Requirements Specification for Controller Design – from Use Cases to IOPT Net Models

João Paulo Barros*, Isabel Sofia Brito†, and Luís Gomes‡
* Instituto Politécnico de Beja - ESTIG & UNINOVA - CTS, Portugal
Email: jpb@uninova.pt
† Instituto Politécnico de Beja - ESTIG, Portugal
Email: isabel.sofia@ipbeja.pt
‡ Universidade Nova de Lisboa - Faculdade de Ciências e Tecnologia & UNINOVA - CTS, Portugal
Email: lugo@uninova.pt

***Abstract*—Non-autonomous Petri nets offer a language especially adapted for controller specifications. They are typically used in the design phase often with no clear connection to the analysis phase, including requirements specification. This paper shows how use cases can be used to support requirements specification amenable to a direct transformation to IOPT nets, a class of non-autonomous Petri nets. To that end, we propose a set of semi-formal rules for use case descriptions, including use case relationships, which take advantage of the concepts available in IOPT nets, namely input and output signals and events and net addition, a net composition operation.**

I. Introduction

Petri nets is a name used to designate a large set of languages. Presently, and especially since the 1980s there is a large number of Petri nets classes, many with supporting tools [1].

One of the areas where Petri nets are commonly used is controller design often using non-autonomous extensions (e.g. [2], [3]). To that end, Petri nets are seen as a design tool amenable to model specification, simulation, and verification. Sometimes, those models are used as a basis for code generation. Often, the analysis phase appears absent: the process towards code generation seems to start at the design phase. Naturally, this is not the case as some form of analysis has to be made. Yet, this is mixed with design decisions in an informal and unstructured way which is not properly documented. Here, we propose the use of use cases (e.g. [4], [5], [6]) for the analysis phase, namely for requirements specification.

The use case approach is a requirements technique frequently used in industrial applications. The use case approach is relatively easy to describe, understand, and trace. Unfortunately, it lacks formal syntax and semantics, needed to help verification and validation as well as the tracing to the design phase of the system development (e.g. [7]).

Controller models must be precise as they are often integrated in critical systems. Hence, we propose a set of semiformal rules that stand between use cases and Petri net models. In this way, we manage to overcome the limitations caused by the informal aspects of the use case approach, as use cases can be unambiguously translated to Petri nets, namely to IOPT nets a class of non-autonomous Petri nets used for controller specification that is formally presented elsewhere [8]. This class of Petri nets is currently supported by a set of development tools, presently at the level of design, verification, and implementation phases [9], [10]. The proposed use of use cases complements those publicly available tools[1] with an integrated approach for the analysis phase. In summary, the present paper shows how to integrate use cases with IOPT nets using a set of semiformal rules that allow an unambiguous translation from use cases to net models.

The paper is structured as follows: Section II presents IOPT nets in the context of controller design, as well as the net addition operation. Section III briefly presents the use case approach and the respective template. Section IV shows how to translate the use cases to Petri nets including relationships between use cases. Finally, Section V discusses the related work and Section VI concludes with some notes about future work.

II. Controller design and IOPT nets

The class of IOPT nets [8] was specifically designed to model synchronous, event-driven controllers. The controller interacts with the environment through input and output *signals* and *events*. *Signals* can take any value from a known domain (here we assume a Boolean domain), and *events* are Boolean values than become true when a change in the respective associated signal has happened.

The controller runs according to sequential *steps*, each one with the following procedure:

1) Input signal values are acquired and stored in a buffer (top left of Fig. 1);
2) Events are computed according to the signal values in step 1 and the known previous values; they are stored in another buffer (top right of Fig. 1);
3) The IOPT net fires using a maximal step semantics (all transitions that can fire are fired); these firings can change the values in output signals (changes related to state modification); these changes are specified by output events in transitions (bottom left of Fig. 1);

[1] http://gres.uninova.pt/IOPT-Tools

978-1-4799-4905-2/14/$31.00 ©2014 IEEE

4) Output actions rules, associated to places, can also change the output signal values (bottom right in Fig. 1);
5) Go back to 1) to execute the next step.

