Modelling Timed Concurrent Systems Using Activity Diagram Patterns*

Étienne André[1], Christine Choppy[1], and Thierry Noulamo[2]**

[1] Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, UMR 7030, F-93430, Villetaneuse, France
[2] University of Dschang, IUT Fotso Victor, LAIA, Bandjoun, Cameroun

Abstract. UML is the *de facto* standard for modelling concurrent systems in the industry. Activity diagrams allow designers to model workflows or business processes. Unfortunately, their informal semantics prevents the use of automated verification techniques. In this paper, we first propose activity diagram patterns for modelling timed concurrent systems; we then devise a modular mechanism to compose timed diagram fragments into a UML activity diagram that also allows for refinement, and we formalise the semantics of our patterns using time Petri nets.

Keywords: Unified Modelling Language, formal modelling, timed systems

1 Introduction

UML (Unified Modelling Language) [UML13] is the *de facto* standard for modelling concurrent systems in the industry. Activity diagrams allow designers to model workflows or business processes. Although UML diagrams are widely used, they suffer from some drawbacks. Indeed, since UML specification is documented in natural language, inconsistencies and ambiguities may arise. First, their rich syntax is quite permissive, and hence favours common mistakes by designers. Second, their informal semantics in natural language prevents the use of automated verification techniques, that could help detecting errors as early as the modelling phase. In this paper, we propose three contributions.

1. We propose activity diagram patterns for modelling timed concurrent systems using Timed Activity Diagram Components (TADCs),

* Work partially supported by STIC Asie project “CATS (Compositional Analysis of Timed Systems)”. This is the author (and slightly extended) version of the paper of the same name accepted for publication at the 6th International Conference on Knowledge and Systems Engineering (KSE 2014). The final version is available at www.springer.com.

** This work is supported by a France-Cameroon cooperation grant through the exchange program funded by the “Service de Coopération et d’Action Culturelle” (SCAC) from the French embassy in Cameroon.

2. we devise a modular mechanism to compose activity diagram fragments into a UML activity diagram that also allows for refinement, and
3. we formalise the semantics of our patterns using time Petri nets [Mer74].

Our approach guides the modeller task, and allows for automated verification using tools supporting time Petri nets such as TINA (Time Petri Net Analyzer) [BV06] or Roméo [LRST09].

Our choice of an extension of Petri nets has four advantages. First, the UML specification explicitly mentions Petri nets, and the informal semantics of activity diagrams is given in terms of token flows. Second, they feature a formal semantics [Mer74]. Third, they provide the designer with a graphical notation (close to that of activity diagrams), which helps to be convinced by the "validity" of our translation (although it cannot be formally expressed, due to the lack of formal semantics for activity diagrams). Last, several tools use extensions of Petri nets as input, and can perform efficient verification.

Outline In Section 2, we discuss related works. In Section 3, we informally recall UML activity diagrams and time Petri nets, and introduce our approach based on the notion of TADCs composed using inductive rules. In Section 4, we provide TADCs with a formal semantics expressed using a translation to time Petri nets. We use as a running example a coffee vending machine. We conclude in Section 5, and provide guidelines for future research.

2 Related Work

An important issue is to propose a formal semantics to UML diagrams using a formal notation, which is essential to allow for automated verification. This has been addressed in quite a variety of works using automata, different kinds of Petri nets, etc., so we mention only a few. Instantiable Petri nets are the target of transformation of activity diagrams in [KT10], and this is supported by tool BCC (Behavioural Consistency Checker). In [DSP11,BM07], the issue is performance evaluation, from activity diagrams and others (use case, state diagrams, etc.) to stochastic Petri nets. Also note that [GRR10] proposes an operational semantics of the activity diagrams (for UML 2.2). Börger [Bör07] and Cook *et al.* [CPM06] present other formalisations of the workflow patterns of [Wor] using formalisms different from Petri nets, viz. Abstract State Machines and Orc, respectively. In [MGT09], patterns for specifying the system correctness are defined using UML statecharts, and then translated into timed automata. The main differences with our approach are that the patterns of [MGT09] do not seem to be hierarchical: the "composition" of patterns in [MGT09] refers to the simultaneous verification of different properties in parallel.

