

Petri Nets and Software Engineering

Giovanni Denaro and Mauro Pezzè

Università degli Studi di Milano-Bicocca,
Dipartimento di Informatica Sistemistica e Comunicazione,
I-20126 Milano, Italy
`{denaro,pezzè}@disco.unimib.it`

Abstract. Software engineering and Petri net theory are disciplines of different nature. Research on software engineering focuses on a problem domain, i.e., the development of complex software systems, and tries to find a coherent set of solutions to cope with the different aspects of the problem, while research on Petri nets investigates applications and properties of a specific model (Petri nets).

When Petri nets can solve some problems of software development, the two disciplines meet with mutual benefits: software engineers may find useful solutions, while Petri net experts may find new stimuli and challenges in their domain.

Petri nets and software engineering have similar age: Karl Adam Petri wrote his thesis in 1962, while the term “software engineering” was coined in 1968 at a NATO conference held in Germany. The two disciplines met several times in the past forty years with alternate fortune. Presently, software engineering and Petri nets do not find many meeting points, as witnessed by the scarce references to Petri nets in software engineering journals and conferences and vice versa, but software engineering is facing many new challenges and the Petri net body of knowledge is extending with new results.

This paper attempts to illustrate the many dimensions of software engineering, to point at some aspects of Petri nets that have been or can be exploited to solve software engineering problems, and to identify new software engineering challenges that may be solved with Petri net results. This paper does not have the ambition of completely surveying either discipline, but hopes to help scientists and practitioners in identifying interesting areas where software engineers and Petri net experts can fruitfully collaborate¹.

1 Introduction

Software engineering presents several problems that can be attacked with many different techniques and methodologies. Software engineers do not focus on a particular technique or model to solve all problems, but select the solutions that best fit the requirements for each different problem and context. Solutions that

¹ This work has been partially funded by the European Union through the EU IST project SegraVis.

are excellent in a specific context and at a given time, may be sub-optimal in other domains, may not suite well other problems, or may become obsolete in other moments. Software specification and design are typical examples: structure analysis based solutions that were very popular in the eighties, became less and less popular in the nineties, and are now substituted by object oriented based solution; client-server solutions that may solve well many classes of problems, may be ignored in contexts that benefit from other equally good solutions. A quick scan of software engineering handbooks, conferences, and journals would clearly give a variegate picture from the methods and techniques viewpoint.

Disciplines like software engineering that focus on problems and search for the best solutions regardless of the underlying methods or techniques can be identified as *problem-oriented* disciplines. The main characteristic of these disciplines is the presence of many complex problems and the co-existence of alternative solutions, none of which optimal per se. Problem-oriented disciplines are eclectic, since problems may be solved in many different ways with radically different techniques, and fickle, since techniques can be adopted and abandoned as the field evolves [1].

Conversely, the research on Petri nets focuses on a “solution”: Petri nets. The research on Petri nets is not driven by a problem domain that asks for successful solutions, but is rather driven by a theory that is studied for solving problems of different nature. Research on Petri nets investigates the various possibilities presented by the theory, and proposes the theory to solve problems in different domains. Advances in Petri nets can be used for attacking problems in computer science, chemistry, biology, hardware design, software specification, distributed computing, multimedia and so forth. Disciplines like Petri net research that focus on theory and offer it for different application domains can be identified as *solution-oriented* disciplines. Solution-oriented disciplines are homogeneous and have a well-defined theory and a stable set of tools.

The meeting of problem and solution-oriented disciplines may bring enormous benefits to both fields: problem-oriented disciplines may find efficient solutions to key problems, while solution-oriented disciplines may find new stimuli in the field. Unfortunately meeting of different disciplines is difficult: few scientists understand different fields well enough to be able to see the potentialities for cross fertilization, and blind attempts to investigate new fields to search for novel solutions are often frustrated by skepticism and lack of successes. However, when problem- and solution-oriented disciplines meet, the whole scientific community can benefit from scientific and technological progresses. This is happening for example in the meeting of biology and research on algorithms that is opening enormous opportunities in bioinformatics.

Software engineering and Petri nets met several time in the past and the meeting seeded interesting ideas in both fields. Useful applications of Petri nets have been proposed in requirement engineering (e.g., [2]), reverse engineering (e.g., [3, 4]), design of user interfaces (e.g., [5]), modeling and analysis of safety critical systems (e.g., [6]), distributed systems (e.g., [7, 8]), real time systems (e.g., [9–12]), multimedia systems (e.g., [13–17]), software process management

(e.g., [18,19]), and software performance evaluation (e.g., [20]). However, the cross fertilization has never stabilized as the two fields are passing a period of scarce communication.

Software engineering is characterized by many dimensions that assume different relevance from different perspectives and are difficult to summarize and frame. Figure 1 suggests three main dimensions: *product development*, *process support* and *application domain*. Software engineers must find an adequate process support to fit the different characteristics of the product for the specific application domain. Each dimension includes many elements with mutually dependent choices hard to concert in a successful project.

The first dimension considered in the figure is related to the development of products, i.e., concerns with the development of software, and is characterized by the specific aspects of the software system, the development phases and the activities performed during development, and the involved stakeholders.

The *aspects* of software systems are the relations among components of the system from different perspectives. They include *structure* and *architecture*, i.e., relations among components, *functions*, i.e., relations among values, *behavior*, i.e., relations among processes over time, and *non-functional properties*, i.e., relations between the system and its environment. The distinct aspects can be instantiated in many ways, but instantiations are not independent: the system structure may strongly impact on the behavior of the system, which may impact on non-functional properties or functions, and so on. For example, a pipeline architecture may limit concurrency that may result in low performances. Thus, factoring aspects independently can be hard and not always possible.

Software development includes different *phases* that span from *requirements analysis* and *specification* to *design*, *implementation* and *test*. Each phase copes with specific problems at distinct abstraction levels, and uses suitable tools and techniques. Phases are not independent. Many activities performed in different phases overlap and influence each other. The distinction of phases over time, as postulated by the waterfall model, is merely conventional and does not reflect the complex intertwining among phases, which characterize real life processes. The real situation is better represented for example by the process model shown in Figure 2, which captures the effort allocation among phases and process iterations. The vertical slices of the figure show how effort is concurrently allocated to different phases. Each vertical slice corresponds to a process iteration that involves all phases. Each process iteration produces a complete version of the software that improves the former version: inception and elaboration iterations produce early prototypes, construction iterations produce beta versions and release candidates, transition iterations produces software evolutions.