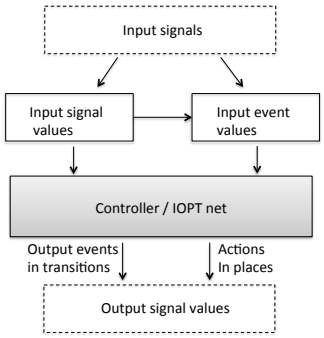


Fig. 1. Controller and environment.

A. IOPT Nets

IOPT nets are based on Place/Transition (P/T) nets (e.g. [11], [12]). Besides marked places, transitions, and weighted directed arcs, as in P/T nets, IOPT nets add the following constructs:

- Each transition can have one or more of the following:
  - Input event — this is a necessary, but not sufficient condition for the transition to fire; in the net models, we use "?" as a prefix for input event names; we also use a "-" and "+" suffix to specify events that are true when the signal decreases or increases, respectively, its value;
  - Guard — Boolean expression that uses the input signals as variables; we use "?" as a prefix for input signal names;
  - Output event — changes the value of an output signal, when the transition fires; in the net models, we use "!" as a prefix for output event names; we also use a "+" or "-" suffix to specify events that set or reset the associated output signal;
  - Priority — used in the case of structural conflicts to resolve effective conflicts.
- Each place can have one or more output action rules — After transition firing (in step 4 above), if the place is marked, an output signal value is changed; we use "!" as a prefix for output signal names.

Regarding its semantics, IOPT nets use a maximal step semantics: in each step, all *enabled* and *ready* transitions, considering effective conflicts, are fired. As already mentioned, effective conflicts are decided based on priorities in transitions. A transition is *enabled* if it has enough tokens in its input places — the usual condition for P/T nets. A transition is *ready* if all its guards and input events are true. A more complete and formal presentation of IOPT nets syntax and semantics can be found elsewhere [8].

A common problem with graphical languages as Petri nets is model dimension. To that end many structured approaches have been proposed for Petri nets (e.g. [13]). Next, we briefly present a simple operation for net composition, here used for IOPT nets.

B. Net addition

Petri net composition is typically based on place fusion, transition fusion, or both. Net addition, defined elsewhere [14], is a structured and flexible operation for net composition. It has two stages: (1) in the first, the operand nets are unified ("put together"), hence still disjoint; (2) next they are merged by fusing the specified nodes: transitions, places, or both.

Fig. 2 illustrates a simple case of net addition where places $p2$ and $p3$ and also transitions $t1$ and $t3$ are fused, generating place $p2p3$ and transition $t1t3$, respectively, yielding net model $N3$. Net addition has an algebraic notation, which, for this example, is the following:

```
N3 = (N1 + N2) (N1.p2 / N2.p3 -> p2p3,
                N1.t1 / N2.t3 -> t1t3)
```

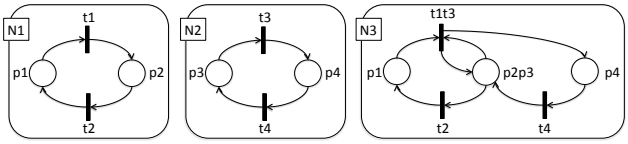


Fig. 2. Net addition of modules `N1` and `N2` and respective result `N3`.

III. Use cases

A use case is defined as "the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system" [4]. A use case model [15] consists of a use case diagrams depicted in UML together with use case descriptions. In this work, a use case is depicted in a diagram and specified in a description. The use case diagram shows use cases, actors, and their relationships. The description of use cases shows, among others, internal flow of events/actions, using natural language.

A. Use case diagram

The use case diagram is the set of use cases, actors, and their relationships. "An actor defines a role that a user can play when interacting with the system. A user can either be an individual or another system." [4]. "Use cases identify and describe each task a system is to able to perform. They are identified by their name and have their description" [6]. The relationships are associations between actors and use cases, as well as use case relationships: in particular the `<<extend>>` and `<<include>>`.

