STATE MACHINES AND PETRI NETS AS A FORMAL REPRESENTATION FOR SYSTEMS LIFE CYCLE MANAGEMENT

Eric Simon
Information Management Institute, Université de Neuchâtel,Pierre-à-Mazel 7, 2000 Neuchâtel, Switzerland
eric.simon@unine.ch

Kilian Stoffel
Information Management Institute, Université de Neuchâtel,Pierre-à-Mazel 7, 2000 Neuchâtel, Switzerland
kilian.stoffel@unine.ch

ABSTRACT

In systems life cycle management (SLCM), there is a gap between the rather informal methodologies for systems development and the sound mathematical formalisms needed for the automatic validation of systems properties and correctness proofs. This paper presents a model based on finite state machines and its translation into Petri nets, a mathematical representation with the desired degree of provability in this context. We argue that the model can bridge that gap, by allowing inexperienced non specialists to represent their development methodologies and all their key features using simple automata and applying a systems thinking approach to problem solving, instead of having to express the model in a more complicated representation from the start.

KEYWORDS

State machine, Petri net, life cycle

1. INTRODUCTION

In the context of systems life cycle management (SLCM), there exist methodologies to assist the different actors performing the activities necessary to develop and maintain large, complex yet reliable information systems. Some are very strict and follow the philosophy of "big design up front", like the waterfall model (Royce, 1970), some take into account the iterative nature of software development, like the spiral model (Boehm, 1986), others are based on adaptive models, like rapid application development (RAD) (Martin, 1991), or the many methods grouped under the general denomination of "agile" (Beck, 2001). These methodologies are usually rather informal and targeted at users: developers, managers, suppliers, acquirers, even lawyers or any other actor taking part in the development process.

On the other hand, in the more technical context of software development, formalisms exist to model software or at least some of its key features with sound mathematical foundations giving the possibility to validate systems properties and do correctness proofs. To mention only the few that are of interest in this particular context of representation, let us say that automata theory in general, and more particularly the theory of virtual finite state machines and event driven finite state machines, allows the execution of a software specification from a formal representation. These techniques are often used to develop either safety critical applications or control software. In the same niche, Petri nets (Petri, 1962) are routinely applied to formalise concurrent or real time systems, in order to ensure a high level of reliability. Other formalisms were developed for systems software with inherent complexity and strong reliability requirements, such as embedded systems or safety critical applications, often with accompanying toolkits:

- The Vienna Development Method (VDM), and its specification language VDM-SL and later VDM++ (Bjørner, 1978)

- Raise (Rigorous Approach to Industrial Software Engineering) (Raise Method Group, 1995) and its specification language: the Raise Specification Language (RSL) (Raise Method Group, 1992)
- The B-Method (Abrial, 1996), derived from Z notation (Abrial, 1980) (now an ISO standard ISO/IEC 13568:2002)

More recently, first order logic inspired formalisms have been appearing in publications as well (Martin, 2004).

The problem lies in the gap between development methodologies or best practices on one hand, and the provable formal models on the other hand. Users often apply some methodology to software development, and may even use rigorous methods like the ones mentioned above, but nothing exists that formalises the actual processes constituting the whole life cycle of the system.

A first step toward formalisation has been taken by large organisations or administration, where the need for such standards is the most critical, with documents such as the US Department of Justice definition of SDLC or the ISO/IEC 12207 *Standard for Information Technology* for SLCM (ISO/IEC, 1995). These standards suffer from two main disadvantages though: First, they're very difficult to follow to the letter, and therefore to enforce, being inherently very complex, and second, although they take into account a certain degree of flexibility regarding the application or exclusion of some process activities, they are not flexible enough for a user to tailor a custom methodology according to the specificities of a given task.

The idea presented in this paper is to take one step further in bridging the gap between SLCM methodologies on the user side, and the mathematical models for validation or proofs. To achieve this goal, a simple extension of finite state machines allowing scalability and synchronisation of concurrent activities is proposed to give non specialists the ability to model well-known or custom SLCM methodologies easily and using a systems thinking approach to problem solving. Section 2 presents the model and Section 3 proceeds by showing that the representation can be mapped to Petri nets, giving the advantages of a mathematically sound formal representation for the validation of systems properties and for correctness proofs. An example illustrating the key features of the model is presented in Section 4. The paper concludes in Section 5 with possible applications, limitations and future work.

