops whereas a flow final node is a final node that terminates a flow. A flow final destroys all tokens that arrive at it. It has no effect on other flows in the activity. An activity final node is a final node that stops all flows in an activity. Therefore, they are mapped to STOP and SKIP in CSP respectively.

Criterion 7 (Final Node) Given a final node $n \in T$, **then,** if $n \in T_a$, $\mathcal{H}(n) = SKIP$, *else if* $n \in T_f$, $\mathcal{H}(n) = STOP$.

H. Interruptible Region

An interruptible activity region is an activity group that supports termination of tokens flowing in the portions of an activity. According to OMG specification [1], all tokens and behaviors in the region are terminated when a token leaves an interruptible region via edges designated by the region as interrupting edges. Furthermore, interrupting edges of a region must have their source node in the region and their target node outside the region in the same activity containing the region. It is mapped to catastrophe event ς in CSP as follows.

Criterion 8 (Interruptible Region) Let *e* is an interrupting edges in a IR, $n \in$ IR, *P* is a process that represent the IR activities, then $\mathcal{H}(n) = P \triangle (\varsigma \rightarrow \mathcal{H}(Tar(e)))\ ;\ (B \,\&\, \chi(e^{'}))$, *where* ς *means the* interrupting event occurs; *B* is a Boolean expression and it is false if the event ς occurs, otherwise, is true; e' is the non-interrupting edge of the region, and it has its source node in the region and its target node outside the region.

I. Exception Handler

An exception handler is an element that specifies a body to execute in case the specified exception occurs during the execution of the protected node. The exception handler defines the type of exception, and a behavior to execute when catching the particular exception. An exception is said to be thrown by the source of the error and caught when it is handled. So it can be depicted by a 3-tuple (protected node, Exception type, exception handler). For example, in the following figure, "Load Image" is a protected node; "Create Dump Image" is an exception handler while "File Not Found" is the type of the exception.

By the semantics, the mapping from exception handler to CSP is proposed below.

Criterion 9 (Exception Handler) Given an action node *n*, and *n* is a protected node, *e*, $e' \in S_{out}(n)$, **if** ***Tar*(*e*)** **is** an exception handler, **then** $\mathcal{H}(Tar(e)) = Tar(e)$. *B* is a Boolean expression and it is true if the exception occurs, otherwise, is false. So, $\mathcal{H}(n) = n \rightarrow ((B \,\&\, Tar(e) \rightarrow \chi(e')) \,\square\, \chi(e'))$.

These mapping have been illustrated by some simple but non-trivial examples in [16].

IV. SOFTWARE TOOLS FOR MODEL CHECKING

An activity diagram can be developed using many standard UML tools, such as Rational Rose, Poseidon or Visio, and ADs can be exported into XMI files directly or indirectly. FDR (Failures-Divergence Refinement) is a model-checking tool for state machines, with foundations in the theory of concurrency based around CSP—Hoare's Communicating Sequential Processes. A software tool for supporting to transfer an AD into CSP_M specification has been implemented according to the mapping functions described in this paper. The machine-readable dialect of CSP (CSP_M) is one result of a research effort with the primary aim of encouraging the creation of tools for CSP. FDR[12] was the first tool to utilize the dialect. Figure 1 shows a UML activity diagram to describe student enrollment process. The corresponding CSP_M specification of the AD is displayed in table 1. Due to the space constrains, the converting work is not depicted here in detail.

