SCALABLE SOCIAL PROTOCOLS TO FORMALIZE SYSTEMS DEVELOPMENT LIFE CYCLES

Eric Simon
University of Neuchatel
Pierre-a-Mazel 7, 2000 Neuchatel, Switzerland

Christophe Künzi
University of Neuchatel
Pierre-a-Mazel 7, 2000 Neuchatel, Switzerland

Kilian Stoffel
University of Neuchatel
Pierre-a-Mazel 7, 2000 Neuchatel, Switzerland

ABSTRACT

There exist many documents outlining methodologies for Systems Development Life Cycles. Although these documents are well adapted for human beings to follow, there is no system supporting the inherent complexity of the involved processes. In this paper, we present scalable social protocols, a recursive extension of social protocols, a formalism for representing human-to-human interactions. We argue that such a recursive definition allows working on a representation of the existing standards at different levels of detail. The resulting flexibility is essential to model real and complex systems and, ultimately, to implement dynamic and highly specialized life cycles.

KEYWORDS

systems development life cycle, scalable social protocols, information systems development

1. INTRODUCTION

Many attempts have been made at formulating methodologies to be able to develop increasingly more complex systems in an organized and reliable way. Most of the methodologies in use nowadays, in particular in the context of developing Information Systems and their main technological components, Software Systems, are to some degree anchored in Systems Thinking, a field of Systems Dynamics founded in 1956 by MIT professor Jay Forrester for testing new ideas about social systems.

The first relevant model for software development, and by extension systems development, was the Waterfall Model, proposed by Royce (1970). It consists in a perfectly ordered sequence of seven phases, each dependent upon completion of the preceding one, as shown in Figure 1.

Figure 1. The seven phases of the Waterfall Model as proposed by Royce.

Nowadays, it is argued that the model and its underlying philosophy of Big Design Up Front, while having undeniable advantages for complex systems, like for instance its emphasis on documentation, is suitable only for projects which are stable, i.e. don't have changing requirements.

There exist modified versions of the Waterfall Model. Royce himself attempted to improve his initial model and proposed a "final model", in which he added feedback from code testing to design and from design back to requirements.

Later on, confronted with the limits of the Waterfall Model, people tried to formulate a model that would take into account the iterative nature of software or systems development and could accommodate both top-down (Mills, 1971) and bottom-up approaches, thus taking into account the inevitable changes in requirements during a system's life cycle. Iterative Development means developing in phases starting with a design goal and ending with the client reviewing the progress. In the mid-eighties, Boehm published an article about the Spiral Model, not the first to discuss iterative development in this context, but the first to explain why iteration matters (Boehm, 1986). The Spiral Model was adopted by the US Military for its ability to handle large, complex and critical systems.

A last attempt was made to further extend the Waterfall and Spiral models led to the Chaos Model, defined by Raccoon (1995). The idea is to apply all phases of the life cycle to all levels of a project. Hence, for example, individual lines of code are treated as a level requiring design, implementation and integration, and so are functions, modules, systems and finally the whole project.

Also during the eighties, at IBM, James Martin developed another approach called Rapid Application Development he formalized in a book (Martin, 1991). The focus was on delivering quality as fast as possible. The speed is achieved through the use of Computer Aided Software Engineering (CASE) tools, and the quality through the early involvement of users in the analysis and design phases. The approach allows delivering an early prototype, usually reduced in features, and adding features incrementally later on.

Rapid Application Development consists in six core elements:

1. *Prototyping*: developing a feature-light application in a short amount of time.
2. *Iterative development*: adding features in short life cycles, feeding the new user requirements to the next release.
3. *Time boxing*: supporting iterative development by pushing off features to future versions.
4. *Team members*: emphasizes that teams should be small and composed of experienced members.
5. *Management approach*: management is involved in keeping life cycles short and enforcing deadlines.
6. *RAD tools*: development speed is more important than the cost of tools, so one should use the latest technologies.

The major drawback of this approach is the reduced scalability of the resulting systems, consequence of the continuous enhancement of an early imperfect prototype.