Each development phase requires many *activities*, *analysis*, *abstraction*, *modeling*, *construction*, *refinement*, *documenting*, *testing*, *comprehension*, *refactoring*, *reverse engineering*, etc... System activities must adapt to the different phases and may require different tools and techniques, depending on the phase in which they are performed and the aspects of the developed system. For example, modern modeling methodologies offer many different models suited to distinct de-

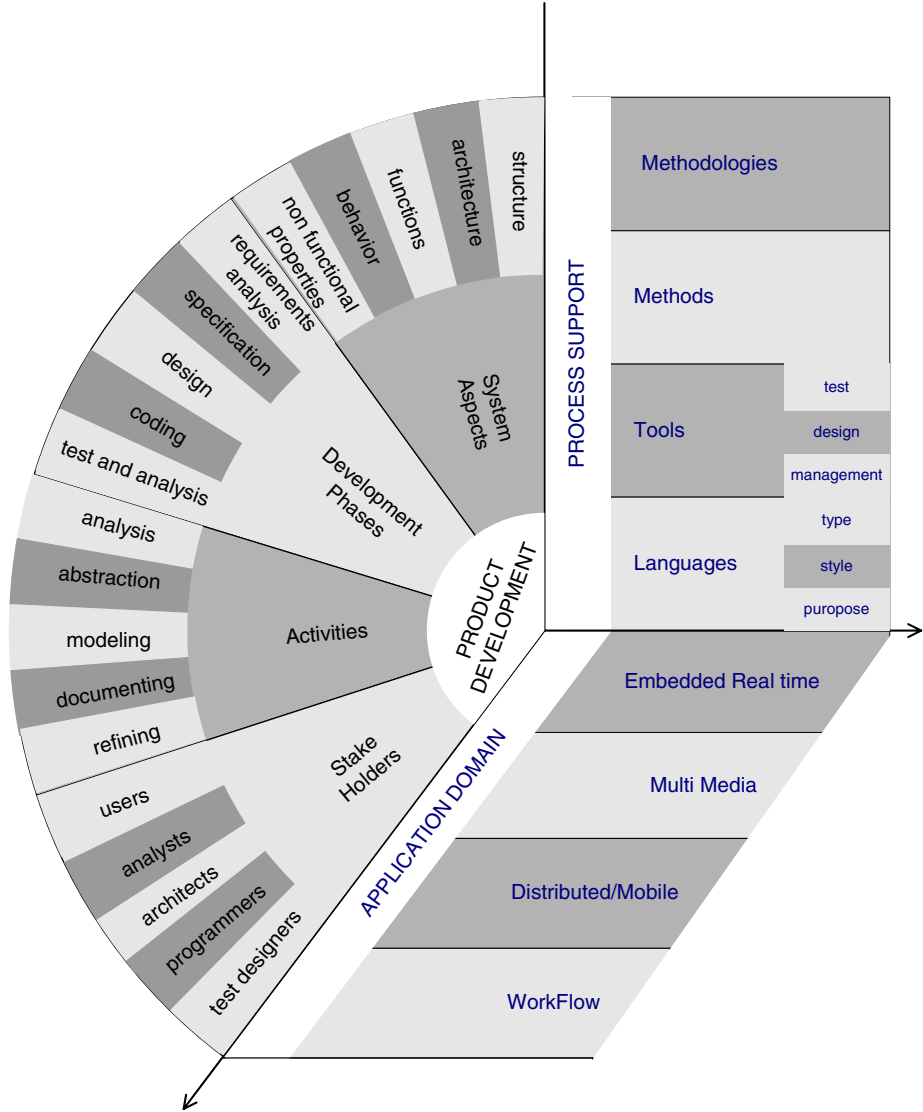


Fig. 1. Software engineering dimensions

velopment stages and to different aspects (Figure 4 at page 447 illustrates this issue in the case of the Unified Modeling Language.)

Software development involves many *stakeholders* who are involved with different roles, speak different languages, have different expectations, and focus on different problems: *users*, *analysts*, *software architects*, *developers*, *test designers*, *managers*, *marketing analysts*, etc... Software engineering must cope with the different needs and must provide a suitable means to coordinate stakehold-

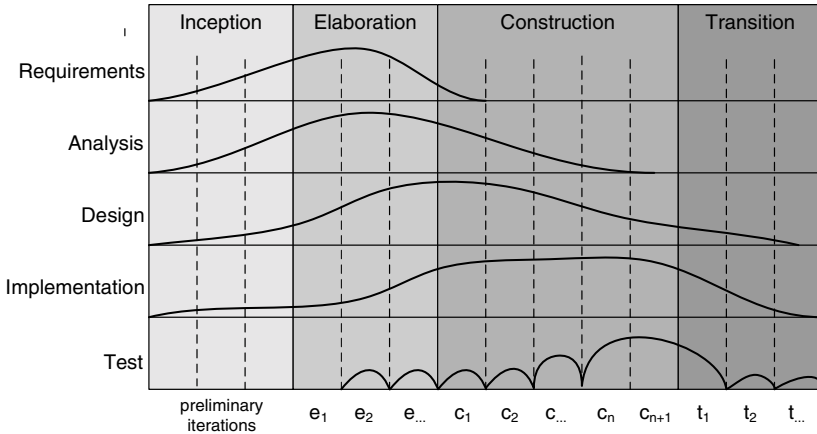


Fig. 2. The unified process model

ers. Lack of communication and comprehension among stakeholders can impact on the overall costs and even on the success of the whole project. For example, difficulties of analysts to understand the user language can lead to ill-designed requirements, while difficulties of users to read design models can lead to poor validation in the early stages, resulting in a final system that does not meet the user requirements. Expertise and needs of stakeholders impact on activities, on phases, and on the way system aspects influence the overall process. For example, familiarity of users with specific notations may influence the organization of system analysis and validation, the presence of an independent quality team may impact on the planning of activities and phases, the lack of familiarity with the application domain and the programming languages required by the user may require specific training, and so forth.

The second dimension considered in the figure is the software process: a suitable blend of methodologies, languages and tools that support the activities of the stakeholders through the development phases of the different aspects of the system.

Software development may involve many *languages* at different stages of development, e.g., specification, design, development, as well as within the same development phase, e.g., class diagrams, statecharts, interaction diagrams during specifications, or different programming languages for different subsystems. Languages involved in the development process are of different *type* (*operational*, *descriptive*, *executable*, etc...) and *style* (*textual*, *visual*, *diagrammatic*, *hybrid*, etc...). Languages affect other factors of the development process. For example, the presence of code generators or tools for analysis for a given language changes the effort required in specific phases, while strong user requirements, due for instance to requests of certification agencies or the legacy of the application, may impact on the organization of activities and roles.

Human activities are supported and complemented by many *tools*, which are responsible for shaping the process: *planning and monitoring, configuration management, design and specification, analysis, test case generators* tools are essential in mature development processes. Availability and functionalities offered by tools may determine choices of methodologies, activities, people and phases.

The third dimension considered in the figure concerns the application domains that emphasize different *software characteristics* that further impact on the development process. The development of *interactive, reactive, embedded, real-time, distributed, mobile, batch* systems may require different techniques, tools, languages, phases, and people.