An `<<include>>` relationship denotes the inclusion of use case as a sub-process of the base use case. Thus, while "running" the base use case, it will reach the point of inclusion. The included use case is "executed" at this stage. When it completes, the base use case is resumed after the inclusion point [7]. "Note that the execution of the included use case is not optional and is necessary for the proper functioning of the base use case" [4].

The <<extend>> ”(...) relationship specifies that the behavior of a use case may be extended by the behavior of another (usually supplementary) use case. The extension takes place at one or more specific extension points defined in the extended use case.” [4].

B. Use case description

The use case description we use is structured as follows (based on [6]): (1) Use case title; (2) Summary; (3) Actor(s); (4) Main/basic scenario; (5) Extension points; (6) Alternative scenario; (7) Precondition; (8) Postcondition.

A scenario is a flow of events/actions. Events are behaviors performed by actors for accomplishing their goals. The main scenario is the description of the ”normal” behavior to success, i.e., the main scenario successfully enables the actor to achieve the goal. Each alternative scenario can be an optional behavior of an exception/error condition. The actions/events of the other use case are inserted at specific places in the use case called extension points. Each extension is realized under a specific condition. A precondition defines when the use case is applicable, and the postcondition defines the state the system must be left in when the flow of events completes.

Common representations of these scenarios are single column and double column narratives, where the actions performed by the actor and the system are segregated and represented in their own column. In this paper, we consider a two-column scenario description that addresses the internals of the system, which are at lower levels of abstraction, a white box perspective. This description helps us transform the descriptions into transformation rules and then into Petri net models.

C. Example

From now on, we will be using the following system as an example:

- The system to be controlled is a car parking;
- The car parking has an entry and an exit in two levels;
- Each car can pass to another level, but that is optional.

Fig. 3 shows the use case diagram for this system.

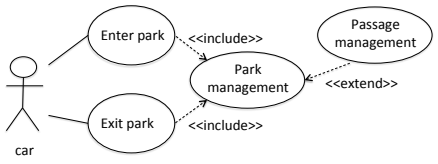


Fig. 3. A use case diagram.

Fig. 4 shows an example of a use case description for the ”Enter park” use case. The ”Exit park” use case is very similar. Due to space restrictions we omit its use case description, as well as the one for the ”Passage management” use case.

D. Rules for scenario description

A use case is a requirements technique that uses textual format. Nevertheless, natural language can lead to ambiguities, making multiple interpretations possible as well as hampering their writing and understanding. To minimize these, we will follow the indications given in [16] and include some additional ones. We consider that a scenario can be regarded as a sequence of *actions*, which we will designate by *use case actions* (UCA) to distinguish them from controller actions. UCA are behaviors of users or systems to accomplish their goals. A set of syntactic rules will be followed while constructing scenarios. As an illustrative example, let us consider the event ”the car gets the ticket”:

Use case description		
Use case title	Enter park	
Summary	A car enters the park, gets a ticket, the gate opens and closes.	
Actor	Car	
Main scenario	**Actor**	**System**
	The car arrives to the entrance.	
	The car is waiting for a ticket	The system prints the ticket when the printer is ok
	The car gets the ticket	*<<include>> Park Management*
	The car is waiting for the gate to open	The system opens the gate
	The car passes through the gate	The system closes the gate
Alternative scenario		
Extension points		
Precondition	The entrance is free	
Postcondition	The entrance is free	

Fig. 4. Use case description for ”Enter park”.