During the nineties, moving further away from the plan-driven and predictive Waterfall Model towards more adaptive methods, people defined a variety of methods grouped since 2001 under the general framework denomination of Agile Software Development (Beck, 2001). The basic idea was to refine iterative methods by reducing dramatically the time between releases, or milestones, now measured in weeks or even days instead of months. Another common denominator of these methods is that work is performed in a highly collaborative manner. Some of the well-known Agile Software Development Methods include Extreme Programming (Beck, 1999), Scrum (Takeuchi, 1986), Agile Modeling, Adaptive Software Development (Highsmith, 2000), Crystal Methodologies, Dynamic Systems Development Method (DSDM Consortium), Feature Driven Development (Palmer, 2002), Lean Software Development (Poppendieck, 2003), Agile Unified Process. The same approach has been applied to the documentation and data life cycles.

All these models share some common phases, or building blocks, at different levels of granularity. For the remaining of this document, we chose to study primarily the Waterfall Model. Despite the limitations mentioned at the beginning of this section, it has the advantage of including most building blocks of subsequently developed systems in a simple linear chain of processes. In the remainder of this paper, we present an approach to the problem of formalizing Systems Development Life Cycles (SDLC) in order to implement a system supporting it. In Section 2, we present what exists so far in this respect. Because of the nature of the considered processes, we chose to use social protocols (Picard, 2006) as the formalism. Section 3 presents briefly what social protocols are. In Section 4, we present the application of social protocols to the formalization of SDLC, and present an extension of the formalism to facilitate its application to the problem at hand in real applications. This allows us to implement SDLC, although some problems remain to be solved, which we mention with possible solutions in the conclusion in Section 5.

2. THE SYSTEMS DEVELOPMENT LIFE CYCLE

Regardless of the criticisms of each of the methods mentioned in Section 1, it is not unusual to find one method used within another on another scale. For example, a developer might use the Waterfall Model on a very small scale at the level of the development of his module, while an Agile Software Development Method is applied by the team for the whole project. This shows the need for a clear definition of common blocks in all approaches that can be formalized and combined to create any new method, which is the long-term aim of this research. A first step towards this goal has already been completed in the context of administrations, notably by the US Department of Justice, who define the Systems Development Life Cycle (SDLC) (The US Department of Justice, 2003). It is again a systems approach to problem solving, made up of several phases, each comprised of multiple steps:

1. *Software concept*: identifying and defining the need for a new system.
2. *Requirement analysis*: analyzing the information needs of the end users.
3. *Architectural design*: creating a specification for hardware, software, data resources, but also people.
4. *Coding and debugging*: creating the final system.
5. *Testing*: evaluating the functionalities.

There are ten official phases described in the document of the Department of Justice: Initiation, System Concept Development, Planning, Requirement Analysis, Design, Development, Integration and Test, Implementation, Operations and Maintenance, Disposition.

The Software Development Life Cycle also specifies which documentation shall be generated during each phase. It is intended for humans to follow though, and in order to have a system able to represent and validate such models, we need a good formalization with appropriate state of the art knowledge representation techniques, something we couldn't find explicitly.

Following today's call for standards and best practices in software and systems development, the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) established a joint technical committee in 1987 with the scope of standardization in the field of information technology systems. This committee initiated the development of an International Standard, ISO/IEC 12207 on software life cycles (ISO/IEC 12207, 1995), that led to its IEEE/EIA implementation for industry in 1998 (IEEE/EIA 12207.0-1996, 1998). The standard establishes a top-level architecture of the life cycle of software. The life cycle begins with an idea or a need that can be satisfied wholly or partly by software and ends with the retirement of the software, as in SDLC. The architecture is built with a set of processes and interrelationships among them. The processes are modular, minimally coupled, and the strict responsibility of a party in the software life cycle.

This standard is meant to be tuned for specific projects and followed by humans, typically using check lists for the validation of the various phases. It is, however, the closest document to a formal definition we could find, and it is the starting point we chose to model the development methodologies mentioned in Section 1, starting with the Waterfall Model.

