IADIS International Journal on Computer Science and Information Systems
Vol. 4, No. 3, pp. 39-50
ISSN: 1646-3692

TOWARDS BRIDGING THE GAP BETWEEN INTUITIVE AND FORMAL REPRESENTATIONS OF SYSTEMS LIFE CYCLE PROCESSES

Eric Simon. *Institut du management de l'information, Université de Neuchâtel, Switzerland. eric.simon@unine.ch*

Kilian Stoffel. *Institut du management de l'information, Université de Neuchâtel, Switzerland. kilian.stoffel@unine.ch*

ABSTRACT

In systems life cycle management (SLCM), a gap exists between the informal methodologies for systems development and the 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 effectively bridges the gap between the intuitive representation of development processes on one hand, and the formal model necessary for validation on the other hand, by allowing users without scientific or technical background to represent their activities and all their key features using simple automata, by applying a systems thinking approach to problem solving, instead of having to express the model in a more complicated representation from the start. Also, this model shows very promising results in other domains routinely modelled as activities, using other formalisms, like business processes.

KEYWORDS

State machine, Petri net, life cycle

1. INTRODUCTION

In the context of systems life cycle management (SLCM), a clear gap exists between the methodologies devised to develop and manage such systems on one hand, and the formalisms used to represent and analyse the involved processes on the other hand.

First, the methodologies applied by development teams are meant as guidelines rather than celarly defined processes, even though some are very strict and follow the philosophy of "big

design up front", like the waterfall model (Royce, 1970). Other take into account the iterative nature of software development, like the spiral model (Boehm, 1986), or are based on adaptive models, like rapid application development (RAD) (Martin, 1991), and the many methods grouped under the general denomination of "agile" (Beck, 2001), in which case they are even more informal and subject to interpretation.

Second, in the technical context of software development, formalisms to model software or at least some of its key features with sound mathematical foundations give the possibility to validate systems properties and do correctness proofs. Automata theory in general, and particularly the theory of virtual finite state machines and event driven finite state machines, allow 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 domain, Petri nets (Petri, 1962) are routinely applied to represent and analyse 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, like the Vienna Development Method (VDM), and its specification language VDM-SL and later VDM++ (Bjørner, 1978), Raise, i.e. Rigorous Approach to Industrial Software Engineering (Raise Method Group, 1995) and its specification language: the Raise Specification Language (RSL) (Raise Method Group, 1992) or the B-Method (Abrial, 1996), derived from Z notation (Abrial, 1980), now an ISO standard (ISO/IEC 13568:2002). Formalisms inspired by first order logic have been appearing in publications as well (Martin, 2004). These formalisms share a common denominator: they are too complicated for people without computer science or engineering background, what we refer to as "non-specialists" in the remainder of this document.

There lies the essence of the gap between these two worlds. Nothing exists to formalise the actual processes involved in the life cycle of the systems while being simple enough for non-specialists to use. A step towards formalisation has been taken by large organisations or administrations, where the need for such standards is 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 resulting from their sheer complexity though: First, they're very difficult to follow to the letter as such, and second, they are very difficult, if not impossible, to validate.

The idea presented in this article is to take one step further in bridging the gap between system development methodologies on one side, and the mathematical models for validation or proofs on the other side. To achieve this goal, a simple representation with the minimum elements is proposed to give non-specialists the ability to model well known or custom SLCM methodologies easily by enforcing a systems thinking approach to problem solving. This representation is then mapped into a well-established formalism with the desired properties for validation: Petri nets. Section 2 presents the representation based on finite state machines and Section 3 proceeds by showing that the mapping to Petri nets is straightforward. An example illustrating the key features of the model is presented in Section 4 and Section 5 concludes with applications, limitations and future work.