2. A MODEL USING FINITE STATE MACHINES AND PETRI NETS

A formalism based on finite state machines (FSM) has been proposed (Simon, 2007) to model the activities involved in SLCM. It extends FSM in general and the social protocols defined by Picard (2005; 2006) that inspired it in two main respects: with recursion, or rather scalability, and with a synchronisation mechanism.

2.1 Scalable and component state machines

The first extension can be seen as a recursive definition, similar in essence to that of recursive state machines (Alur, 2005), which allows the replacement of activities (arcs) in a FSM by another FSM. In other words, it allows the viewing of an activity as a simple transition, or as a more detailed process. It is much simpler than the definition of a recursive state machine, and based on the semantic equivalence of two different state machines seen at different levels of detail. In this sense it is more appropriate to speak of "zooming" or "scalability", so we use the terminology "*scalable state machine*" (SSM) and "*component state machine*" (CSM) in the remainder of this document when we refer to the extended FSM.

Intuitively, as a prelude to a more formal definition of the model that follows, there are two conditions to satisfy for this scalability property to hold:

1. The source states (entry nodes) and the destination states (exit nodes) must be univocally identified and respectively identical for the considered transition T and the component FSM it stands for. This corresponds to the requirement of a well-defined interface in the definition of a recursive state machine, with the limitation that we consider only one entry state and one exit state.
2. The role associated with an arc must be in phase with the "overall role" of the component automaton. A certain freedom exists as to how to define this "overall role", depending on the semantics of the process.

Definition. A *scalable state machine* Σ is a finite state machine $(S, s_{src}, S_{dst}, A, T)$:

- S is the set of all states
- $s_{src} \in S$ is the source state, or entry state
- $S_{dst} \subseteq S$ is the set of destination states, or exit states
- A is the set of *activities*
- $T : S \times A \rightarrow S$ is the state-transition function

Definition. An *activity* $\alpha \in A$ is a tuple *(role, action).* A *role r* is a label, it identifies the users or entities that performs the action. Note that a role can be an abstract thing played by many users (a team) or software agents. An *action a* is the execution of a task in the context of SLCM. Usually such a task produces a deliverable, such as a document or a piece of software.

Definition. A *component state machine* is a scalable state machine Σ with exactly one source state $s_{src} \in S$ and exactly one destination state $s_{dst} \in S$.

Definition. A scalable state machine $\Sigma = (S, s_{src}, s_{dst}, A, T)$ is *semantically equivalent* to another scalable state machine Σ' where the transition $t = (s^t_{src}, \alpha^t, s^t_{dst}) \in T$ has been replaced by a component state machine $\mathrm{K} = (S^{\mathrm{K}}, s^{\mathrm{K}}_{src}, s^{\mathrm{K}}_{dst}, A^{\mathrm{K}}, T^{\mathrm{K}})$ if and only if the following conditions hold:

1. Source and destination states respectively are identical for the CSM K and the transition t it replaces: $s^t_{src} = s^{\mathrm{K}}_{src}$ and $s^t_{dst} = s^{\mathrm{K}}_{dst}$.
2. A meaningful *overall role* r^{K} and a meaningful *overall action* a^{K} of the SSM K that correspond respectively to the role r^t and action a^t in activity $\alpha^t = (r^t, a^t)$ can be defined.

Condition 1 implies the uniqueness of the destination state. An arc has exactly one source and one destination state. In effect, the overall process can be represented as a SSM, with multiple destination states, but any activity itself is only represented by a CSM, i.e. it must have only one destination state. This makes sense in our context. A process can very well have two distinct destination states, one for success and one for failure for example, but any arc in itself must lead to a well-defined, unique state, else the composition doesn't make sense.

In general, condition 2 always holds as a role is only a label and one can define a meta-role capturing the semantics of all the roles involved in the new SSM K replacing transition t, or simply use the last activity completing the task, i.e. the last arc reaching the destination state of K. The same holds for the action.

Using the full definition of recursive state machines would allow for much more flexible compositions. However, for the problem at hand, it is not necessary. The semantics behind the SLCM activities is about producing deliverables or results, and alternative destination states can therefore always be combined into one single ending state which signifies the acceptance of the considered process after some possibilities that would otherwise be considered as final states have been reached. In other words, a scalable state machine (with multiple destination states), can be transformed into a component state machine (with only one destination state) with no loss of functionality. In the success/failure example, on possibility consists in defining another state meaning the end of the project and making it the only final state.