3. SOCIAL PROTOCOLS

As mentioned in Section 1, systems development methodologies are anchored in Systems Thinking and Systems Dynamics, a field strongly influenced by sociology. Therefore, it seemed natural to use a formalism able among other things to accommodate human to human interactions. Professor Picard proposed such a formalism, called social protocols (Picard, 2006). It is important to mention here that Professor Picard's social protocols have nothing in common with the notion of social protocol defined by Reagle in the context of the W3C (Reagle, 1997). Although this approach is relatively new and it is not yet clear whether it is more expressive than established modeling techniques such as Conceptual Graphs or Petri Nets, we decided to investigate our problem with it for two reasons: first, its simplicity, and second, its support for human-to-human interactions. It also allows a form of negotiation, a very important aspect of systems development and team work in general. We mention preliminary results in this respect in Section 4.5.

The social protocols as proposed by Professor Picard provide a formalism to model the interactions between humans. With this tool we have a flexible and adaptive approach. The definition of a social protocol consists in the following concepts:

- **A Role**: a role *r* is a label. The set of all roles is *R*.
- **An Action**: an action *a* is an execution of a software entity. The set of actions is written *A*. In this representation a human plays a role and does some actions, so a role and an action are combined to form a group called a behavioral unit. Formally $BU = R \times A$
- **A State**: a state *s* is a label associated with a given situation in a collaborative process. Let's denote *S* the set of states.

The idea is to combine a starting (source) state and an ending (destination) state with a behavioral unit: $(bu, S^{source}, S^{destination})$. This defines a **Transition**, and the set of all transitions together with the states constitute the social protocol. The collaborators are moving from state to state via the execution of behavioral units. But one has to be careful that a behavioral unit may only be executed by a collaborator labeled with the appropriate role.

Another important part of the social protocols is the ability to add a **desirability function** Δ_p. This function is always between 0 and 1. The higher the value of the desirability function for a transition, the higher the desirability of this transition for the group. For example, in a project, the transitions leading to success are more desirable than the transitions leading to failure.

An interesting point in the social protocols is the ability to change the desirability function or to change the structure of a model by **negotiation**. This is necessary because human systems are changing and adapting themselves. By experience we may observe that a new transition is needed or that some transitions are less "desirable". This flexible property is likely to be central to solving our problem, especially when it comes to adapting existing models to the needs of a specific project, and to adapting the needs on the go during the development of a system. This, however, is beyond the scope of this paper (see Section 5).

4. SCALABLE SOCIAL PROTOCOLS FOR SDLC

In this section, we present a case study of the application of social protocols to a Systems Development Life Cycle, using the Waterfall Model as the basis for our work. First, we propose an extension of the social protocols we call "scalable social protocols", that allows for recursion, accommodating the drill-down from large processes to smaller components without having to resort to graph theory or formally complex solutions. The goal is to provide an application supporting the processes of systems development, which calls for a pragmatic approach rather than a complete formal solution. Then we present briefly the advantages of our approach in the context of Systems Development Life Cycles. Finally, we tackle the problems that our choice raises concerning parallel processes and propose another simple solution to this problem.

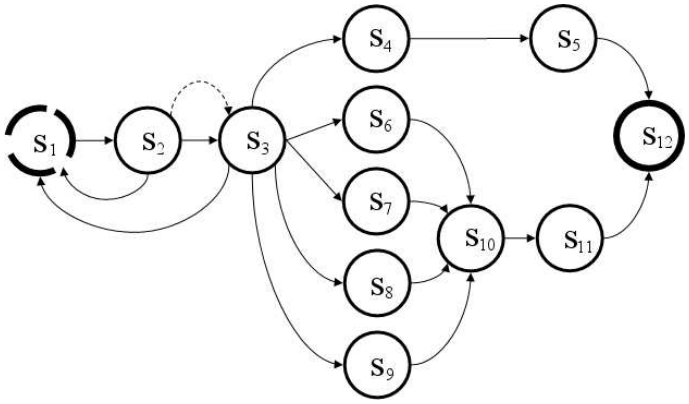
4.1 Extension of the Formal Model

The modeling of a particular process as a scalable social protocol depends on the level of detail we need. For instance, some partners of a project want to see a transition like a black box, while other partners have to deal at a much higher level of detail. In our example, taken from the IEEE/EIA 12207 document (IEEE/EIA 12207.0-1996, 1998), the initiation phase is the responsibility of a role called "Acquirer". A social protocol diagram of this phase seem as a black box for a manager is show in Figure 2.