Thus large variety of dimensions and choices that characterize problem oriented disciplines adds a critical dimension to the problems to be solved. Finding solutions to single problems is not sufficient: single techniques must be suitably blended within a general context and changes in one solution may affect many other solutions to different problems. For example, techniques for test and analysis may require different approaches to specification, design and coding, they may change the overall organization of the different development phases, they may require new skills and training, they may be based on new tools that may in turn impact on organization, methodologies, phases, and so forth.

Petri nets, as any solution-oriented discipline, cannot cope with all software engineering problems, but they can help for an unexpected variety of problems that involve all dimensions of software engineering. They can and have been successfully used during software development for modeling and analyzing behavior as well as non functional aspects both in the specification and design phases; They can support analysis, abstraction, modeling and documenting activities; They can provide a means for communication among users and analysts. They have been proposed as specification language for analysis as well as a means for modeling and enacting software processes. They have been used in many application domains that include real-time, workflow, multimedia, and distributed systems. Figure 3 summarizes the software engineering dimensions that can benefit from Petri nets.

Surveying either of the two disciplines would be impossible in the length of a single paper and it is out of the scope of this paper. Main goal of this paper is to illustrate how Petri nets can and have been used through the three outlined dimensions of software engineering, i.e., as models for software development, for describing and enacting software processes, and for solving problems in the specific domain of embedded real time systems.

2 The Role of Models in Software Engineering

The unexpectedly wide spectrum of applicability of Petri nets in software engineering derives from the central role of models in this discipline. Engineering software means describing and reasoning about the problem domain, the software solution, and the process evolution: analysts must capture the problem domain

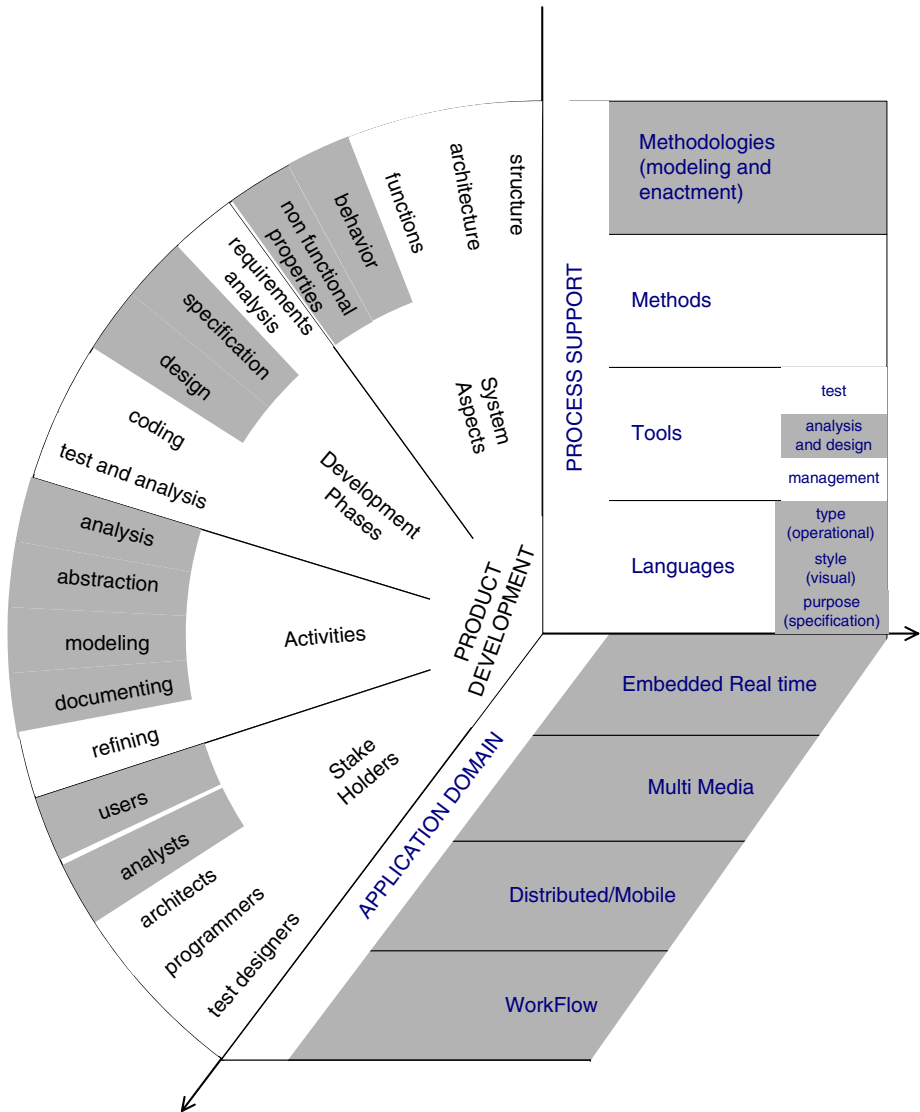


Fig. 3. Software engineering dimensions that can be supported by Petri nets are highlighted with grey background

to understand what has to be solved, designers must describe the system in terms of its architecture, programmers and test designers must understand the data and the control flow through the program, architects must capture the structure of systems to engineer and evolve applications, managers must build a cost model to plan and monitor the process.

Models are essential for communicating and reasoning about systems and must adapt to the people and the properties of interest. Models must capture the relevant system aspects in the different design phases; They must abstract from details that hide the overall picture; They must provide a common language for the different actors; They must support the analysis of the properties of interest. No model suites all phases, aspects, activities, stakeholders, and characteristics of software. The development of a single product usually requires the construction and analysis of many different models. The requirements of a good model depend from the goal of the model: models used for communication among people must be easily understandable for all involved specialists; models used for reasoning about properties must support efficient analysis of the target properties. For example, a detailed data flow model of a program can hardly be used for discussing software requirements or design strategies, but may be excellent for identifying anomalies in the code; conversely, use cases or interaction scenarios provide little help for analyzing program properties, but are often used to discuss the system requirements among software specialists, and between software specialists and domain experts.

During software development we need to discuss and reason about all aspects of the systems: structure, function, behavior, non-functional properties. These aspects cover a wide spectrum of elements, relations and views of the system. Capturing such a variety of elements with a single language requires enormous flexibility and generality that is hardly available in a single language. Universal languages, e.g., natural languages, provide such wide-spectrum coverage, but introduce ambiguities that reduce the possibility of analyzing properties. Modern methodologies, e.g. UML [21], are grounded on sets of complementary languages that cover different aspects for supporting communication and analysis of many aspects at different levels. Sets of languages help describing different aspects at different abstraction levels. In the case of UML, use case and sequence diagrams can be used in the early analysis phases to discuss early requirements with domain experts, class diagrams, collaboration diagrams and Statecharts can support modeling of behaviors during the detailed design of the system, component and deployment diagrams can model design and implementation details, as in Figure 4.