Furthermore, as stated in the introduction, the idea is to keep the formalism as simple as possible for a non-specialist, and combining automata with interfaces of multiple entry and exit states recursively is far more complicated, even though it is mathematically much more elegant and powerful than the proposed model.

2.2 Synchronised state machines

The second extension is a simple "rendez-vous" type of synchronisation mechanism to model parallel activities and hold subsequent dependant activities until the completion of all the pre-requisites. The synchronisation is expressed by a logical AND between the set of final states of the synchronised automata and the set of source states of the dependant activities. It isn't formally part of the mathematical model itself, but only a convenient way of representing concurrent activities using SSM for non-specialists. Figure 1 shows an example of such a synchronisation and the chosen notation. The arrows are dashed to emphasise the fact that the arcs are not transitions of the SSM as defined above and do not carry meaning about a role or an action, but are there only for synchronisation.

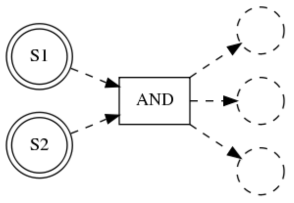

Figure 1. The logical AND defines as a "rendez-vous" synchronisation mechanism between the final states S1 and S2, before the triggering of the three events on the right. The arrows are dashed because they're not strictly arcs of the automata

3. MAPPING THE MODEL TO PETRI NETS

It is known that FSM can be seen as a special case of Petri nets, where each transition, now represented as a box, can have one and only one incoming arc and one and only one outgoing arc. The mapping of a SSM to a Petri net is then trivial as it goes from the more specific to the more general model. First, Petri nets are presented briefly in Section 3.1 in the context of our problem, together with existing related work. Sections 3.2 and 3.3 then show that the two extensions presented in Section 2, scalability and synchronisation, do not break any property that would prevent the mapping.

3.1 Petri nets

The strength of the model (its simplicity for users) is also its weakness. The two extensions are not really part of the mathematical system itself, so in order to represent explicitly these processes and validate system properties or do correctness proofs another model is needed that satisfies the following properties:

- The model must take into account the possibility of concurrent activities (while retaining the other features of SSM/CSM of course, like the causal dependencies)
- A mapping must exist between SSM and the new model that doesn't break any condition or include new information requiring human intervention.

As mentioned in the Introduction, Petri nets, also often referred to as place/transition nets, were proposed as a mathematical language and representation for discrete distributed systems, where concurrency and causal dependencies must be represented explicitly. Their application to workflow management in particular has been investigated by Aalst (1998).

There also exists a refinement of social protocols using the coloured Petri defined by Jensen (1992) as a mathematical language (Picard, 2008). The purpose of the development of social protocols is to model human-to-human interactions over a network from the ground up, while retaining a strong mathematical foundation, and explore the possibilities offered by successive refinements of the model, for example a direct structural validation (Picard, 2007). Our approach is quite different in that it is inspired by very pragmatic concerns: The purpose is to give non specialists the simplest possible representation language for SLCM, and then transform it into a sound mathematical model allowing the verification of certain system properties, the validation of the processes and automatic correctness proofs. Petri nets are mathematically sound and offer such possibilities (Reisig, 1985). Furthermore, the mapping of our extension of FSM to a Petri net is straightforward, as showed in Section 2.3.

Other possibilities to represent our model with a sound mathematical language have been investigated: Conceptual Graphs (Sowa, 1976) and Description Logic. They possess sound mathematical foundations and are very expressive languages, but that very last feature is what ruled them out. The complete argument is beyond the scope of this paper, but to state it in one sentence: Representing processes using conceptual graphs, description logic (e.g. as ontologies in OWL-DL) or any other static model or language like first-order predicate logic adds the burden of representing dynamic activities explicitly, while Petri nets were designed from the ground up to fulfil that very requirement while being very simple to represent graphically and be comprehensible by human beings.

Coloured Petri nets were ruled out because the semantics of what is being produced is attached to the states themselves. Adding types would permit the explicit representation of what is being produced, a specific document or a software module for example, but this information would have to be also explicit in the simple representation of the processes on the side of the users, i.e. in the SSM/CSM, and that would add unnecessary complexity and burden to the users.

3.2 Scalability

In the mapping of FSM to Petri nets, a state-transition (an arc) of the FSM becomes a transition in the Petri net, usually represented by a rectangle. Since scalability deals only with the (semantic) equivalence of two SSM at two different level of detail, a SSM where a transition has been replaced by a CSM is nothing more than a special case of FSM and as such can be mapped directly to another Petri net.