- Names
  - Rule 1[16]: Use the same word for the same description in different sentences (e.g. in our example, we should always use ”ticket” instead of ”entrance document”);
  - Rule 2: Clarify/normalize UCA names; UCA names initially given in the use case description may need to be changed; for example, different users may use different terms to indicate the same UCA;
  - Rule 3[16]: Use the name of a subject/object to replace a pronoun of the subject/object;
- Use case action construction
  - Rule 4[16]: Each UCA is written using simple sentence based on $Subject + Verb + Object$, and each has only one subject and one predicate. A $subject$ can be something or somebody (including an $actor$ or $system$) who performs the event (in our example is ”the car”). The verb is an action (in our example is ”gets”). An object can be something that is handled by the event (in our example is ”ticket”);
  - Rule 5: The verb in the previous rule must be in the present tense or present continuous.
- Template structure
  - Rule 6: Identify preconditions; identify relevant system state and specify preconditions for use cases as a predicate involving the system state;
  - Rule 7: Check if postconditions are specified by examining the effects each use case has on the set of known state variables.
  - Rule 8: The extension points are depicted in the description of the base use case at the respective row of the template description.

– Rule 9: The use case <<include>> relationship is depicted in the $System$ column of the main scenario.

Notice, these rules cannot be completely automated and users and domain experts will still be needed to understand the problem domain.

IV. Use cases as Petri nets

In this section, we present a mapping from use cases to IOPT nets: this mapping has two parts: (1) from use case descriptions to IOPT nets and (2) from use case relationships (<<include>> and <<extend>>) to IOPT nets.

For the translation from use case descriptions (as exemplified in Fig. 4) to IOPT nets, we define an intermediate step: this step uses two types of rules: IF [WHEN] THEN rules and WHEN rules.

These rules were defined with two objectives: (1) to be close enough to the real world so as to allow an easy translation from the use case descriptions, as presented in the previous section; (2) to allow a direct, semi-automatic, translation to IOPT net models. Hence, the rules offer a bridge between two domains: one closer to the "real world" and natural language and the other closer to the "machine world" and the IOPT nets formal language.

The rules — directly translatable to a IOPT net model — have the following format, presented in a simplified form:

1) IF (present state)* AND (input event)* AND (input signal has value X)* [WHEN (internal state)+] THEN (next state)* AND (output event)*;
2) (change output signal value)* WHEN (present state)*; this rule is translated to a place with an associated action.

Regarding the first type of rule, the [WHEN] part is optional and the asterisk is used as *zero or more* and the plus sign as *one or more*. This rule is translated to one transition with its associated annotations — input events, guard checking the input signal (input signal has value X), and output events (that change output signal values) — and the respective neighbor places. These are zero or more input places (*present state*), zero or more output places (*next state*), and zero or more places that are tested (*internal state*) using an input and an output arc.

The second type of rule translates to a single place with an associated output action that can change the value of an output signal depending on the specified condition.

Fig. 5 illustrates the mapping between a use case description and the IF [WHEN] THEN and WHEN rules. It has four columns: (1) use case description; (2) transformation rules; (3) Petri net models resulting from each rule; and (4) the Petri net model that adds all those net models, including the pre and postconditions.

Fig. 6 illustrates the mapping between another use case description, for the "Park management" use case, and the IF [WHEN] THEN rules and respective IOPT model. In fact, this use case could be presented as the result from the composition of two use cases — the "Car arrive" and the "Car leave". Yet, due to space restrictions, we omit that step.

The mappings are based of the following set of assumptions:

- Use case preconditions are expressed as the net initial marking;
- The first UCA and the corresponding IF [WHEN] THEN rule can explicitly refer to the precondition;
- The last UCA and the corresponding IF [WHEN] THEN rule can explicitly refer to the postcondition;
- UCA that use the present tense are translated to IF [WHEN] THEN rules;
- UCA that use the present continuous are translated to WHEN rules;
- The preconditions, the postconditions, and the nets resulting from the previous assumptions will have common places; these should be fused to obtain the complete use case scenario;
- <<include>> relationships are specified in the "System" column;
- <<extend>> relationships are specified in "extensions points"; these are places in the Petri net model that can be fused with places in another use case (net model).