Models are used for communicating design decisions among different stakeholders. To this end, languages must be comprehensible to the involved people and must suite goals such as documentation, analysis, testing, early validation and problem understanding. For example, requirements analysis languages must provide a means for communication among analysts, users, architects, test designers, as well as software specialists, managers and marketing staff. Different attitudes and purposes inspired a large variety of languages that span from textual to visual and diagrammatic, from informal to formal, from detailed to abstract, etc...

Models are used for systems with different characteristics and requirements, e.g., interactive, reactive, embedded, real-time, control, workflow, distributed, mobile, multimedia, web-based systems. Different languages provide means for

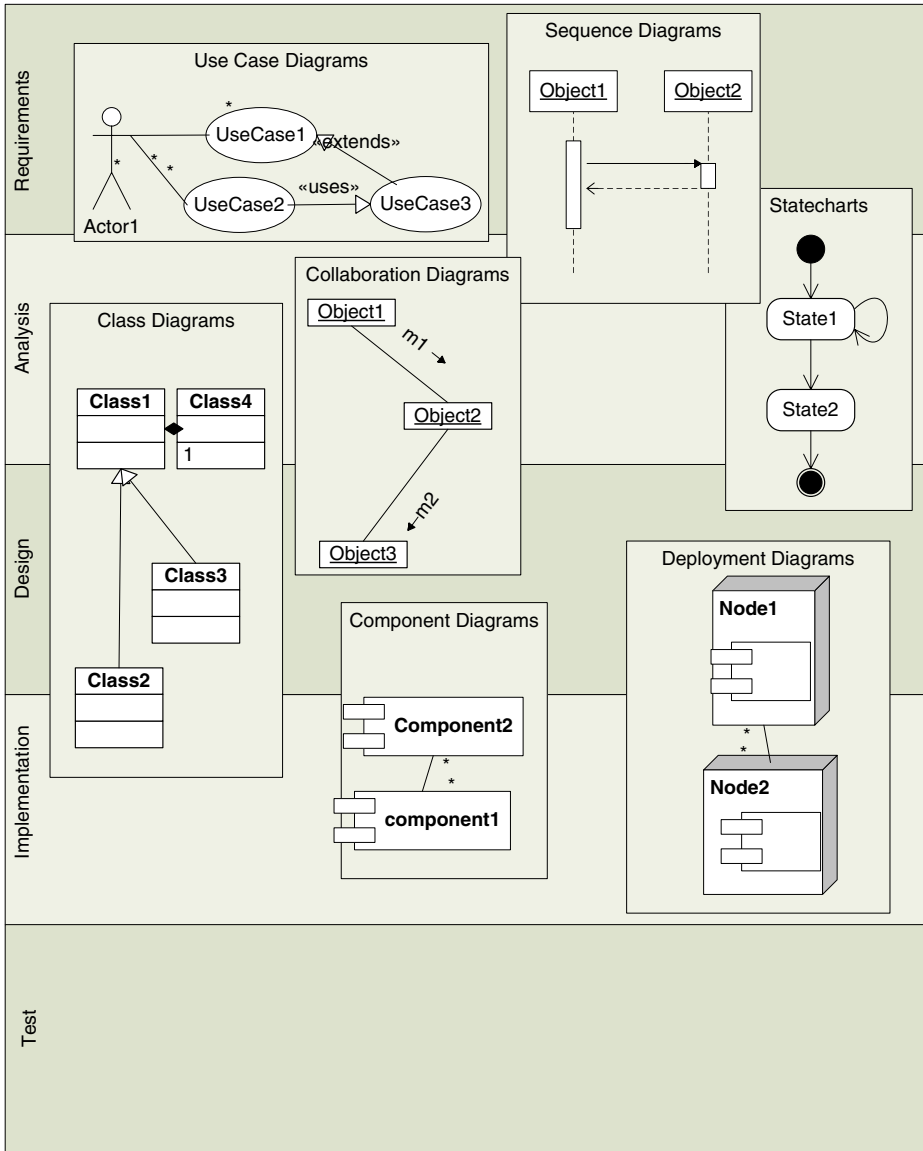


Fig. 4. Support of UML diagram to the software process

dealing with distinct characteristics: for example, Statecharts have been designed for reactive systems, Petri nets and process algebras for concurrent systems, UML-RT for real time systems.

Software systems are extremely complex and can change both during and after development. Constructing a complete and consistent model of the system is almost impossible and never cost effective. Modeling languages must support

flexibility, adaptability and instrumentability of specification and design. Software developers need a suitable blend of precision, for supporting analysis, and incompleteness that stems from lack of knowledge of the application domain and evolving design.

3 Petri Nets for Specification and Verification

The variety of uses of models in software engineering requires modeling languages with properties that may be very different if not contradicting. It is difficult to imagine a single language that satisfies all requirements and needs, rather, software engineers tend to use different modeling languages through the many phases of software development, as well as within the same phase. Modern methodologies, e.g., UML, are based on sets of modeling languages with different characteristics that are integrated in a unifying framework as in Figure 4.

The main challenge in software engineering is rarely to invent yet another modeling language, but more often it is to identify modeling languages suitable to the specific needs, and to integrate them in a coherent framework.

As any other modeling language, Petri nets cannot satisfy all needs of software engineering, but present features that can be appealing in many contexts within the software development process. The ability of easily modeling concurrency and synchronization aspects, the intuitive graphic notation, the formal semantics that supports powerful analysis capabilities and the availability of several supporting tools make Petri nets an appealing candidate in many situations. However, despite these advantages and the success stories in several application domains, Petri nets are not widely used in software engineering, and successful methodologies often suggest alternative modeling languages, e.g., SDL or Statecharts.

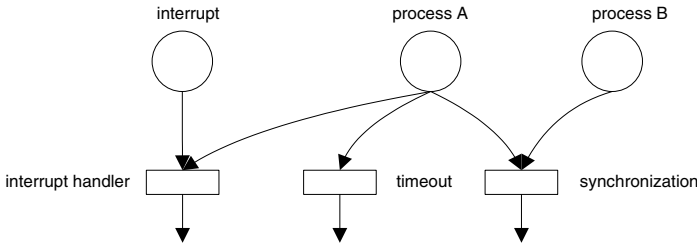
Goal of this paper is not to discuss the mutual diffusion of alternative modeling language, nor to identify the remote causes of relative successes and failures that may change in a few years, as happened many times in the still young history of software engineering. Rather, in this section, we will try to understand limits of Petri nets in coping with software engineering problems aiming at providing directions for further investigation.

Petri nets are available in many variants and extensions that span from place/transition nets to high-level nets and timed Petri nets. Here we focus on untimed models, leaving timed extensions to the next section, where we discuss the usage of Petri nets in the domain of embedded real-time systems.