In [ACR13], we introduced so-called "precise" activity diagram patterns to model business processes. We inductively defined a set of precise activity diagrams (as a subset of the syntactic constructs of the UML specification [UML13]), and translated them into coloured Petri nets [JK09]. Although this work shares with [ACR13] the definition of patterns and their translation into an extension

Element	[UML13]	[ACR13]	This work
Activities	Yes	Yes	Yes
Data	Limited	Yes	Limited
Participants	Limited	Yes	No
Initial / final nodes	Yes	Yes	Yes
Decision	Yes	Restricted	Yes
Merge	Yes	Restricted	Yes
Fork	Yes	Restricted	Yes
Join	Yes	Restricted	Yes
Timed transitions	Limited	No	Yes

Table 1. Summary of the syntactic aspects considered

of Petri nets, this work is orthogonal to [ACR13] in the following sense: first, it extends the UML specification with timed constructs, which were not considered at all in [ACR13]. Second, and most importantly, the patterns proposed here are much less restrictive and give more freedom to the designer: the TADCs we define here allow for an arbitrary number of input and output connectors, whereas [ACR13] requires at most one of each. In contrast to [ACR13], we do not restrict the use of the syntactic constructs (in [ACR13], we required that, e.g. each fork is eventually followed by a join). We do not claim that the modular mechanism proposed in this work is better or more useful than [ACR13]: it can be used for a different purpose, when slightly less restrictions are needed. Finally, the scheme that we propose here allows to *refine* TADCs by replacing them with other TADCs, that can be "plugged" into a higher-level one, as long as the connectors are the same. In contrast, due to the presence of at most one input and one output connectors, the patterns of [ACR13] hardly allow this. We give in Table 1 a comparison of the syntactic elements from the specification [UML13] taken into account in [ACR13] and/or in this work. Note that an element not taken into account in one of these two works does not necessarily mean that that work is less interesting; recall that a main difference between [ACR13] and this work is that [ACR13] is more restrictive in terms of syntax, which may also help to avoid mistakes, whereas this work is more permissive. (We only consider in Table 1 elements taken into account in at least [ACR13] or in this work.)

3 Modelling Timed Systems Using Activity Diagrams

3.1 Preliminaries: Activity Diagrams

We briefly recall here UML activity diagrams [UML13], and use the coffee vending machine in Fig. 1 to introduce the syntax. First, activity diagrams feature global variables (that are mentioned several times in the UML specification, and used in the examples). In our setting, we require the global variables to be finite domain. This includes Booleans, bounded integers, enumerated types, and possibly more evolved structures such as finite tuples or lists. Such finite domain variables are often met in tools supporting Petri nets and their extensions.

Fig. 1 features four variables, viz. `Prod` (of enumerated type {`TEA`, `COFFEE`}), `avail` (Boolean), `state` (of enumerated type {`on`, `serving`, `stand by`}), and `w_state` (of enumerated type {`water_ok`, `water_lack`}).

Activity diagrams feature activities (all rounded rectangles in Fig. 1). These activities can involve global variables modifications, either by assigning a value (or the result of a predefined function) to a global variable, or by calling a function with side-effects on several global variables. In our setting, we require this modification to be discrete (i.e. instantaneous). For example, in Fig. 1, the action associated with the `Choice` activity assigns the result of function `P_button()` to the global variable `Prod`.

Activity diagrams also feature an *initial node* (e.g. the upper node in Fig. 1), and two kinds of final nodes, viz. *activity final* that terminates the activity globally, and *flow final* that terminates the local flow (Fig. 1 features no final node). Activity diagrams feature *decision nodes* (e.g. the `ChooseProduct` node in the middle of Fig. 1), i.e. depending on guards, one path among others is taken; they feature *merge nodes* (e.g. the bottom-most node in Fig. 1), that is the converse. They also feature *fork nodes*, that split the flow into different subactivities executed in parallel, and *join nodes*, the converse operation.

3.2 Timed Activity Diagram Components

We define here Timed Activity Diagram Components (TADCs), obtained by composition of basic activity diagrams fragments using inference rules (that will be given in Section 3.3). As shown in Fig. 2(a), a TADC has in general $n \in \mathbb{N}$ input connectors, and $m \in \mathbb{N}$ output connectors. These connectors will be used by the inductive rules to build more complex TADCs, as well as to perform some refinement.

TADCs have three main purposes:

1. define "well-formed" activity diagrams, by both restricting the set of syntactic constructs available in the specification, and augmenting it with some timed constructs,
2. allow a translation to another formalism using an inductive mechanism, and
3. allow a modular specification with possible refinement, or the definition of "black-box" components.

As an example, we give in Fig. 2(b) an example of a TADC responsible for the choice of the drink in a coffee vending machine. This TADC has 1 input connector (i.e. `E_Choice`) and 2 output connectors (i.e. `C_TEA_water` and `C_COFFEE_water`). This TADC can be refined into another one with a different structure (e.g. a more complex one that could prepare some sugar too), that can then still be plugged into the coffee vending machine, as long as it still has 1 input and 2 output connectors. It is also possible to hide the internal specification of this TADC, by only giving its input and output connectors.