The <<include>> relationship implies that the corresponding transition will be fused with another transition in the included use case. According to the UML Specification "Execution of the included use case is analogous to a subroutine call. All of the behavior of the included use case is executed at a single location in the included use case before execution of the including use case is resumed" [4]. Yet, we assume a weaker form where the included use case is "called" but does not return immediately, hence it is fact similar to the creation of a new process that is synchronized at the time of call. The included use case can optionally again synchronize on the return, but that implies a new transition fusion. This form is more general and takes advantage of the inherent concurrency of Petri nets.

Regarding the <<extend>> relationship, the place fusion allows the net model to optionally execute distinct paths. This is in accordance with the UML specification where the <<extend>> "(...) relationship specifies that the behavior of a use case may be extended by the behavior of another (usually supplementary) use case". In our example this happens with the "Passage management", which extends the "Park management" in places "hasParkedCars" and "hasFreePlaces", as the passage of cars is optional — each car can stay in one level (park) or move to the other park.

Finally, Fig. 7 shows the IOPT net model for the "Park Management" and "Passage management" use cases. The latter allows cars in one level to optionally pass to another level. The respective resulting net composition is illustrated in Fig. 8 where a park with two entrances ("Enter Park" use case), two exits ("Exit Park" use case), two "Park managements", and one "Park passage" are used. The cars in one level can pass to the other level through transitions passUp and passDown. The resulting model effectively forbids park access if there is no free parking place in the respective level. Yet, a more general model can also be specified to allow car entrance when there is at least a free parking place in some level.

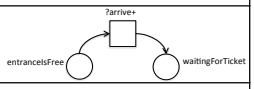
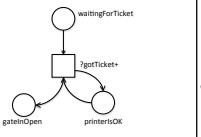
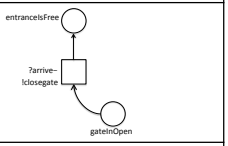
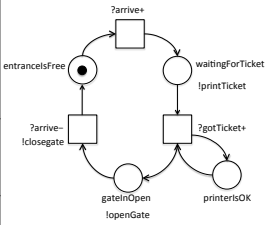
Use case description			Transformation rules	Petri net models	Composed Petri net model
Use case title	Enter park				
Summary	A car enters the park, gets a ticket, the gate opens and closes.				
Actor	Car				
Main scenario	**Actor**	**System**			
	The car arrives to the entrance.		**IF** the entrance is free **AND** car arrives **THEN** car is waiting for a ticket		
	The car is waiting for a ticket	The system prints the ticket when the printer is ok	System prints the ticket **WHEN** car is waiting for a ticket		
	The car gets the ticket	*<<include>> Park Management.enter*	**IF** car is waiting for a ticket **AND** car gets a ticket **WHEN** printer is ok **THEN** car is waiting for the gate to open		
	The car is waiting for the gate to open	The system opens the gate	System opens the gate **WHEN** car is waiting for the gate to open		
	The car passes through the gate	The system closes the gate	**IF** gate is open **AND** car passes through the gate **THEN** entrance is free **AND** system closes the gate		
					
Alternative scenario					
Extension points					
Precondition	The entrance is free				
Postcondition	The entrance is free				

Fig. 5. Mapping from use case ”Enter car” to IOPT model.

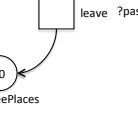
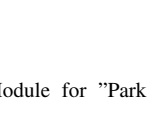
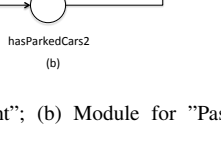
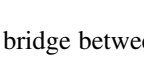
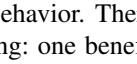
Use case description			Transformation rules	Petri net models	Composed Petri net model
Use case title	Park Management				
Summary	Counts the entrance and exit of cars in the park				
Actor	Car				
Main scenario	**Actor**	**System**			
	The car arrives		**IF** car park has free places **AND** car arrives **THEN** has parked cars		
		Has parked cars			
	The car leaves		**IF** has parked cars **AND** car leaves **THEN** park has free places		
Alternative scenario					
Extension points					
Precondition	Park has 50 free places				
Postcondition	Park has 50 free places				

Fig. 6. Mapping from use case ”Park management” to IOPT model.