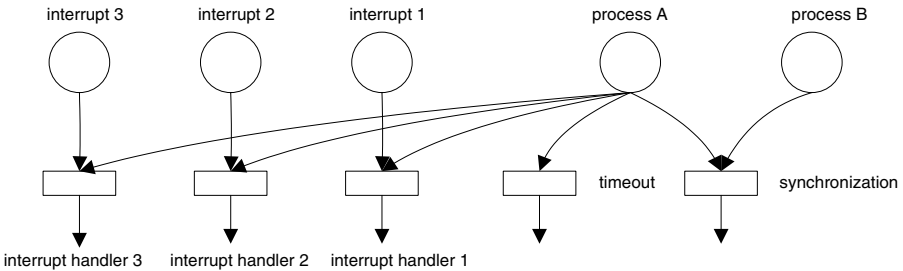
Place/transition nets provide an essential model of concurrency that can be very useful in addressing specific problems. The absence of constructs for dealing with data and negative conditions results in powerful analysis mechanisms, but limits the applicability to many interesting software engineering problems, and affects scalability.

Figure 5 illustrates the limits of place/transition nets from the modeling viewpoint. The place/transition net of Figure 5 (a) captures the essence of the problem (a process A that can either synchronize with process B or be inter-

rupted by an asynchronous interrupt or by the expiration of a timeout), but cannot model intuitively the duration of the timeout, or the conditions that may govern the timeout or the interrupt handler: if all places *interrupt*, *process A*, and *process B* are marked, the transition that fires is chosen non deterministically among *interrupt handler*, *timeout*, or *synchronization*. We cannot easily specify that transition *timeout* fires only if the token in place *process A* has a given age, or that transition *interrupt handler* fires only if the interrupt has a given priority or is of a given type. Figure 5 (b) illustrates the problems of scaling the description to the presence of different handlers for different interrupts. We can model several instances of interrupts of the same nature increasing the number of tokens, but we cannot distinguish the single tokens, and thus, to keep track of the identity of tokens we need to use different subnets.



(a) Process A synchronizes with process B unless timed out or interrupted.



(b) Presence of different interrupt handlers for different signals.

Fig. 5. Limits of place/transition nets as modeling language. (Transitions and places are labeled only for easy referencing)

Place/transition nets have been extended in several ways to overcome their modeling limits: inhibitor arcs, priority, time and predicates help solving different problems. For example, we can use predicates to distinguish different interrupt handling routines, like in Figure 6 that shows a colored Petri net model for three types of interrupt handlers that access resources “L” and “M” in different ways.

The top rectangle groups declarations of colors and variables. In the example, we have two colors: “IH”, corresponding to tokens of type “Interrupt Handlers”,

and “R”, corresponding to tokens of type “Resources”. We have three types of handlers: “i”, “j” and “k”, and two types of resources “l” and “m”. Variable “x” of type “IH” is used in the expressions that annotate arcs to indicate an interrupt handler. Places are annotated with the type of tokens they can contain, and with a marking and an initialization expression. The marking is expressed as a number in a circle and an expression nearby. Place “L” is initially marked with three tokens of type “l” and Place “M” is marked with 2 tokens of type “m”. The figure is a subnet of a larger model, place A is marked by the firing of “ancestor” transitions. The initialization expression is expressed as an underlined expression beside places and helps initializing the net. In the figure, the initial marking corresponds to the effect of applying the initialization expression. Arcs are annotated with expressions that indicate the number and type of tokens flowing on the arcs. The firing of transition “T1” removes a token of type “IH” from place “A” and either a token of type “l” from place “L” if the considered interrupt is of type “i” or “j”, or two tokens of type “l” if the interrupt is of type “k”, and produces an “IH” token in place “B” with the same identity of the token removed from the input place.

The special keyword “empty” indicates that no tokens of that type flows on the arc. For example, handlers of type “j” and “k” release resources of type “l” after the firing of transition “T2” (one and two resources, respectively), while handlers of type “i” returns the resource of type “l” only after the firing of transition “T3”. Similarly, handlers of type “i” and “k” perform actions “T1”, “T2” and “T3”, while handlers of type “j” stop after action “T2” and continue with “T4”.

We can see that colored Petri nets allow identifying different handlers, and to model handlers of different types without affecting the complexity of the net structure, capturing identity and actions with colors and annotations.

The information captured by structure and annotations can be balanced in different ways. For example, we could compact all states of the interrupt handlers in a single state and use colors and predicates to describe the evolution of the computation.

The various extensions of Petri nets and in particular colored Petri nets (or, more generally, high-level Petri nets) are very useful for modeling many software engineering problems, and find some important applications. Unfortunately, adding modeling capabilities and “compacting” the structure solves only some of the limits of the modeling language. Software engineers need flexible, adaptable and scalable modeling notations: they need to change and adapt the notation to the different abstraction levels, to the different stakeholders involved in the development, to the different application domains, and to the evolution of requirements during and after development. They need to cope with large problems and they look for models that help mastering complexity and size. Petri nets, as many other formal methods, rarely provide the flexibility, adaptability, modularity and scalability required in software development. Research in Petri nets moved in two main directions: adding either modeling power or user-friendly interfaces to Petri nets.

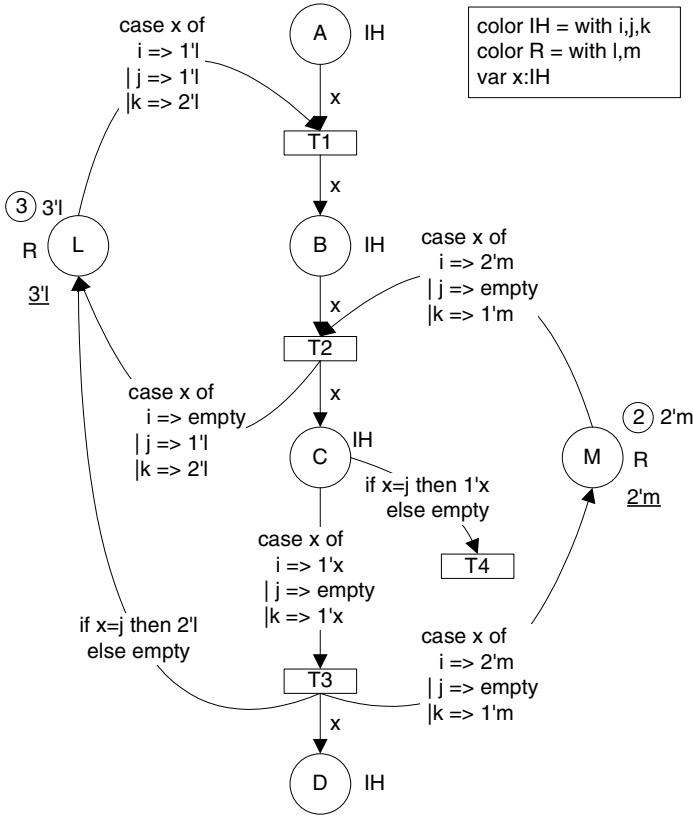


Fig. 6. A colored Petri net model for a set of interrupt handling routines that access resources “L” and “M”