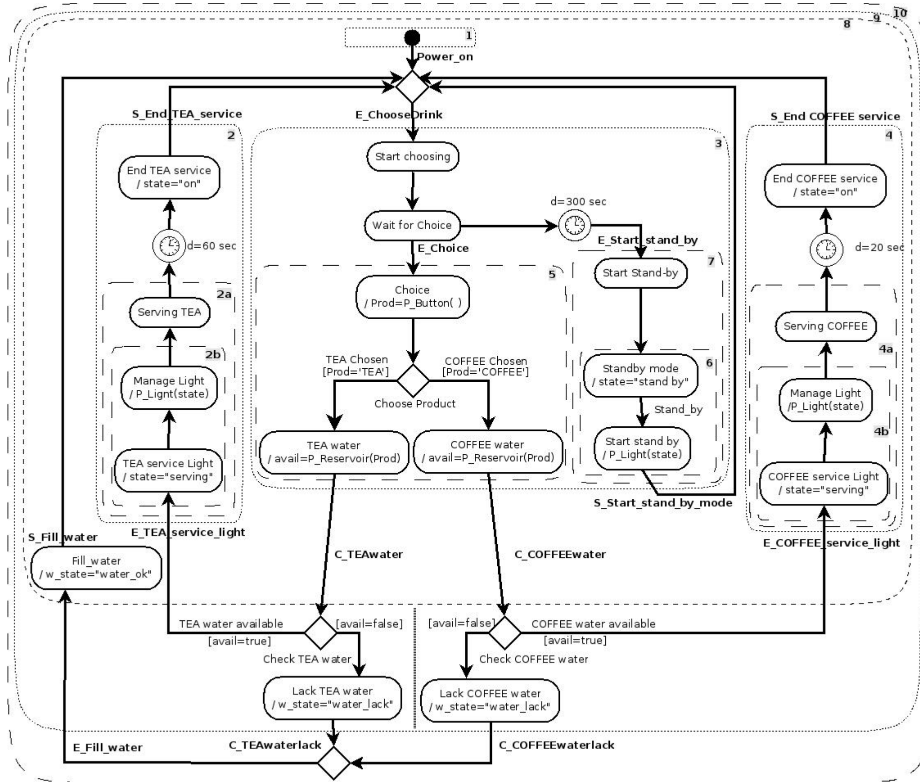


Fig. 1. Specification of a coffee vending machine using TADCs

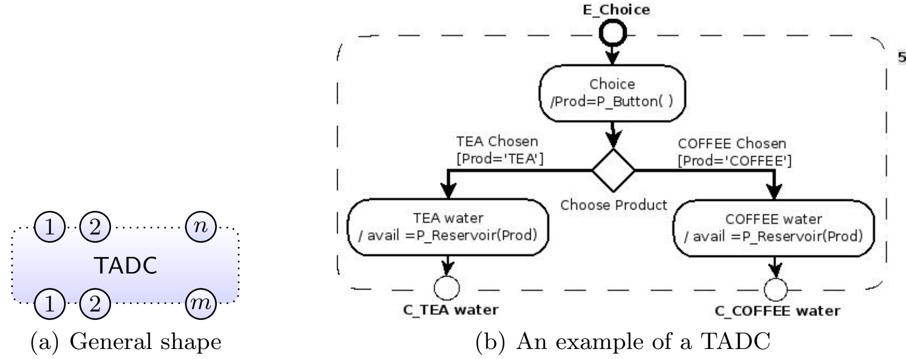
3.3 TADC Patterns

We introduce in Table 2 our patterns for inductively creating and composing TADCs. We define 3 basic patterns (1–3), that define TADCs from single activity diagrams syntactic constructs, and 7 composite patterns (4–10), that define TADCs by combining other TADCs together with syntactic constructs (usually transitions or choice nodes). We only depict the connector used in the compositions; recall that each TADC can have an arbitrary number of input and output connectors. (The last column, introduced in Table 2 to save some space, will be used in Section 4.)

Pattern 1 (initial node) This pattern is made of the initial node. It has no input connector, and one output connector, which is itself.

Pattern 2 (flow final node) This pattern is made of the flow final node.[3] It has one input connector, which is itself, and no output connector.

[3] For sake of conciseness, we do not include the activity final node. It could be added to our patterns using the same translation as in [ACR13], i.e. using a "global Boolean variable" that is tested to be true in all guards, and set to false when the activity final



(a) General shape

(b) An example of a TADC

Fig. 2. TADC: general shape and example

Pattern 3 (simple activity) This pattern is made of a simple activity. It has one input connector, which is itself, and one output connector, which is itself too. Recall that actions can modify the global variables, by either assigning a value (or the result of a predefined function) to a global variable, or using a function with side-effects on several global variables.