2. A MODEL BASED ON FINITE STATE MACHINES

A representation and formalism based on finite state machines (FSM) has been proposed (Simon, 2007) to model the processes involved in systems life cycle management. It extends state machines in general and the social protocols defined by Picard (2005; 2006) that inspired it in particular in two respects: with scalability, and with a synchronisation mechanism. This section introduces finite state machines and its limitations in the context of life cycle processes representation and proceeds by presenting the two proposed extensions the model, one for synchronisation, which is outside the model itself, and one for scalability, which is formally defined.

2.1 Finite State Machines

The theory of finite state machines, or finite state automata, is a well-known model ideal for the representation of linear behavioural processes. Without going into formal details or classifications, which are extensively described in the appropriate literature (Gill, 1962; Ginsburg, 1962; Carroll, 1989), it consists in a set of states, and transitions indicating a change of state triggered by a certain condition. The model is well suited to represent linear processes with alternatives, such as the ones involved in systems life cycle management, but lacks two important properties to be applied in this context:

It doesn't allow the representation of parallel processes, which is essential in the development of complex systems by whole teams of developers, managers etc.

It is not scalable, in the sense that it is very difficult to break down a complex process involving many states and transitions into simpler components. This becomes even more critical when the users involved lack a strong background in the systems thinking approach to problem solving, which is often the case for people without scientific or engineering training, people we referred to in Section 1 as non-specialists.

We propose two extensions of the model to allow the representation of all systems life cycle processes:

Synchronised state machines

Component-based state machines

2.2 Synchronised State Machines

This extension is a simple “rendezvous” type of synchronisation mechanism to model parallel activities and hold subsequent dependant activities until the completion of all the pre-requisites. Note that it is different from the real rendezvous in the context of parallelism, which synchronises threads (processes) that continue after having met at that point. In our case the parallel processes themselves do not continue, it is the whole process that is held until the parallel processes all reach a final state. The synchronisation is represented by an AND between the set of final states of the synchronised automata and the set of source states of the dependant activities. It is not formally part of the mathematical model itself, but only a convenient way for non-specialists of representing concurrent activities using state machines. 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 state machine and do not carry meaning about a role or an action. In this particular example we have a whole automaton

composed of states S1 to S7, that is itself composed of two components, the red and the blue automata respectively. The synchronisation is two-fold: first S2 and S3 are triggered by an AND with two outbound arrows, and second S4 and S6 must both be reached in order to proceed to the final state S7. The red and blue state machines are therefore parallel activities that can be conducted separately, but must both be finished before the whole process can continue.

Note that there is no need for an OR, as an alternative is of course already part of the state machine theory: a state with multiple outbound arcs represents such an alternative, dependant on the conditions expressed by said arcs.

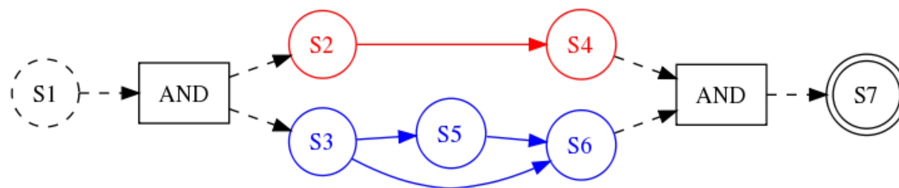

Figure 1. Synchronisation between two parallel automata: The AND on the left means that both red and blue state machines are started after S1. The AND on the right means that both final states of the red and blue state machines must be reached in order to proceed to the final state S7 of the whole process. The dashed arrows emphasise the fact that those arcs are not part of the model.

2.3 Component-based State Machines

This extension can be seen as a sort of 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, there are two conditions to satisfy for this property to hold:

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.

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:

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 $s^t_{dst} = s^{\mathrm{K}}_{dst}$.

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, one 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.

3. GOING FORMAL

The strength of the model presented in Section 2, its simplicity for non-specialists, 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 parallel activities, while retaining the other features of SSM/CSM.

A mapping must exist between the proposed SSM and the model that doesn't break any condition or include new information requiring human intervention.