Petri nets have been augmented in many additional ways with hierarchy and object oriented constructs to add flexibility, adaptability and scalability, i.e., the *modeling power* sought by software engineers. Although the different extensions provide useful capabilities and find interesting applications, none of them has a prominent role in software engineering yet. All these useful attempts move in a single direction, ignoring the complexity of the software engineering domain. Software engineering must consider modeling power, precision, analyzability, but also costs, understandability, tool support, and familiarity with the notation. Similarly to programming languages, modeling languages succeed when they present an appealing balance among the different needs. Stochastic Petri nets represent a notable case: they do not present specific features that make them more modular, flexible, adaptable or scalable than other Petri net extensions, but they address a specific problem, namely performance evaluation. When performance is prominent, and then the tradeoff among costs, training, understandability, flexibility, adaptability and scalability is unbalanced towards analysis, software engineers do not hesitate to use stochastic Petri nets in the mosaic of notations adopted in the software development. The (limited, but no-

table) success of stochastic Petri nets in software engineering may indicate a direction to pursue to increase the applicability of Petri nets, or, from a pessimistic viewpoint, a limit of their applicability: finding specific problems where Petri nets can provide an advantageous solution and adapting Petri nets to the identified problems.

Recognizing that the main advantage of Petri nets, as well as other formal methods, relies not in their intuitive modeling power as communication means, but in their powerful analysis capabilities suggests a different research direction that has been pursued by many scientists: finding “user-friendly” interfaces. The approach is somehow similar to high level programming languages that provide a useful abstraction of the underlying machine and hide the complexity of machine languages, but allow programmers to execute their code, i.e., to take full advantage of the underlying machine language. Similarly, several scientists have been worked on “dual-language” approaches where successful user-friendly specification and design notations are paired with formal models that support powerful analysis capabilities. The approach has been investigated with many specification notations and formal models, including Petri nets. The straightforward operational semantics of Petri nets that supports different types of analysis including “partial” analysis that can be obtained from example by executing partial specifications, provide a strong advantage over other formal models, whose semantics cannot be paired with many specification notations as easily as Petri nets. Moreover, the huge body of knowledge on Petri nets and the immediate modeling of concurrency aspects makes them more appealing than other formal models with operational semantics.

Many scientists defined “compilers” from different specification notations (recently UML, but in the past structured analysis, SDL, etc...) to Petri nets. “Traditional style compilers” that freeze a notation and provide a specific semantics through a fix mapping to Petri nets forget the tradeoff among the variety of aspects to be considered in software engineering: while the primary need of executing the final code overcomes many other requirements, and thus makes acceptable the use of programming languages with fixed and precise semantics, flexibility, understandability and adaptability often overcome the need of analyzability in specifications, thus making it difficult to accept “frozen specification notations”. Many projects tried for example to find “the” semantics of structured analysis and provided tools for automatically capturing the identified semantics with a mapping to Petri nets. The resulting frameworks allow for formally analyzing structured analysis, but limit the freedom of the analysts or the architects, who cannot adapt the notation to the specific needs of the end-users, of the application domains or of the changes in requirements. Few attempts survived a few pilot projects.

Several scientists pursue an interesting alternative that consists of providing flexible semantics, i.e., mappings from specification notations to formal models that can be adapted to new needs and requirements. Mappings are given as sets of rules that can be adjusted to meet different needs that result in different interpretation of the same syntactic element or in modifications of the notation.

Flexible approaches seem a better tradeoff for software engineering, but their success is still bound to the ability of identifying a clear advantage in terms of analysis capabilities added with Petri nets.

Software engineering is dealing with new problems that derive from the rapid spread of new applications: pervasive computing, mobile applications, heterogeneous environments, software components that are reused in new unforeseen frameworks, context awareness and new constraints derived from resource bounds like screen size (palm devices), variable bandwidth (mobile computing) present new challenges that may not be easily addressable with traditional techniques. The software engineering community is actively seeking new solutions and ideas to address these new problems. Petri nets as many other modeling languages may provide useful support to some new challenges, thus starting a new time for collaboration among the two communities.

4 Petri Nets for Embedded Real-Time Systems

Petri nets can be used to address the needs of specific application domains. Here we survey embedded real-time systems, which seem particularly well suited for time and stochastic extensions of Petri nets. We will try to summarize the state of art and the future trend in this important domain with respect to possible uses of Petri nets.

In many application domains, software is *embedded* in larger systems. The software is the heart of the systems: it sends control signals and receives feedback. The behavior of these systems is time dependent: the correctness of the software cannot be expressed merely in terms of functional relations between inputs and outputs, but depends on the instants at which the results are produced. A functionally correct result produced too late may be wrong. For example, a drive-by-wire system that computes the correct maneuver for avoiding an obstacle too late, e.g., after crashing into the obstacle, is obviously wrong regardless of the produced value. Results produced too early may be wrong as well. For example the signal for controlling the delivery of power to an electrical engine cannot be produced too early, otherwise the engine may reach a wrong speed at a wrong time.

Missing deadlines can have different consequences for different systems. In some cases, it can be tolerated if it does not happen too frequently, while in other cases, results must be always available within the deadlines. Although the distinction is not sharp, we often classify real-time systems as *hard* and *soft*. Hard real time systems do not tolerate missing deadlines. An approximate results produced within the deadline may be preferable to an exact result produced too late. This is the case of many control systems that must send control signals when they are needed by the controlled systems: we prefer a vehicle to break a bit too suddenly, because the control software computes an approximate control signal, but avoids the collisions, to a vehicle that crashes because the ideal signal is computed after the collision.

Soft real time systems can tolerate some late results. For example, voice packets must be received with specific frequencies for reproducing the correct voice signal, but a small percentage of late packets can be ignored without appreciable degradation of the reproduced voice signal.

Hard and soft real time constraints often coexists in the same systems: the GPS signal used by the drive-by-wire system as well as by the position display on the driver console owns hard real time constraints in one case, since a late signal to the drive-by-wire system may cause the vehicle to crash, and soft real time constraints in the other, since a late signal to the driver display may not be even perceivable to the end user.

The distinction between hard and soft real time system is important to identify suitable analysis techniques: performance analysis may be enough for soft real time systems, but is rarely sufficient for hard real time systems.