Pattern 4 (sequence) This pattern combines two TADCs with a sequence transition. The input (resp. output) connectors of the resulting TADC are all input (resp. output) connectors of TADC_1 and TADC_2, except the output (resp. input) connector of TADC_1 (resp. TADC_2) connected by the sequence transition. In other words, any free input (resp. output) connector of one of the component TADCs will remain a free input (resp. output) connector of the resulting composite TADC. The mechanism will be the same for all subsequent patterns.

Pattern 5 (deterministic delay) This pattern combines two TADCs with a deterministic delay $d \in \mathbb{R}_{\geq 0}$. The lower TADC will start exactly d units of time after the upper TADC has completed its activity. Again, the input (resp. output) connectors of the resulting TADC are all input (resp. output) connectors of TADC_1 and TADC_2, except the output (resp. input) connector of TADC_1 (resp. TADC_2) connected by the delay transition.

Pattern 6 (non-deterministic delay) This pattern generalises the previous pattern to a non-deterministic delay, i.e. comprised in an interval $[0, d]$.

Pattern 7 (deadline) In this pattern, after TADC_1 completes its execution, activity A is executed. Then, TADC_2 starts after at most d units of time; alternatively, TADC_3 starts after exactly d units of time.

Pattern 8 (decision) This pattern reuses the syntactic "decision" node from the UML specification. After TADC completes its execution, the conditions $\mathrm{cond}_1, \dots, \mathrm{cond}_n$ are evaluated, and the TADC with a true guard is executed. The conditions cond_i (for $1 \leq i \leq n$) are Boolean expressions over the activity

node is reached. This mechanism ensures that concurrent activities do not progress any more once an activity final node is reached.

diagram's variables. If several conditions are true simultaneously, a destination TADC is selected non-deterministically among those with a true condition. In contrast to [ACR13], we do not require that at least one guard is true; this however can result in ill-formed models.

Pattern 9 (merge) This pattern reuses the syntactic "merge" node from the UML specification. After some TADC_i (for $1 \leq i \leq n$) completes its execution, then TADC starts executing.

Pattern 10 (synchronisation) This pattern merges the syntactic "fork" and "join" nodes from the UML specification. After all TADC_i (for $1 \leq i \leq n$) complete their execution, all TADC_j (for $1 \leq j \leq m$) start executing simultaneously. Note that the classical UML fork (resp. join) from [UML13] can be obtained from this pattern by setting $n = 1$ (resp. $m = 1$).

3.4 Example: A Coffee Vending Machine

The coffee vending machine in Fig. 1 has been built using the inductive rules of Table 2. For example, all activities can be defined by applying Rule 3 ("activity"). The result of the applications of composite rules are outlined using dashed boxes in Fig. 1. For example, the application of Rule 8 ("decision") to the three activities in the centre of Fig. 1 gives the TADC number 5. By applying Rule 7 ("deadline") to this TADC, to its right-hand TADC (number 7), to the `wait for choice` activity and to the `Start choosing` activity considered as a TADC, we get the (large) TADC number 3. The rest of the coffee vending machine is obtained similarly.

4 Translation into Time Petri Nets

4.1 Time Petri Nets With Global Variables

Time Petri nets (TPNs) [Mer74] are a kind of automaton represented by a bipartite graph with two kinds of nodes, places (e.g. p_1 in Fig. 3(a)) and transitions (the rectangle nodes in Fig. 3(a)). In time Petri nets, transitions are associated with a firing interval: for example, in Fig. 3(a), the transition between p_1 and p_2 (say, t_2) can fire between 1 and 3 time units after a token is present in p_1, and the transition between p_1 and p_2 (say, t_3) can fire between 2 and 4 time units after a token is present in p_1. We use the strong semantics where time elapsing cannot disable a transition: that is, transitions must fire at the end of their firing interval, unless the transition becomes disabled by another transition in the meanwhile.

We extend the usual TPNs with global variables as follows. We assume a set of finite domain variables over a set of types. For example, Fig. 3(a) makes use of b (of type Boolean) and i (of type integer bounded by some predefined constant). These global variables can be tested in guards (e.g. $\{\neg b\}$ in Fig. 3(a)), and updated when firing transitions (e.g. $i := i + 1$ in Fig. 3(a)). Note that we