Petri nets are a well-defined formalism that allows the representation of parallel activities and synchronisation. Section 3.1 presents briefly Petri nets and compares them to other formalisms that can be applied in this domain. Section 3.2 then proceeds by showing that the model proposed in Section 2 can be mapped directly to Petri nets.

3.1 Petri nets

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 extensively by Aalst (1998).

Without going into formal definitions, which like in the case of state machines are extensively described in the relevant literature (Petri, 1968), a Petri net is composed of places and transitions. In the simplest case, which concerns us, a single token is moving from place to place by being consumed and produced by the corresponding enabled transition.

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 rules 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.

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 3.2.

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, and make the proposed model more complicated.

3.2 Mapping the Model

It is well 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 FSM to a Petri net is then trivial as it goes from the more specific to the more general model. The following two sections show that the extensions presented in Section 2, synchronisation and scalability, do not break any property that would prevent such mapping.

3.2.1 Synchronisation

What happens when we represent as a Petri net the synchronisation mechanism ("rendezvous" or 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 an AND in a finite state machine and the corresponding Petri net notation where the AND becomes transition T1 with corresponding inbound and outbound transitions. The numbers on the arcs mean that one token is consumed (inbound arc) or produced (outbound arc) by the transition.

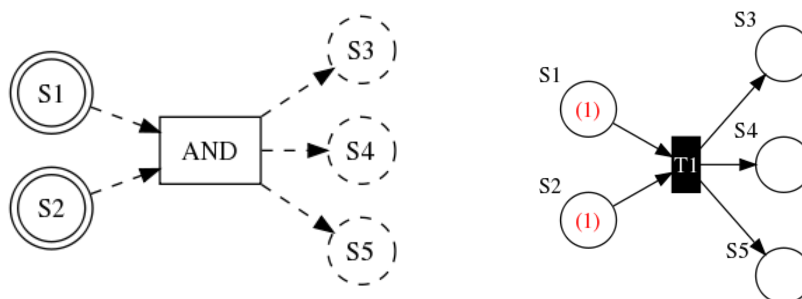

Figure 2. The logical AND shown on the left is mapped as a new transition T1 with multiple incoming and outgoing arcs in the corresponding Petri net on the right. Note the tokens in the two final states on the left, symbolised by red numbers 1 in parenthesis, meaning that the transition is ready to fire, i.e. to be executed.

3.2.2 Scalability

In the mapping of FSM to Petri nets, a state-transition (an arc) of the FSM becomes a transition in the Petri net. 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.

In practice and in other words, the Petri net used for validation is simply the (automatic) mapping of the whole state machine, result of the combination and synchronisation of as many components as the design required.

4. EXAMPLE

This section illustrates the application of the model and it's 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 developing the software in-house (5.1.1.2 using 5.3), or buying it from a third-party (5.1.1.2 outsourced and 5.1.1.3) respectively. The mapping to the corresponding Petri net is straightforward.

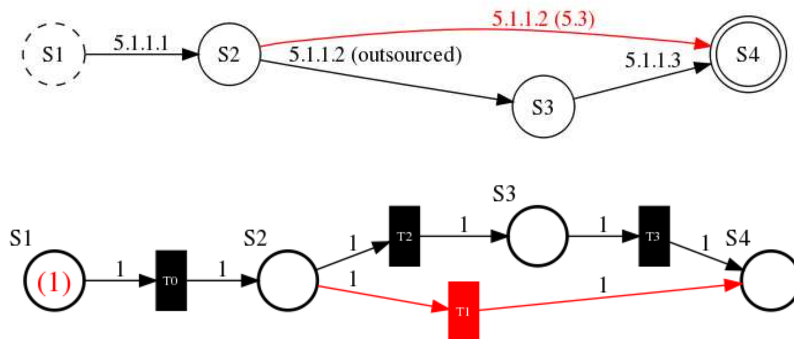

Figure 3. Activity 5.1.1 expressed as a SSM and as the corresponding Petri net. This is a much-simplified version of the actual process described in ISO/IEC 12207. The arcs correspond to sub-activities and are labelled accordingly.