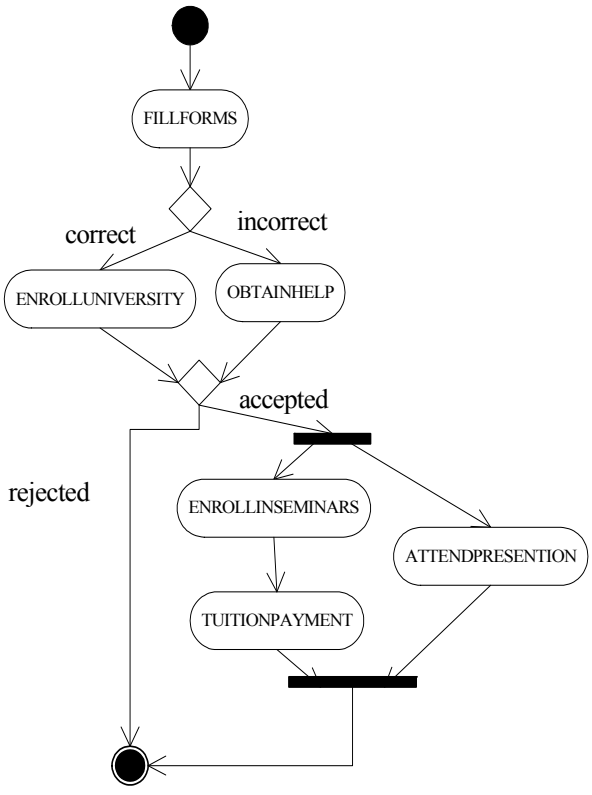

Figure 1. Example of an activity diagram

Machine-readable representation of the CSP language can be automatically checked with FDR2. The equivalence of the two diagrams can be verified by obtaining their CSP_M representation with the tools and checking in FDR2 that the two corresponding CSP processes refine each other. Moreover, FDR supports three kinds of refinement: Traces refinement, Failures-divergence refinement and Failures refinement. These checking approaches are shown in the screen-shot in Figure 2 together with some analysis data.

TABLE I. THE CSP_M SPECIFICAITON OF THE AD EXAMPLE

```
channel incorrect, correct, accepted, rejected
P1 = incorrect -> OBTAINHELP -> ENROLLUNIVERSITY
P2 = correct -> ENROLLUNIVERSITY
P0 = P1 [] P2
P3 = FILLFORMS -> P0
P4 = ATTENDPRESENTION [{}||{}] (ENROLLINSEMINARS ->
TUITIONPAYMENT)
P5 = accepted -> P4 -> SKIP
P6 = rejected -> SKIP
P7 = P5|~|P6
ENROLLMENT = P3 -> P7
assert P0 [T= P1
assert P0 [T= P2
assert ENROLLMENT [T= P3
```

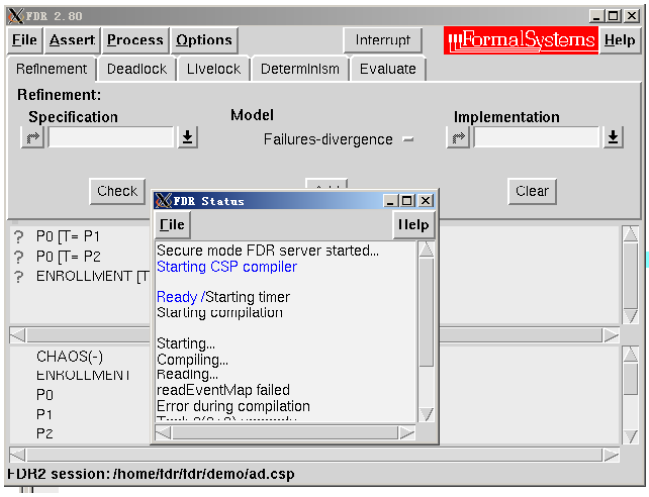

Figure 2. Screenshots of FDR2

V. RELATED WORK

Various attempts have been made to provide such a model checking on formal basis for UML artifacts. Most of them, however, focus on the formalization of UML State Diagrams [2, 3, 4, 10, 11] and so on. Among these attempts, the work reported in [2, 3, 4, 10] is formalizing UML State Diagrams in terms of Communicating Sequential Processes (CSP). UML state chart diagrams have been formalized in other formalisms such as pi-calculus [13], LOTOS [14]. Roscoe [15] also developed a compiler (translator) for translating a textual representation of state charts into CSP for model checking with FDR2. As far as we know, there is less work reported to give a formal semantic analysis to the UML ADs. [8] maps the basic elements of activity diagrams to Petri nets and discuss several syntactic and semantic questions concerning activities but it does support model checking automatically. Our previous work presented an approach to formalize the UML ADs in CSP[16]. However, three were a few mistakes in the paper and they are revised here.

Moreover, the later vision of the UML 2.0 has changed the ADs significantly, and ADs have no longer been a special kind of UML state machines. As far as we are aware, no work has been reported in model checking the UML ADs in CSP formally. Most previous work did not tackle UML 2.0 activity diagrams.