#	Pattern	Syntax	Translation
1	Initial node		
2	Final node		
3	Simple activity	A /action	action
4	Sequence	TADC_1 TADC_2	$Tr(\mathsf{TADC}_1)$ $Tr(\mathsf{TADC}_2)$
5	Deterministic delay	TADC_1 d TADC_2	$Tr(\mathsf{TADC}_1)$ $[d, d]$ $Tr(\mathsf{TADC}_2)$
6	Non-deterministic delay	TADC_1 $[0, d]$ TADC_2	$Tr(\mathsf{TADC}_1)$ $[0, d]$ $Tr(\mathsf{TADC}_2)$
7	Deadline	TADC_1 A d TADC_2 TADC_3	$Tr(\mathsf{TADC}_1)$ $[0, d]$ $[d, d]$ $Tr(\mathsf{TADC}_2)$ $Tr(\mathsf{TADC}_3)$
8	Decision	TADC $[\mathrm{cond}_1]$ $[\mathrm{cond}_n]$ TADC_1 ··· TADC_n	$Tr(\mathsf{TADC})$ $[\mathrm{cond}_1]$ $[\mathrm{cond}_n]$ $Tr(\mathsf{TADC}_1)$ ··· $Tr(\mathsf{TADC}_n)$
9	Merge	TADC_1 ··· TADC_n TADC	$Tr(\mathsf{TADC}_1)$ ··· $Tr(\mathsf{TADC}_n)$ $Tr(\mathsf{TADC})$
10	Synchronisation	TADC_1 ··· TADC_n TADC'_1 ··· TADC'_m	$Tr(\mathsf{TADC}_1)$ ··· $Tr(\mathsf{TADC}_n)$ $Tr(\mathsf{TADC}'_1)$ ··· $Tr(\mathsf{TADC}'_m)$

Table 2. Activity diagram patterns

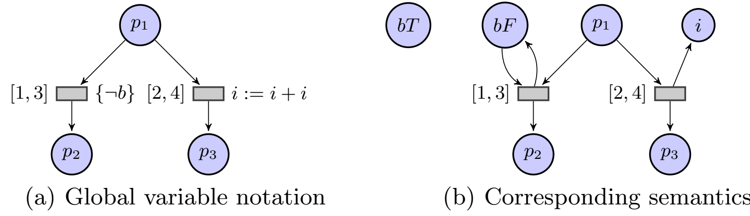


Fig. 3. An example of a time Petri net with global variables

depict guards within braces (e.g. $\{\neg b\}$ in Fig. 3(a)) to differentiate with firing times depicted within brackets (e.g. $[1, 3]$).

This extension of the usual formalism to finite domain variables is only syntactic, i.e. this formalism does not add expressiveness to usual time Petri nets. Each finite domain variable can be encoded into a finite set of places. For example, a Boolean variable (b in Fig. 3(a)) could be encoded into 2 places bT and bF, that encode the fact that b is true (or false, respectively) if it contains a token. Then, testing whether b if false is equivalent to checking the presence of a token in bF. For integers, an option in some situations is to encode the value of the integer with a number of tokens in a dedicated place; then adding one to the integer is encoded by adding one token to the dedicated place. (In general, the translation may require n places for an integer bounded by n.) These two constructions are depicted in Fig. 3(b). It is clear that using such global variables makes the resulting net much more compact and readable.

A formal definition of TPNs with global variables is available in Appendix A.

Remark 1. Our definition of time Petri nets with global variables is not very far from coloured Petri nets [JK09], where types, guards and assignments are also defined. A major difference is that coloured Petri nets feature *coloured* tokens, whereas we use here standard, "null-typed" tokens. A second major difference is that, of course, our definition features time too.

4.2 Translation Mechanism

We now explain how to translate the activity diagrams defined in Section 3 into time Petri nets with global variables. The inductive definition of our TADCs (following the rules in Table 2) makes it easy to define an inductive translation.

General Scheme Recall that each TADC has a set of input and output connectors, that can be used to compose the TADCs in an inductive manner. Here, we translate each TADC into a TPN fragment where the connectors are translated into places. Hence, two TPN fragments can be composed by fusing the corresponding connector places together.

We give the translation of each pattern in the last column of Table 2. In the following, we explain the translation of each pattern.

Pattern 1 (initial node) The translation of the initial node is a simple place, that contains a token. At the beginning of the execution of the translated TPN, only these places encoding the initial nodes contain tokens.

Pattern 2 (final node) The translation of the final node is a simple place.

Pattern 3 (simple activity) The translation of an activity is a TPN transition, preceded and followed by a place, so as to connect in a proper manner with the other translated TADCs. The assignment of variables is easily translated to an assignment on the TPN transition. Note that the functions involving a user input (e.g. `P_Button()` in Fig. 1) are translated into a non-deterministic choice in the resulting TPN. Hence, the verification will consider all possible choices.

Pattern 4 (sequence) The translation of the sequence pattern is obtained by recursively translating each of the two TADCs, and then by fusing the output place of the upper translated TADC with the input place of the lower TADC.

Pattern 5 (deterministic delay) The translation of this pattern is obtained as follows: the upper TADC is connected to a TPN timed transition that can fire exactly d units of time after it was enabled, i.e. after a token was present in the output place of the upper TADC. Then, after the transition fires, the token moves to the translation of the lower TADC.