A single token is present in the first place S1 of the Petri net, showing that the transition T0 corresponding to activity 5.1.1.1 is enabled. The token itself carries no meaning. The

semantics of the state of the system is dependent 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. The numbers on the inbound and outbound transitions of each activity are the number of consumed or produced tokens, respectively. In our case, this number is always one. 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.

The example diagrams were produced using a prototype built to validate the model, which uses automatic layout engines provided by open-source libraries, and usually produces different orderings for FSM and Petri nets. Colours were added manually to the diagrams to improve legibility.

4.1 Scalability

In Figure 3, activity 5.1.1.2 in red is meant to be executed using activity 5.3, which defines a development procedure inspired by the 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 5.1.1.2 in red 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 5.1.1.2, which would replace the corresponding transition in a more detailed view. The states S2 and S4 are retained, as they are the same for both the SSM in Figure 3 and Figure 4, fulfilling the conditions expressed in Section 2.3.

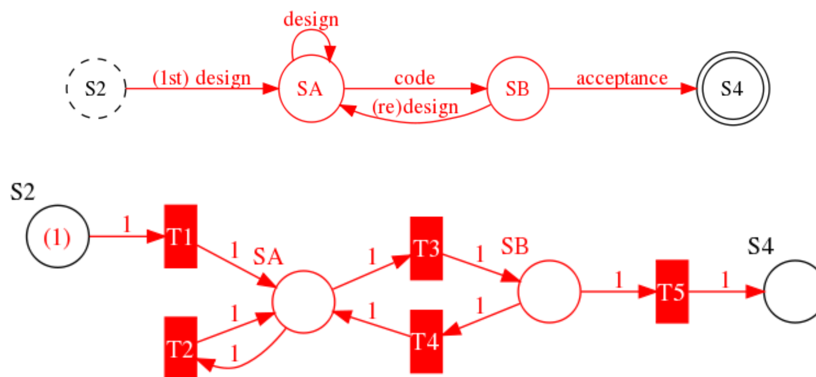

Figure 4. A possibility for activity 5.1.1.2 expressed with a higher level of detail, as a SSM and as the corresponding Petri net. States S2 and S4 are the same as the corresponding S2 and S4 in Figure 3.

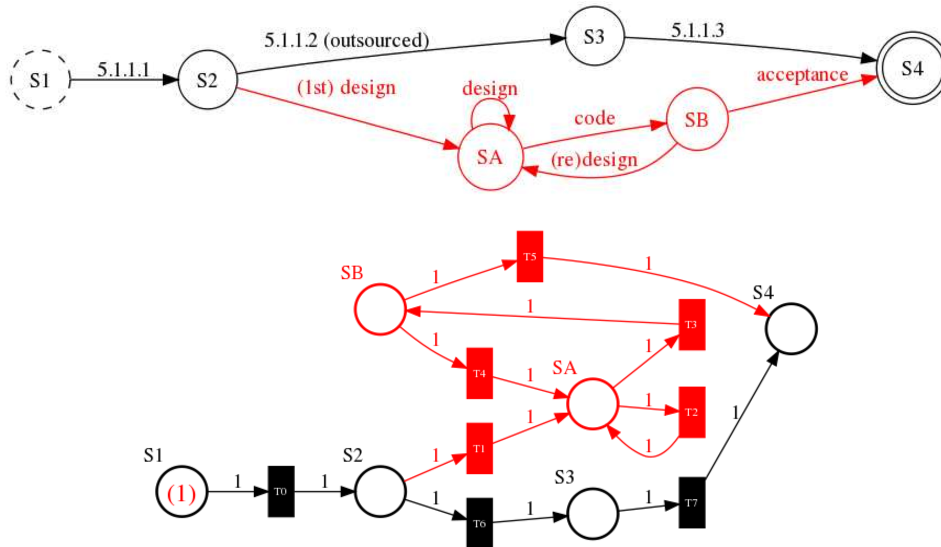

Figure 5. The whole state machine and corresponding Petri net when replacing the arrow (transition) in red in Figure 3 by the component state machine (Petri net) of Figure 4, also in red.