Source State	Destination State	Role	Action
S1	S2	Acquirer	Do the acquisition (5.1.1)

Figure 2. Initiation 5.1.1 shown at lowest level of detail.

On the other hand, the role "Acquirer" for example, will want to see this particular process at a much higher level of detail, as in Figure 3.



Source State	Destination State	Role	Action
S1	S2	Acquirer	Describe the concept.
S2	S1	Acquirer	Concept is badly described.
S2	S3	Acquirer	Define and analyze requirements.
S2	S3	Supplier	Define and analyze requirements.
S3	S1	Acquirer	Requirement is badly described.
S3	S4	Acquirer	Purchase an off-the shelf soft.
S3	S6	Acquirer	Develop the software.
S3	S7	Acquirer	Develop the software through contract.
S3	S8	Acquirer	Enhance an existing system product.
S3	S9	Acquirer	Combination of s6, s7, s8.
S4	S5	Acquirer	Satisfy 4 conditions.
S6	S10	Acquirer	Prepare the acquisition plan.
S7	S10	Acquirer	Prepare the acquisition plan.
S8	S10	Acquirer	Prepare the acquisition plan.
S9	S10	Acquirer	Prepare the acquisition plan.
S10	S11	Acquirer	Define the acceptance strategy.
S5	S12	Acquirer	Finish the acquisition.
S11	S12	Acquirer	Finish the acquisition.

Figure 3. Initiation 5.1.1 shown at higher level of detail.

The formal definition of a social protocol as given in Section 3 defines an action as the execution of a software entity. We extend the definition of action to include another social protocol Σ, leading to a recursive definition that has the desirable properties for our formalization, namely the ability to represent processes at different levels of detail. Of course, merely saying that an action can be either atomic or another social protocol is not sufficient.

First and foremost, an action A has exactly one source state S_A^{source} and one destination state $S_A^{destination}$. For an action to be seen as a more complex process, in particular a social protocol Σ, the source and destination states of A must be identical to the source and destination states of Σ. Formally:

$$S_\Sigma^{source} = S_A^{source} \wedge S_\Sigma^{destination} = S_A^{destination}$$

There follows that the destination state $S_A^{destination}$ must be unique for Σ. Hence, we define a **scalable social protocol** Σ as a social protocol satisfying the conditions above.

Furthermore, a behavioral unit *bu* being defined as a pair *(role, action)* and being the cornerstone of the definition of a social protocol, the role R involved in a particular transition $((R, A), S^{source}, S^{destination})$ has to be the same as the overall role of the social protocol Σ. In a purely pragmatic approach such as ours, the overall role can be defined according to the need of the domain one is modeling, while remaining consistent with the definition above. For instance, in SDLC, we use a loose definition:

The **overall role** of a scalable social protocol is the major role involved in the transition leading from the set of states at step $n - 1$: $\{S_{n-1}\}$ to its (unique) destination state S_n. Indeed, our research indicated that the significant role for most validation tasks involved in SDLC is the one that allows reaching the destination and proceeding further. One could take another approach, defining the overall role of an actionable social protocol to be the role involved in the majority of transitions of the protocol, or defining a super-role as a parent class of all the roles involved, if this conveys meaning for the domain.

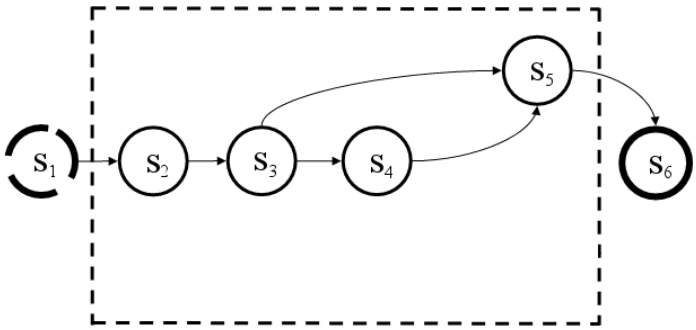
This recursive property allows us to work with social protocols and scalable social protocols in a natural fashion to model SDLC. In the next two sections, we present the advantages of this definition in this context.