Pattern 6 (non-deterministic delay) This pattern translates the same as the previous pattern, with the exception that the transition can fire any time between 0 to d units of time.

Pattern 7 (deadline) First, the upper TADC is translated. Then, it is connected to the TPN transition modelling activity A. This transition is then connected to a place. When a token enters this place, both outgoing transitions are enabled. The left-hand transition can fire any time between 0 to d units of time after it is enabled; if it does not, then the right-hand transition must fire exactly d units of time after it is enabled, due to the TPN strong semantics we use.

Pattern 8 (decision) The translation of this pattern is straightforward: the output place of the translation of the upper TADC is connected to a set of transitions that have the same guards as the initial TADCs, and then lead to the translation of these TADCs. The semantics of TPNs is the same as for the UML, i.e. if several guards are true, then one is non-deterministically chosen.

Pattern 9 (merge) The translation of this pattern is straightforward and is the converse of the previous pattern.

Pattern 10 (synchronisation) The translation of this pattern is again straightforward: once the n upper TADCs finish their execution, their corresponding token is consumed by the TPN transition; then m fresh tokens are created, and the m lower TADCs start executing concurrently.

Initial Marking The initial marking assigns one token to each place corresponding to an initial node. The initial value of the variables can be set to the initial value of the variables in the activity diagram, if any such value is defined, or to a predefined standard value otherwise (e.g. 0 for integers, true for Booleans, etc.).

4.3 Application to the Coffee Vending Machine

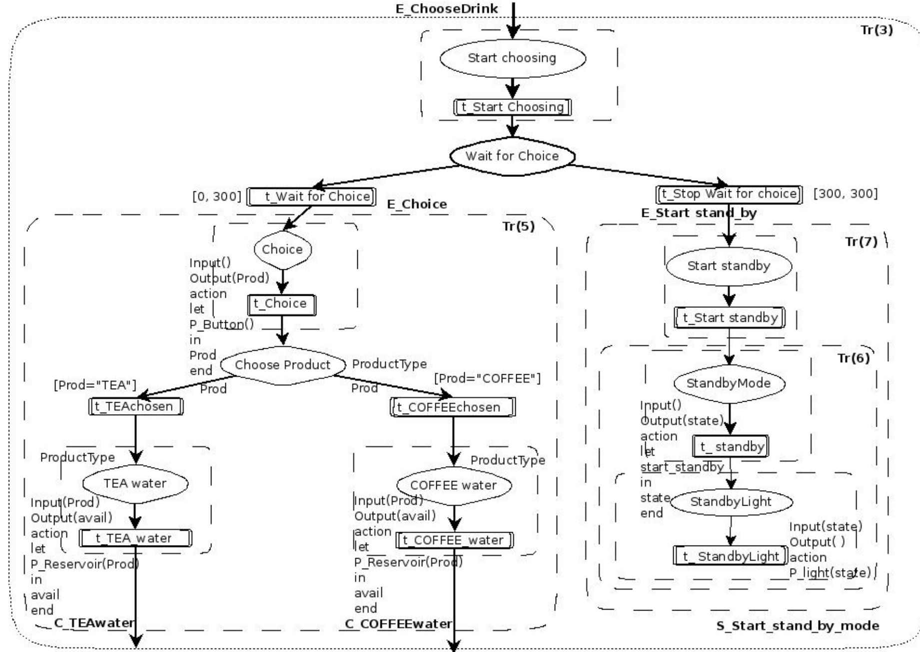


Fig. 4. Translation of a fragment of the coffee machine into a TPN

We give in Fig. 4 the translation into a TPN of the TADC fragment #3 of Fig. 1 (that corresponds to the application of the deadline pattern to TADCs 5 and 7). We show with dashed lines the patterns, and write in their top-right corner their number. Observe that this TPN is indeed timed (e.g. firing interval $[0, 300]$), and that it features variables: for example, variable `Prod` is checked in some guards (e.g. `Prod` = `TEA` in transition `t_TEAchosen`), read in side-effect functions (e.g. `P_Reservoir`(`Prod`)), and assigned to the result of a function (e.g. `P_Button`()).

Once the TPN corresponding to the whole TADC of Fig. 1 has been input to a model checker, one can formally verify properties mixing time and value of the global variables, e.g. "if initially `w_state = water_ok`, then a drink may eventually be delivered within 400 time units". This could be done using tools supporting time Petri nets, such as TINA [BV06] or Roméo [LRST09].

We give the translation of the coffee vending machine of Fig. 1 in Fig. 5 in Appendix B. Observe that the only token is in the top-most place encoding the initial node.