Note that it cannot be said that this second Petri net is equivalent to the first one, as the equivalence of Petri nets is something completely different that has to do with the way the nets behave (isomorphism) and implies among other things that two equivalent systems always have the same number of cases, events and steps, which is clearly not the case here.

3.3 Synchronisation

The argument for the second extension is the same as for scalability above. What happens exactly when we represent as a Petri net a logical AND between the set of the destination states of some SSM on one side and the triggering of initial states of other SSM on the other side? In the theory of Petri nets, such a synchronisation is simply a transition (a rectangle) with multiple incoming arcs, one for each of the place (state) that has to contain a token in order to enable the transition, and where the outgoing arcs, possibly only one or even none, enables the places (states) that have to be activated. The example in Figure 2 shows the Petri net corresponding to Figure 1.

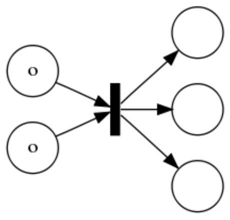

Figure 2. The logical AND show in Figure 1 mapped as a new transition with multiple incoming and outgoing arcs. Note the tokens in the two final states on the left, meaning that the transition is ready to fire, i.e. to be executed

4. EXAMPLE

This section illustrates the application of the model and its mapping to Petri nets using an activity defined in the ISO/IEC 12207 document (ISO/IEC, 1995). It encompasses a much simplified version of activity 5.1.1 “Acquisition process” (by the role “acquirer”), which was reduced to the SSM shown in Figure 3, retaining only those properties necessary to illustrate scalability and synchronisation. It is evident that real processes are rather more complex than the one presented in this paper, which serves only as an illustration. The arcs are sub-activities as defined in the reference document. For instance, in this particular case, there is an alternative between two activities to reach the final state, corresponding to either buying the software (5.1.1.3), or developing it in-house (5.1.1.4), respectively. The mapping to the corresponding Petri net is straightforward. The transitions (the black rectangles) are not labelled on purpose, for legibility reasons. In reality, they carry the same semantics as the activities in the SSM, i.e. a role and an action.

A single token is present in the first place of the Petri net, showing that transition corresponding to activity 5.1.1.1 is enabled. The token itself carries no meaning. The semantics of the state of the system is dependant only on the place (state) it is in and what is being produced: a deliverable, document, report, piece of software, or several of those items.

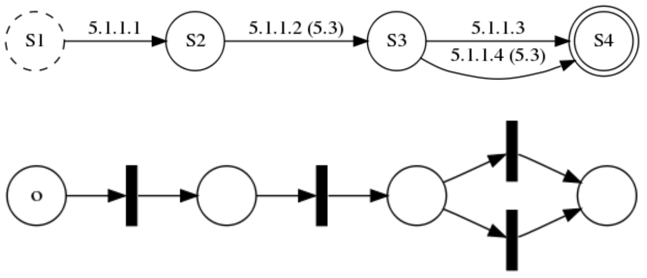

Figure 3. Activity 5.1.1 expressed as a SSM and as the corresponding Petri net. This is a much simplified version of the actual process described in ISO/IEC 12207. The arcs correspond to sub-activities and are labelled accordingly

Note that including several tokens in one single state of a given process would carry no meaning in this particular case, the processes being in essence completed when the next place in the diagram is reached.

Scalability. In Figure 3, activities 5.1.1.2 (design) and 5.1.1.4 (development) are meant to be executed using activity 5.3, which defines a development procedure using a strict waterfall model. This is why 5.3 has been included in parenthesis in the example automaton. This information is meaningless for the mapping itself: it is intended only to render the labels more precise. The first strength of the model is that it allows someone without many skills to take the SSM in Figure 3 and replace activity 5.1.1.4 by some development process: either 5.3 or a custom tailored one. Figure 4 shows a possible CSM that could be devised to perform activity 5.1.1.4, which would replace the corresponding transition in a more detailed view. The states S3 and S4 are retained as they are the same for both the SSM in Figure 3 and Figure 4.