Fig. 7. (a) Module for ”Park management”; (b) Module for ”Passage management”.

V. Related Work

The mapping between use cases and Petri nets has been presented before, although not in the context of controller design. The paper by Somé [7] presents a formal mapping from textual descriptions of use cases to a class of Petri nets named Reactive Petri nets. This mapping allows to automatically bridge between use cases and a formal model of the systems behavior. There are several potential benefits for such a mapping: one benefit is to ensure use cases effectively express properties that can be verified and are consistent. Additionally, the mapping provides a definition of formal execution semantics for textual use cases. A distinction between this and our approach is that while we propose transformation rules from requirements to design phase, Som [7] proposes transformation rules from textual use cases to formal models at the requirements phase, i. e., those rules are proposed as a formalism for use cases description and analysis. Meanwhile they also provide a definition of execution semantics for use cases. The UML `<<include>>` and `<<extend>>` relationships are considered in both papers.

In [17], the authors propose the Constraints-based Modular

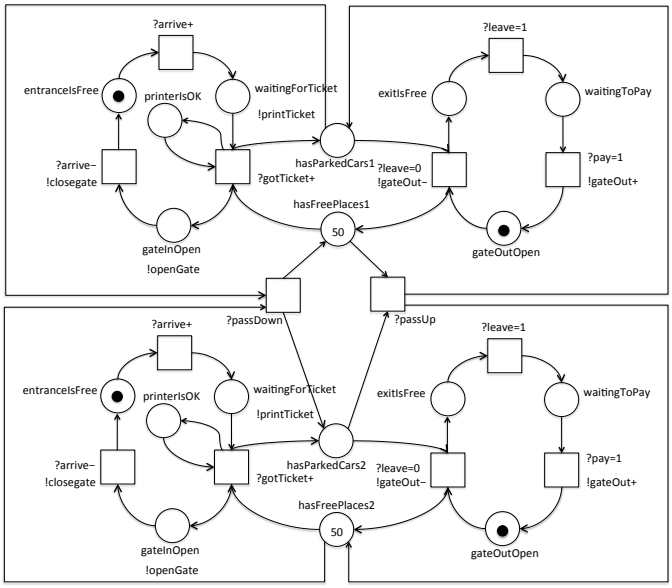


Fig. 8. IOPT park model.

Petri Nets (CMPNs) approach to formalize textual use cases descriptions. To accomplish this, they propose (1) a systematic procedure to convert use cases description to a CMPN model and (2) a set of guidelines to find inconsistency and incompleteness in CMPNs. Regarding use case specification, this work differs from ours mainly because they use an action condition table, instead of rules, to convert use case descriptions to CPMNs.

In [16] Timed and Controlled Petri Nets are used for overcoming limitations of the use case approach, more specifically TCPNs are proposed as the formal description and verification mechanism for use cases. The work elicits the requirements and specifies scenarios based on use cases. After specifying the scenarios, each of them can be transformed into its correspondent Petri net model. Analyzing these Petri net models, some flaws or errors on the requirements can be detected.

Comparing with these previous works, our approach shares the support to translate use case descriptions to Petri net models. Yet, in our case, this translation is added to a complete design flow allowing the effective implementation of controllers in specific hardware. This is made possible due to the use of IOPT nets, a classe of non-autonomous Petri nets to which a full web-based developing environment is available, including model verification and code generation [9], [10].

VI. Conclusions

As other Petri net models, IOPT models are often build with no clear analysis phase. The contributions of this work is a set of rules to support traceability between the use case approach and IOPT Net models. These rules improve use case descriptions and provide a semi-automatic way to translate use case descriptions and diagrams into IOPT models. In this way, the controller specification can be started closer to the real world and be systematically translated to IOPT net models and executable code.