5 Conclusion and Perspectives

In this work, we introduce Timed Activity Diagram Components (TADCs) that help designers to devise complex activity diagrams by the inductive application of predefined patterns. This mechanism guides the designer in the modelling process, and allows one to define black-box components or to replace components by other components. Furthermore, a translation to an extension of time Petri nets allows for formal verification.

Future Work We first discuss the extension of our patterns. Our timed patterns define a minimal set of timing constructs; they could be further enriched with more complex features, such as timed synchronisation between activities, or deadlines on TADCs instead of on simple activities. Our notion of refinement is only syntactic; for example, the TADC in Fig. 2(b), that checks the presence of water in the reservoir, could be replaced with a TADC that does not. Ensuring semantic guarantees (e.g. "the presence of the water in the reservoir has been checked when exiting the TADC") is a challenging future work, that could be handled using interface constraints to be satisfied on the TADC's variables.

Another challenging future work is to formally compare the semantics given in terms of translation to time Petri nets with a formal semantics given to activity diagrams (e.g. [GRR10], with the problem that it does not consider time).

Our translation has not been automated yet, but there is no theoretical obstacle: each pattern can be directly translated to a TPN fragment using our inductive rules. This is the subject of future work. Once an automated translation tool is implemented, it will also be possible to "test" our translation mechanism, i.e. to check for large input models that the resulting behaviour (in terms of time Petri nets) is compatible with what is expected from the informal semantics of [UML13].

References

ACR13. Étienne André, Christine Choppy, and Gianna Reggio. Activity diagrams patterns for modeling business processes. In *SERA*, volume 496 of *Studies in Computational Intelligence*, pages 197–213. Springer, 2013. 2, 3, 5, 7

BM07. Simona Bernardi and José Merseguer. Performance evaluation of UML design with stochastic well-formed nets. *Journal of Systems and Software*, 80(11):1843–1865, 2007. 2

Bör07. Egon Börger. Modeling workflow patterns from first principles. In *ER*, volume 4801 of *Lecture Notes of Computer Science*, pages 1–20. Springer, 2007. 2

BV06. Bernard Berthomieu and François Vernadat. Time Petri nets analysis with TINA. In *QEST*, pages 123–124. IEEE Computer Society, 2006. 2, 11

CPM06. William R. Cook, Sourabh Patwardhan, and Jayadev Misra. Workflow patterns in Orc. In *COORDINATION*, volume 4038 of *Lecture Notes of Computer Science*, pages 82–96. Springer, 2006. 2

DSP11. Salvatore Distefano, Marco Scarpa, and Antonio Puliafito. From UML to Petri nets: The PCM-based methodology. *IEEE Transaction on Software Engineering*, 37(1):65–79, 2011. 2

GRR10. Hans Grönniger, Dirk Reiss, and Bernhard Rumpe. Towards a semantics of activity diagrams with semantic variation points. In *MoDELS*, volume 6394 of *Lecture Notes of Computer Science*, pages 331–345. Springer, 2010. 2, 12

JK09. Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009. 2, 9

KT10. Fabrice Kordon and Yann Thierry-Mieg. Experiences in model driven verification of behavior with UML. In *Monterey Workshop*, volume 6028 of *Lecture Notes of Computer Science*, pages 181–200. Springer, 2010. 2

LRST09. Didier Lime, Olivier H. Roux, Charlotte Seidner, and Louis-Marie Traonouez. Romeo: A parametric model-checker for Petri nets with stopwatches. In *TACAS*, volume 5505 of *Lecture Notes of Computer Science*, pages 54–57. Springer, 2009. 2, 11

Mer74. Philip Meir Merlin. *A study of the recoverability of computing systems*. PhD thesis, University of California, Irvine, CA, USA, 1974. 2, 7

MGT09. Ahmed Mekki, Mohamed Ghazel, and Armand Toguyeni. Validating time-constrained systems using UML statecharts patterns and timed automata observers. In *VECoS*, pages 112–124. British Computer Society, 2009. 2

UML13. OMG unified modeling language. version 2.5 beta 2, 2013-09-05. http://www.omg.org/spec/UML/2.5/Beta2/PDF/, 2013. 1, 2, 3, 7, 12

Wor. Workflow Patterns Initiative. Workflow patterns home page. http://www.workflowpatterns.com. 2

Appendix

A Formal Definition of Time Petri Nets

Definition 1 (TPN). *A time Petri net (TPN) is a tuple* $\mathcal{N} = \langle P, T, \mathcal{D}, \mathcal{V}, {}^\bullet(.), (.)^\bullet, I_s, G, A, M_0 \rangle$ *where*