VI. CONCLUSION AND FURTHER WORK

An approach to formal and model check UML ADs by CSP and FDR respectively has been presented in this paper. Firstly, the language of CSP is introduced briefly. The UML ADs are then analyzed and some definitions are given in formal. Next, the mappings of the UML ADs to CSP have been illustrated with several small examples. The examples show the use of CSP in reasoning about the semantics of ADs by means of algebraic manipulation. Hence, the formalization provided in this paper actually complements the operational semantics of the UML ADs. CSP is also supported by the FDR2 model checking tool, which can automatically make some verifications to ADs. The formalisms and model checking are explicitly illustrated through a simple but non-trivial example.

The work presented in this paper has so far covered the formalization for initial nodes, final nodes, action nodes, decision-merge nodes, fork-join nodes, interrupting regions and exception handlers of the UML ADs. Due to the limitation of space, some concepts and notations have not been introduced in the formalization, e.g., pins, streams, expansion and object flow, and so on. Further work in this direction includes extending the mappings to cater for more features of the UML ADs. It is also desirable to develop the support tools and integrating them in the model driven architecture in the future.

REFERENCES

[1] OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2, OMG Document Number: formal/2007-11-02. Standard document URL: http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF.

[2] M.Y. Ng, M. Butler. Towards formalizing UML state diagrams in CSP, in: Proc. 1st IEEE International Conference on Software Engineering and Formal Methods, Brisbane, Australia, IEEE, 2003, pp. 138–147.

[3] W.L. Yeung, K.R.P.H. Leung, J. Wang, W. Dong, Improvements towards formalizing UML state diagrams in CSP, in: Proc. 12th Asia Pacific Software Engineering Conference, Taipei, December, 2005, pp. 176–184.

[4] W.L. Yeung, K.R.P.H. Leung, J. Wang, W. Dong, Modeling and model checking suspendible business processes via statechart diagrams and CSP. The Journal of Science of Computer Programming, Elsevier press, 65 (2007) 14–29.

[5] C.A.R. Hoare, Communicating Sequential Processes, Prentice Hall, 1985.

[6] A.W. Roscoe, The Theory and Practice of Concurrency, Prentice Hall, 1998.

[7] Formal Systems (Europe) Ltd. Failures-Divergence Refinement: FDR2 User Manual, 2003.

[8] Harald St¨orrle, Jan Hendrik Hausmann. Towards a Formal Semantics of UML 2.0 Activities.

[9] Dong Xu, H. Li and C. P. Lam, "Using Adaptive Agents to Automatically Generate Test Scenarios from the UML Activity Diagrams", Proc. of 12th APSEC 2005, IEEE Computer Society Press, Taipei, 2005.

[10] W.L. Yeung, Towards formalizing UML state diagrams with history in CSP, Tech. rep., Lingnan University. http://cptra.ln.edu.hk/~wlyeung/history.ps, 2005.

[11] W. Dong, J.Wang, X. Qi, Z.-C. Qi, Model checking UML statecharts, in: Proc. 8th Asia-Pacific Software Engineering Conference APSEC'01, IEEE, 2001.

[12] The Formal Systems Website, vendor for the FDR2. http://www.fsel.com/

[13] V.S.W. Lam, J. Padget, Formalization of UML statechart diagrams in the pi-calculus, in: Proc. 13th Australian Software Engineering Conference, 2001, pp. 213–223.

[14] B. Cheng, L. Campbell, E. Wang, Enabling automated analysis through the formalization of object-oriented modeling diagrams, in: Proceedings of IEEE International Conference on Dependable Systems and Networks, IEEE, 2000, pp. 305–314.

[15] B. Roscoe, Compiling statemate statecharts into CSP and verifying them using FDR—abstract. http://web.comlab.ox.ac.uk/-oucl/work/bill. roscoe/publications/94ab.ps, January 2003.

[16] Dong Xu, Nduwimfura Philbert, Zongtian Liu, at el. Towards Formalizing UML Activity Diagrams in CSP. Proc. 2008 International Symposium on Computer Science and Computational Technology. IEEE Computer Society Press. 2008.12. pp450-453.