As future work, we intend to generate test cases from use cases and to expand the translation of the `<<include>>` relationship to allow a structured way to specify the subroutine call semantics assumed in the UML specification. Finally, the present proposal will be applied to larger examples, which should provide useful feedback for improvements and the construction of a CASE tool to be integrated with IOPT tools.

Acknowledgment

This work is financed by National Funds through Portuguese Agency "FCT - Fundação para a Ciência e a Tecnologia" in the framework of project PTDC/EEI-AUT/2641/2012.

References

[1] "Petri nets tool database," accessed on 2012/04/07. [Online]. Available: https://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html

[2] L. E. Holloway, B. H. Krogh, and A. Giua, "A Survey of Petri Net Methods for Controlled Discrete Event Systems," *Discrete Event Dynamic Systems*, vol. 7, pp. 151–190, 1997.

[3] R. David and H. Alla, *Discrete, Continuous, and Hybrid Petri Nets*, 2nd ed. Springer Publishing Company, Incorporated, 2010.

[4] OMG. OMG Unified Modeling Language™ (OMG UML), Superstructure, version 2.4.1. http://www.omg.org/spec/UML/.

[5] ——. OMG Systems Modeling Language (OMG SysML™), version 1.3. http://www.omg.org/spec/SysML/1.3/.

[6] K. Bittner and I. Spence, *Use Case Modeling*, ser. The Addison-Wesley object technology series. Addison Wesley, 2003.

[7] S. S. Somé, "Formalization of textual use cases based on petri nets." *International Journal of Software Engineering and Knowledge Engineering*, vol. 20, no. 5, pp. 695–737, 2010. [Online]. Available: http://dblp.uni-trier.de/db/journals/ijseke/ijseke20.html\#Some10

[8] L. Gomes, J. Barros, A. Costa, and R. Nunes, "The Input-Output Place-Transition Petri Net Class and Associated Tools," in *Proceedings of the 5th IEEE International Conference on Industrial Informatics (INDIN'07)*, Vienna, Austria, Jul 2007.

[9] L. Gomes, F. Moutinho, and F. Pereira, "IOPT-tools — A Web based tool framework for embedded systems controller development using Petri nets," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, Sept 2013, pp. 1–1.

[10] GRES Research Group, "IOPT Tools," accessed on 2014/03/06. [Online]. Available: http://gres.uninova.pt/IOPT-tools

[11] J. Desel and W. Reisig, "Place/transition Petri nets," in *Lectures on Petri Nets I: Basic Models*, ser. Lecture Notes in Computer Science, W. Reisig and G. Rozenberg, Eds. Springer Berlin Heidelberg, 1998, vol. 1491, pp. 122–173. [Online]. Available: http://dx.doi.org/10.1007/3-540-65306-6_15

[12] W. Reisig, *Understanding Petri Nets - Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013.

[13] L. Gomes and J. Barros, "Structuring and composability issues in Petri nets modeling," *IEEE Transactions on Industrial Informatics*, vol. 1, no. 2, pp. 112–123, May 2005, http://dx.doi.org/10.1109/TII.2005.844433.

[14] J. Barros and L. Gomes, "Net model composition and modification by net operations: a pragmatic approach," in *Proceedings of the 2th IEEE International Conference on Industrial Informatics (INDIN'04)*, June 2004.

[15] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-oriented software engineering – a use case driven approach*. Addison-Wesley, 1992.

[16] J. Zhao and Z. Duan, "Verification of use case with petri nets in requirement analysis," in *ICCSA (2)*, ser. Lecture Notes in Computer Science, O. Gervasi, D. Taniar, B. Murgante, A. Laganà, Y. Mun, and M. L. Gavrilova, Eds., vol. 5593. Springer, 2009, pp. 29–42.

[17] W. J. Lee, S.-D. Cha, and Y.-R. Kwon, "Integration and analysis of use cases using modular petri nets in requirements engineering," *IEEE Transactions on Software Engineering*, vol. 24, no. 12, pp. 1115–1130, Dec 1998.