4.2 Views

The development of a system is conducted independently between the different departments. In fact the manner one sees the system depends on the responsibility one has in the project. So naturally the development can be seen like a system of parallel processes. We have in fact several processes that are evolving independently of the others. But this implies some constraints. First and foremost, of course, one has to provide some simple mechanisms to synchronize the different processes in our system. (see Section 4.4). This leads to the concept of views, each view being seen as a different social protocol. Without this ability to structure the system into more or less independent subsystems, the burden to ensure that everyone in the project agrees on each single transition would mean that every single actor has to be aware of the entire system.

4.3 Building Blocks

Despite the flexibility given by views, our model is very static and isn't easy to change. We want to let the user of our model be able to adapt a life cycle to his particular needs, or to adapt it during the course of the project. But for the user the construction of a life cycle should be more intuitive than creating new states with no meaning for him at a much too high level of detail. A solution is to provide the user with the ability to combine different parts of a social protocol in blocks that the user can use to build his own life cycle. Every block should have a source state and a destination state, so it can be seen like a black box. The definition of a scalable social protocol given in Section 4.1 allows just that. For example, in the IEEE/EIA 12207 document, the initiation part 7.1.1 is always needed independently of the kind of process we want to initiate. So we can provide the user with a block that he can plug in when needed, as shown in Figure 4.



Source State	Destination State	Role	Action
S1	S2	Manager	Establish the requirements.
S2	S3	Manager	Establish the feasibility.
S3	S4	Manager	Modification of the requirements.
S3	S5	Manager	Initiation of the process is finished.
S4	S5	Manager	Initiation of the process is finished.
S5	S6	Manager	Go to the planning process.

Figure 4. Initiation 7.1.1, a process common to other tasks, is shown as a scalable social protocol, i.e. a building block.

4.4 Synchronization

We argued in the previous sections that social protocols are well adapted to the task of formalizing the processes of the Systems Development Life Cycle. One problem remains though before one can use such formalisms in a real system: the model doesn't allow for parallel execution of processes and synchronization. Since social protocols are in essence Finite State Automata, themselves a special case of Petri Nets, one could probably stretch the definition of social protocols to Petri Nets structures. While this is certainly a direction to explore in the general case of applying social protocols in the context of complex systems, for our particular problem it seems quite an overkill in formalism. Instead, we propose a synchronization mechanism that is completely independent of the formalism of social protocols itself.

The idea is that independent processes can take place at any given time separately, but at some point in the overall process synchronization is ensured by a "rendez-vous", as in simple parallelism. In essence, some subset of source states S2 from the overall process can be considered if, and only if, another subset of destination states S1 has been reached. It is in fact a simple logical AND between destination states, as shown in Figure 5. This type of synchronization is sufficient for linking sub-processes in a partial order, as is the case for the SDLC as described in the IEEE/EIA 12207s document (ISO/IEC 12207, 1995).

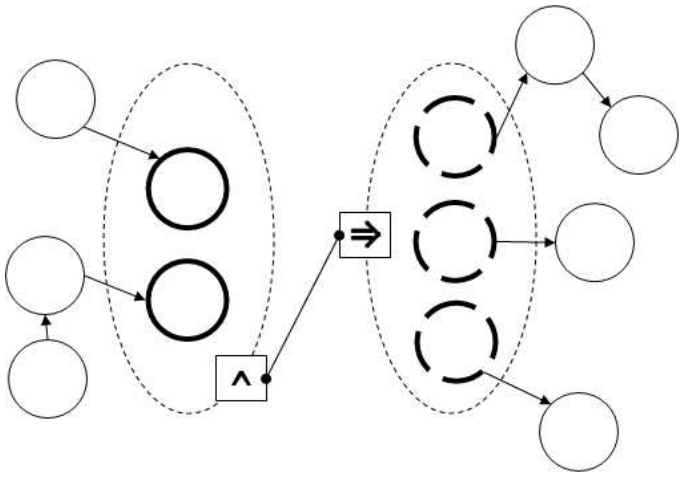


Figure 5. A simple mechanism of synchronization, independent of the formalism itself, is quite sufficient for the task at hand: the initial states on the right-hand side of the implication can be considered only when both the destination states on the left-hand side are reached, which is shown by the ^ symbol.