Embedded real time systems can be composed of several concurrent subsystems. They include at least the controller and the controlled system, but more often, both the software controller and the controlled system include several concurrent subsystems. Different components are often of different nature, the behavior of the components of the controlled system is usually time-continuous, while the behavior of the controlling software is usually time-discrete. The controlled system and the controlling software interact through special purpose devices (sensors and actuators) that may be responsible of failures. Moreover, the timing of the system depends on elements that are usually not considered in “traditional” systems: hardware, operating system, and middleware. Abstracting from such elements may not be possible for not trivial real-time systems.

Embedded real time systems present many new challenges. Modeling and analyzing systems in the early development phases requires models and analysis technique that can capture the subtle intertwining between functional and timing aspects, and that can model both continuous and discrete timing. The correspondence between requirement specifications and code must take into account limited availability of resources and constraints that can derive from the hardware and software platform. Analysis techniques must cope with new properties that include timing and safety properties.

Petri nets were originally proposed for modeling concurrent systems abstracting away from timing aspects. Extensions of Petri nets for dealing with time have been studied since the early seventies. We can identify two different approaches: *timed Petri nets* that augment Petri nets with deterministic time, and *stochastic Petri nets* that augment Petri nets with time probabilities.

Timed Petri nets have been proposed as early as 1974 by Ramchandani [22], and by Merlin and Farber in 1976 [9]. Since the early proposals, Petri nets have been extended with time in several ways, by adding time to either places or transitions or both, by interpreting time as firing delay or firing duration, by adding a single time value or a time set (interval) to transitions or places. Different models satisfy different needs, but they are all substantially equivalent from the semantic viewpoint.

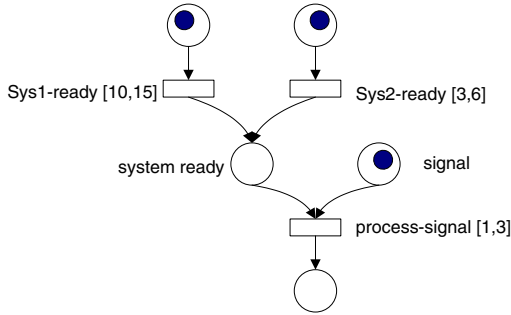


Fig. 7. A simple timed Petri net. Transitions are augmented with pairs of numeric values that represent the minimum and maximum firing time relative to the enabling time, i.e., the instant at which all input places are marked. Labels are added only for referencing. Marking is represented by black dots in places

Extending Petri nets with time can greatly affect the semantics. While *weak time semantics* does not affect the locality of enabling, *strong time semantics* violates the locality principle. Informally, *weak time semantics* considers the time constraints as instants at which the modeled events will happen, if they happen, while *strong time semantics* considers the time constraints as instants at which the events must happen.

Let us consider for example the timed Petri net of Figure 7 that represents a simple systems where an incoming signal can be handled by two different signal handlers (*Sys1* and *Sys2*). *Sys1* required from 10 to 15 time units to become ready, *Sys2* requires from 3 to 6 time units. The signal is processed in 1 to 3 time units.

If we consider each transition “locally”, i.e., ignoring relations among firings that may derive from time constraints, both transitions *Sys1-ready* and *Sys2-ready* are enabled. If transition *Sys1-ready* fires at time e.g. 10, transition *process-signal* is enabled in the interval $\langle 11, 13 \rangle$, i.e., between 1 and 3 time units after the enabling at time 10. The firing of transition *process-signal* at time e.g. 12 consumes the tokens. Thus, the token in place *signal* is not available any more. If we now consider again transition *Sys1-ready* enabled between 3 and 6, it can fire e.g. at time 5. The considered sequence of firings can be ordered to obtain a monotonically non-decreasing sequence with respect to time: *Sys1-ready* at time 5, *Sys2-ready* at time 10, *process-signal* at time 12, obtaining a legal firing sequence according to weak time semantics. This is true in general: each firing sequence obtained by considering transitions locally is equivalent to a legal time monotonically non-decreasing firing sequence according to weak time semantics. This means that analysis performed on the underlying Petri net produces results that are valid also for the timed net.

In the considered example, the obtained sequence represents the case in which *Sys1* becomes available for handling the signal at time 5, but does not handle the signal for some reasons that are not explicitly captured by the model. The signal is handled later by *Sys2* that becomes available at time 10.

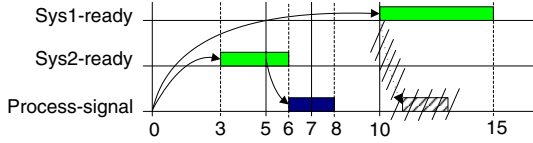


Fig. 8. Weak vs. strong time semantics

Unfortunately, strong time semantics does not have this nice property. The same firing sequence cannot be ordered according to strong time semantics, as illustrated in Figure 8. Initially both transitions *Sys1* and *Sys2* are “locally” enabled, but transition *Sys1* must fire before its deadline (time 6). The firing of transition *Sys2* e.g. at time 5 enables transitions *process-signal* within the time interval $\langle 6, 8 \rangle$. Transition *process-signal* must fire before time 6. Transition *Sys1* can fire only after the firing of transition *process-signal*, e.g., at time 10. Since transition *process-signal* is forced to fire before time 8, removing the token from place *signal*, the token produced by the firing of transition *Sys1* cannot enable transition *process-signal*, differently from the case of weak time semantics.

The model used in Figure 7 that associates firing fixed time intervals to transitions cannot capture all aspects of real time systems. In particular, we cannot model complex intertwining between timing and functional aspects. Let us assume for example that the processing of the signal depends on the load of the system that handles it, and that the choice of the handling system depends on the characteristics of the incoming signal. The fixed time interval associated to transition *process-signal* that indicates the minimum and maximum firing time as constants can approximate the modeled system by indicating an upper and a lower bound, and cannot express complex conditions for selecting the signal handler depending on the nature of the signal.

Complex intertwining between timing and functional aspects can be modeled by merging timed and high-level Petri nets (HLTPN). HLTPNs associate data (and timestamps) to the tokens, and predicates, actions and time functions to transitions, as shown in Figure 9.