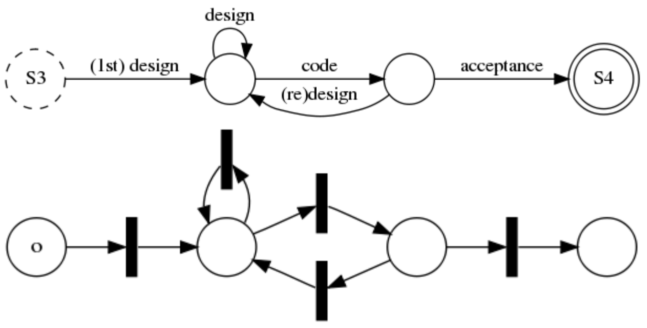

Figure 4. A possibility for activity 5.1.1.4 expressed with a higher level of detail, as a SSM and as the corresponding Petri net. States S3 and S4 are those of Figure 3.

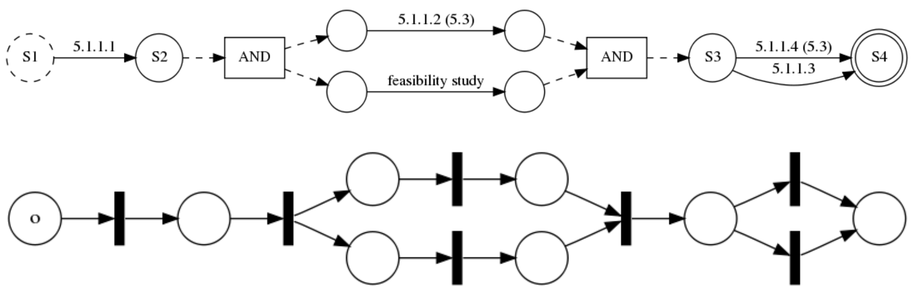

Figure 5. Activity 5.1.1.1 as a synchronised SSM, with the addition of a parallel activity “feasibility study” and the synchronisation mechanism, and the corresponding Petri net

Synchronisation. To illustrate synchronization, a new activity "feasibility study" conditioning the buying or development of software in Figure 1 (5.1.1.4 and 5.1.1.3 respectively) is added. This activity must take place after state S2, and before state S3, but can be conducted in parallel to activity 5.1.1.2 (design). Figure 5 illustrates the new SSM with the synchronisation, and its corresponding Petri net.

Finally, just as an illustration, Figure 6 shows the complete resulting system as a Petri net where both activities 5.1.1.2 and 5.1.1.4 have been replaced by the custom development process presented in Fig. 4. It is still quite simple, due to the nature of the examples, but one can easily imagine the problems arising with real processes if one was to model them using Petri nets directly. Of course, a trained user could do it by combining component Petri nets in much the same way one would do it with SSM. The main advantage of using synchronised SSM is that it actually forces even unskilled people to break down the processes into smaller pieces, which is the essence of systems thinking, the root of all SLCM methodologies.

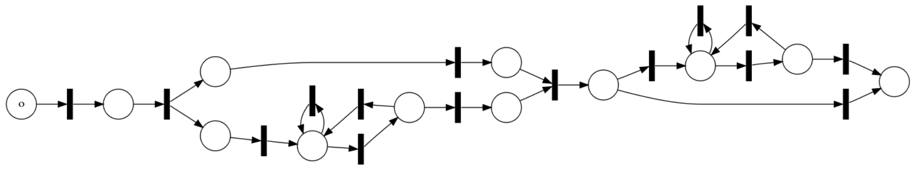

Figure 6. Complete Petri net for activity 5.1.1. It is still rather simple, but an untrained user would have trouble breaking down anything as complicated as the processes described in the ISO/IEC 12207 document into simpler components from the start and express them as Petri nets from the start

5. CONCLUSION

While many methodologies for SLCM exist, as well as many formal representations for software development, there is a gap between the two approaches that prevents from applying rigorous validation methods to development methodologies in the context of SDLC. The model presented in this paper is one step further in the direction of a system that could assist users in SLCM.

The main contribution is the clear definition of a representation for SLCM processes, which cannot be used as such for validation of systems properties, but which can be mapped easily to a more general and expressive representation that has the desired properties: Petri nets. The model has the advantage that it enforces a systems thinking approach to problem solving from the start, while being simple enough for non specialists. It therefore sacrifices mathematical elegance to usability, and has a certain trade-off between expressiveness and simplicity, which we argue is not a limitation in our case.