4.5 Desirability and Negotiation

So far, we have mostly ignored the desirability function and the negotiation, which nevertheless play a central part in social protocols.

Some transitions in the process of systems development or maintenance are optional, or are likely to be ignored if the project is in a crisis. To model this, one has to take into account some probabilistic approach, which is exactly what the desirability function provides in this context. On the other hand, systems development is highly dynamic. A transition having to do with extended documentation could suddenly become much less important in regard to development when the project approaches a beta release, for example. The negotiation of the desirability function allows the model to adapt to changes in its environment. Finally, negotiation of the structure itself can take place before or during a project, to adapt the model to one's particular needs. This idea of being able to tune the processes of SDLC is central to the application of the methodology to real projects and is stated explicitly in the IEEE/EIA 12207 document (ISO/IEC 12207, 1995). The very idea of building blocks as we described it in Section 4.3 becomes almost worthless without the negotiation mechanism.

5. CONCLUSION

In this paper, we proposed an extension of social protocols with recursive properties we called scalable social protocols, and showed how it was applied to the problem of formalizing Software Development Life Cycles. The Waterfall model was used as an example, and the processes were modeled using the IEEE/EIA 12207 document as reference. The recursive nature of scalable social protocols allows an elegant formalization of processes at different levels of detail (drill-down), for different roles (views) and to build composite processes from simpler building blocks. Furthermore, we solved the problems of formalizing multiple concurrent processes and their synchronization by a pragmatic approach suitable for this particular domain.

While our approach was clearly pragmatic, a more complete formal solution, using Petri Nets to handle parallel processes for example, will have to be investigated and compared to our current proposal. While some work has been done regarding the cornerstone of social protocols, the desirability function and the resulting negotiation, these properties will have to be validated by a real prototype and human users. The formalization of SDLC in this context is a first step towards a complete implementation. Another aspect that will have to be taken into account in this domain where human beings play a central role is uncertainty and incompleteness. We believe that the desirability function could provide the necessary functionalities in this respect, although it will also have to be validated by a real application.

ACKNOWLEDGEMENT

This work is supported by the Hasler Foundation, project ManCOM 2085 and the Swiss National Science Fundation, project 200021-103551/1.

REFERENCES

Beck, K., 1999. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, Boston, MA, ISBN 0321278658.

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

Boehm, B., 1986. A spiral model of software development and enhancement. *In ACM SIGSOFT Software Engineering Notes*, Vol. 11.

DSDM Consortium. *Dynamic Systems Development Method*. http://www.dsdm.org/.

Highsmith, J.A., 2000. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House, New York, ISBN 0932633404.

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

Martin, J., 1991. *Rapid Application Development.* Macmillan Coll. Div., ISBN 0023767758.

Mills, H., 1971. *Top-down programming in large systems, debugging techniques in large systems*.

Palmer, S.R. and Felsing, J.M., 2002. *A Practical Guide to Feature-Driven Development.* Prentice Hall. ISBN 0130676152.

Picard, W., 2006. Computer support for adaptive human collaboration with negotiable social protocols. *In Abramowicz, W., Mayr, H.C., eds.: BIS*, Vol. 85 of LNI., GI, pp. 90–101.

Poppendieck, M. and Poppendieck, T., 2003. *Lean Software Development: An Agile Toolkit for Software Development Managers. 1st edn.* Addison-Wesley Professional, ISBN 0321150783.

Raccoon, L.B.S., 1995. The chaos model and the chaos life cycle. *In ACM Software Engineering Notes*, Vol. 20, No. 1, pp. 55–66.

Reagle, J. Jr. and Faith Cranor, L., 1997. Designing a Social Protocol: Lessons Learned from the Platform for Privacy Preferences Project. *In Proceedings of the Telecommunications Policy Research Conference.* Alexandria, VA.

Royce, W.W., 1970. Managing the development of large software systems. *In IEEE WESCON*, pp. 1–9.

Takeuchi, H. and Nonaka, I., 1986. The New New Product Development Game. *In Harvard Business Review*, Vol. 64, No. l, pp. 137–146.

The US Department of Justice, 2003. *Systems Development Life Cycle Guidance Document*. http://www.usdoj.gov/jmd/irm/lifecycle/table.htm.