- P *(resp.* T*) is a non-empty finite set of* places *(resp.* transitions*),*
- $\mathcal{D}$ *is a finite set of* finite domains *(or types),*
- $\mathcal{V}$ *is a finite set of variables whose type is in* $\mathcal{D}$*,*
- ${}^\bullet(.)$ *(resp.* $(.)^\bullet$*)* $\in (\mathbb{N}^P)^T$ *is the* backward *(resp.* forward*)* incidence function*,*
- $I_s \in \mathcal{I}^T$ *is the function that associates a* firing interval *with each transition,*
- $G \in T \rightarrow \mathcal{B}(\mathcal{V})$ *is the guard associating with each transition a Boolean expression over* $\mathcal{V}$*,*
- A *is a function that associates with each transition an assignment of each of the variables to a value in its domain, and*
- M_0 *is the* initial marking *that associates with each place an integer number of tokens, and with each of the variables a value in its domain.*

B Translation of the Coffee Machine

We give below the translation of the coffee machine following the rules of Table 2.

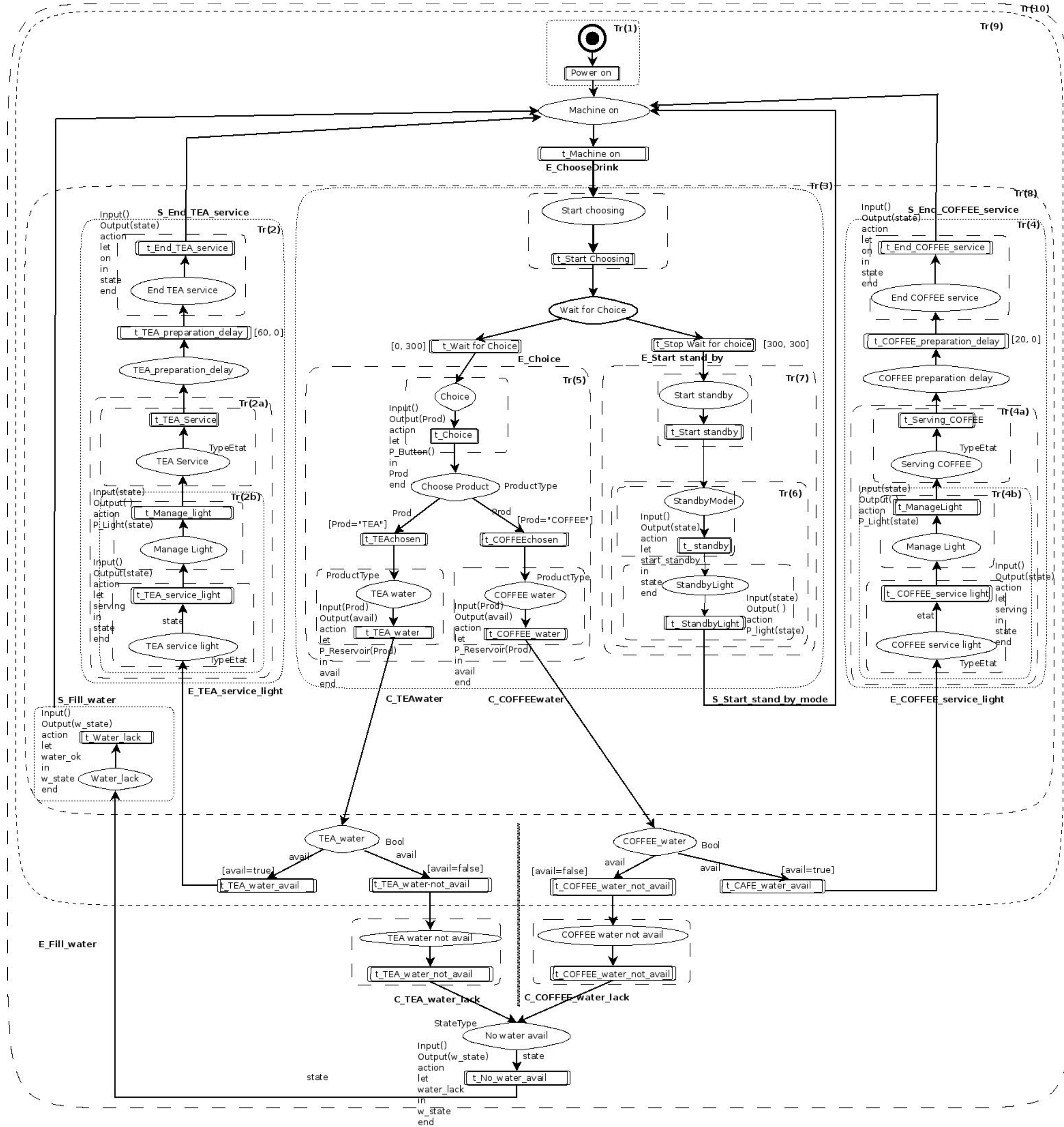


Fig. 5. Translation of the coffee vending machine into a time Petri net