Finally, Figure 5 illustrates the state machine and Petri net obtained by such replacement, which is the essence of the ability to zoom on an arrow.

4.2 Synchronisation

To illustrate the mapping when synchronisation is used, a new activity "feasibility study" is added in parallel to activity 5.1.1.1. This activity must take place after state S1, and before state S2. Figure 6 illustrates the new SSM with the synchronisation, and its corresponding Petri net.

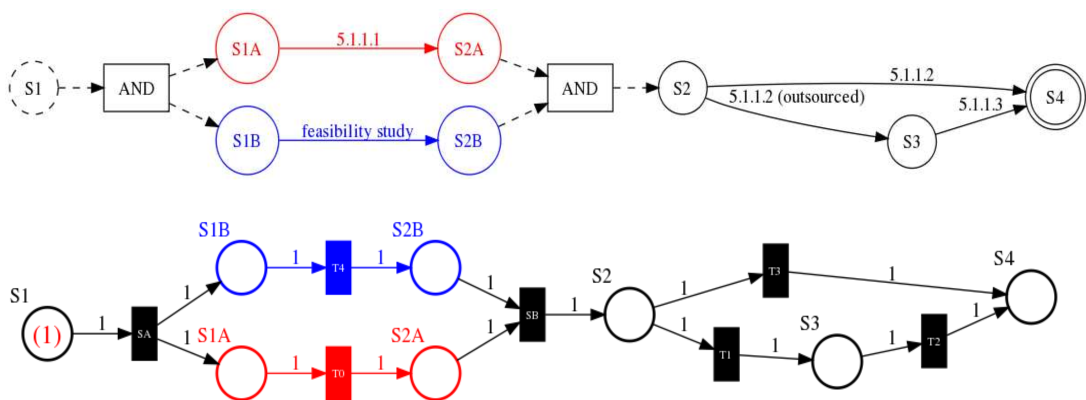

Figure 6. Activity 5.1.1 as a synchronised SSM, with the addition of a parallel activity "feasibility study" in blue and the synchronisation mechanism, and the corresponding Petri net

5. CONCLUSION

The model presented in this article is a first step towards bridging the gap between an intuitive representation for systems life cycle processes on one hand, and a formal representation suited for validation of such processes on the other hand.

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.

This method has been successfully applied to some basic processes inspired from the ISO 12207 document, but the main application seems to be beyond that of systems life cycle processes only. Indeed, the model was used to represent business processes representing legal and administrative documents (Coţofrei, 2008). The method needs to be refined as the mapping is not entirely straightfoward, but even the simple proof of concept using the prototype written to validate our idea showed very promising results. We were able to invalidate a lot of otherwise accepted administrative processes, and it seems that the mistakes found would have escaped traditional methods such as Pi calculus.

Most mistakes were actually directly visible using only the representation of the state machines instead of the business process modelling language diagrams themselves. Some very important properties, like the generation of useless documents, potentially endless cycles or contradictions were visible with a simple graphical simulation of the Petri net.

Future work is necessary in the actual analyses that can be done on the generated Petri nets, most algorithms used to analyse systems properties in this context (reachability, liveliness, boundedness, cycle detections etc.) being at least exponential, unless the graphs are constrained in some ways. A complete analysis of the properties of the generated graphs and state diagrams needs to be conducted to show the applicability of such methods.

ACKNOWLEDGEMENT

This Work was sponsored 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.

Carroll, J. et al,1989. *Theory of Finite Automata with an Introduction to Formal Languages.* Prentice Hall, Englewood Cliffs.

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

Gill, A., 1962. *Introduction to the Theory of Finite-state Machines.* McGraw-Hill.

Ginsburg, S., 1962. *An Introduction to Mathematical Machine Theory.* Addison-Wesley.

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.