The next step will be the practical application of the model, by providing SLCM actors with a system, first theoretical then an actual software prototype, to represent their methodologies and validate such system properties as the detection of inconsistencies, the reachability of some states, the structural validation when a change occurs, the simulation of possible cases and any feature that the analysis of Petri nets permits. A prototype exists that implements the work presented in this paper, i.e. both the representation of SLCM as SSM and the mapping of those to Petri nets. We intend to extend it with the inference mechanisms and algorithms necessary for these features to be tested in practice.

Another research that could benefit from a similar mapping is the modelling of legal or administrative documents as business processes (Coțofrei, 2008).

ACKNOWLEDGEMENT

This Work is supported by the Hasler Foundation, Switzerland: project ManCOM 2085.

REFERENCES

van der Aalst, W.M.P., 1998. The Application of Petri Nets to Workflow Management. *In The Journal of Circuits, Systems and Computers*, Vol. 8 No. 1, pp. 21-66.

Abrial, J.R. et al, 1980. *On the Construction of Programs.* Cambridge University Press, UK.

Abrial, J.R., 1996. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, UK.

Alur, R. et al, 2005. Analysis of Recursive State Machines. *ACM Transactions on Programming Languages and Systems,* Vol. 27 No. 4, pp. 786-818.

Beck, K. et al, 2001. *Manifesto for Agile Software Development.* URL: http://agilemanifesto.org/.

Bjørner, D. and Jones, C.B., 1978. The Vienna Development Method: The Meta-Language. *In Lectures Notes in Computer Science,* Vol. 61, Springer-Verlag, Berlin Heidelberg, Germany.

Boehm, B., 1986. A Spiral Model of Software Development and Enhancement. *ACM SIGSOFT Software Engineering Notes,* Vol. 11.

Coțofrei, P. and Stoffel, K., 2008. Business Process Modelling for Academic Virtual Organizations. *In Pervasive Collaborative Networks*, Springer, Boston, pp. 213-220.

International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), 1995. *ISO/IEC 12207.* Standard for Information Technology.

Jensen, K., 1992. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* Vol. 1. Monographs in Theoretical Computer Science, Springer-Verlag, Germany.

Martin, A., 1991. *Rapid Application Development.* Macmillan Coll Div.

Martin, A., 2004. Relating Z and First-Order Logic. *Lectures Notes in Computer Science,* Vol. 1709, No. 1999, Springer, Berlin Heidelbert, Germany, pp. 715.

Petri, C. A., 1962. *Kommunikation mit Automaten*. Ph.D. Thesis. Schriften IIM Nr. 2, Institut für Instrumentelle Mathematik, University of Bonn, Germany.

Picard, W., 2005. Modeling Structured Non-monolithic Collaboration Processes. *Proceedings of the 6th IFIP Working Conference on Virtual Enterprises: Collaborative Networks and their Breeding Environments. Camarinha-Matos, L., Afsarmanesh, H., Ortiz, A., eds.* Springer, Valencia, Spain, pp. 379-386.

Picard, W., 2006. Computer Support for Adaptive Human Collaboration with Negotiable Social Protocols. *In Lectures Notes in Informatics (LNI),* Vol. 85*: BIS 2006.* Abramowicz, W., Mayr, H.C., eds. GI, Germany, pp. 90–101.

Picard, W., 2007. An Algebraic Algorithm for Structural Validation of Social Protocols. *Proceedings of Business Information Systems, 10th International Conference, BIS 2007. Poznan, Poland, pp. 570-583.*

Picard, W., 2008. Modelling Multithreaded Social Protocols with Couloured Petri Nets. *In IFIP International Federation for Information Processing,* Vol. 283*: Pervasive Collaborative Network.* Camarinha-Matos, L.M., Picard, W., eds. Springer, Boston, pp. 343-350.

Raise Method Group, 1992. *The Raise Specification Language.* Prentice-Hall, US.

Raise Method Group, 1995. *The Raise Method Manual.* Prentice-Hall, US.

Reisig, W., 1985. *Petri Nets: An Introduction*. Springer-Verlag, Berlin Heidelberg, Germany.

Royce, W.W., 1970. Managing the Development of Large Software Systems. *IEEE Wescon,* pp. 1-9.

Simon, E. et al, 2007. Scalable Social Protocols to Formalize Systems Development Life Cycles. *Proceedings of IADIS International Conference e-Society*. IADIS Press, Lisbon, Portugal, pp. 177-184.

Sowa, J.F., 1976. Conceptual Graphs for a Data Base Interface. *IBM Journal of Research and Development,* Vol. 20, No. 4, pp. 336-357.