Soft real time systems can be modeled and analyzed with stochastic Petri nets that were first proposed in the late seventies by Sifakis [23] and later extended by many scientists. Stochastic Petri nets augment transitions with a distribution of probability that the transition will fire. Figure 10 shows a simple example of generalized stochastic Petri nets: a processing task that may or may not require instrumentation, and may or may not require elaboration. The two choices are represented with the two pairs of conflicting transitions *need instrumentation*, *instrumentation ok*, and *need processing*, *system ok*. Black transitions indicate immediate transitions, i.e. transitions that fire immediately, representing instantaneous decisions, while white transitions indicate timed transitions, representing the termination of actions with given durations. Transitions are associated with a priority π and a weight W . Immediate transitions fire first, while timed transitions fire only when no immediate transition is enabled. Pri-

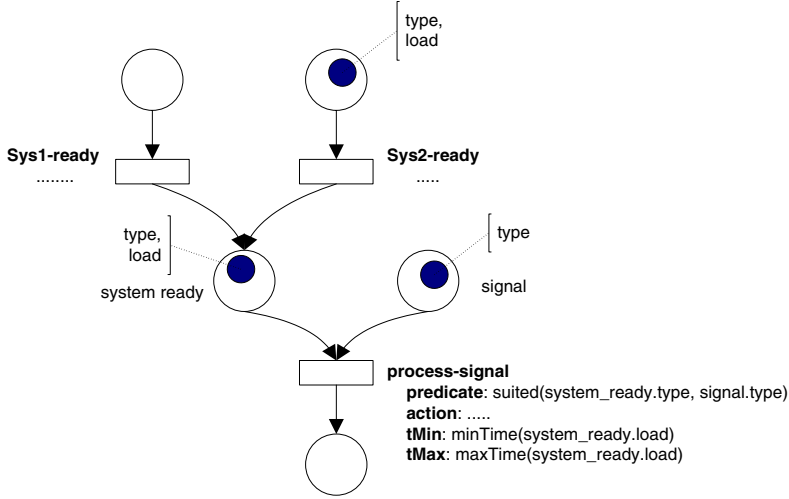


Fig. 9. A simple HLTPN. Data associated with tokens are represented with types. Predicates, actions and time intervals associated with transitions are partially given only for transition *process-signal*. The predicate requires the evaluation of a boolean function *suited* that computes the suitability of the system to process the signal. The time interval can be computed by evaluating functions *minTime* and *maxTime* that compute the minimum and maximum firing time according to the load of the system

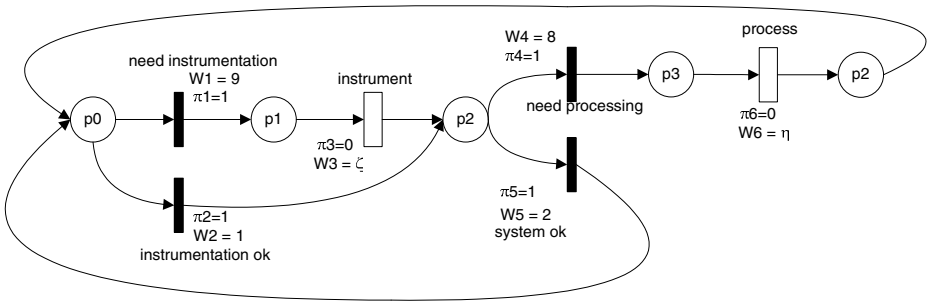


Fig. 10. A simple generalized stochastic Petri net

ority defines a (partial) order of firings among transitions of the same kind. The weight indicates the frequency of firings for immediate transitions with equal priority, and a distribution of probability that describes the firing time of timed transitions.

In the example, transitions *need instrumentation* and *instrumentation ok* have the same priority and fire with a ration of 9:1, while transitions *need processing* and *system ok*, also with equal priority, fires with a ration 8:2.

Stochastic Petri nets support powerful performance analysis that determines their success in important application domains. Timed Petri nets support time reachability analysis.

Real time systems are quickly evolving: the spread of SoC (System on a Chip), the vanishing distinction of hardware and software components, the increasing use of COTS (Components-Of-The-Shelf) in complex real time systems, the introduction of Internet connectivity introduce new challenges that call for new methodologies and techniques and open new potentialities to Petri nets as well as other formal methods.

5 Software Processes

Complex software is developed by a set of specialists that use many techniques and tools, and collaborate over a long period of time to design, develop and maintain a suitable product. Often, nobody knows all the details of a software product, but different actors share partial views of the system. For example, managers and analysts may have an abstract view of some aspects of the overall system, but may not know all implementation details, while architects, designers, programmers and test engineers may have a detailed view of part of the software, but not be familiar with other parts.

People, techniques and tools must be suitably coordinated and organized over time to assure the success of a project, i.e., the developed of the right product within time, resource and environmental constraints. The overall organization of the activities required to develop, test and maintain a software product is called a software process.

A software process consists of a set of interacting software engineering activities aimed at producing (and maintaining) a software product. A key property of a software process is *visibility*, i.e., the ability of examining progresses and results. Process visibility gives the possibility to monitor and steer the process towards its goals. Visibility is often achieved by identifying different phases and associating activities and phases with the production of intermediate artifacts, such as, requirements specifications, design specifications, code and quality reports, which are often associated with process milestones.

Large projects span over many months. Requirements are seldom clear at the beginning of the project. Usually a first core of requirements is detailed and expanded through the process following the increasing understanding of the problem and the solution, and adapting to the evolution of the domain. Systems are rarely developed as single monolithic products. More often, systems are developed incrementally though several iterations that produce many prototypes and releases that increasingly approximate the final product. Process phases can seldom be organized as separate development steps, as postulated by the waterfall process model. They often overlap with complex interaction patterns that must be suitable organized and monitored. Software projects usually involve separated teams that work concurrently on different phases of the process.

The large variety of situations results in many different requirements that cannot be fully captured by a standard process. Each project has its own properties, and requires a specific process. The definition and implementation of a software process is a complex activity that can and shall be suitably programmed and executed [24]. Software processes must be suitably described to guide and coordinate the key activities, and to push forward repeatability and controllability of the processes. Rigorous software process descriptions enable the development of tools to enact process descriptions thus automating coordination of activities, tools and people. The definition of precise process models is also referred to as *software process programming*.

The goal of software process programming is the creation of *process-centered software engineering environments* (PSEEs), i.e., information systems that support the enactment of software processes. The core of a PSEE is a *Process Modeling Language* (PML), i.e., the language used to describe the target processes. Ambriola et al. outline the main requirements for a PML [25], which can be summarized as follows:

Modeling concurrency: Concurrency is intrinsic in software processes. PMLs must be able to clearly capture the concurrency of activities and their synchronization.

Modeling products: The artifacts produced during software development are complex and strictly interrelated. For example, a test report that points out a failure of a software system must be related to the tested software version and to the revealing test case, which in turn is related to a number of implementation artifacts (e.g., test drivers and stubs). PMLs must take into account the structure of the artifacts involved in the process and their mutual relationships.

Managing tool integration: Software process activities are supported by several tools (e.g., editors, compilers, debuggers, configuration management systems, and so forth). Mechanisms to control at a fine-grained level the tools involved with the process are essential for managing the interactions between a PSEE and its users in an effective way. PMLs must provide both active and reactive mechanisms to allow the PSEE to send messages to external tools and to react to messages from external tools, respectively.

Supporting process enactment: Software process descriptions must be interpreted to provide automatic support to processes when they are executed (process enactment). PMLs must produce operational descriptions with well-defined non-ambiguous semantics.

Supporting analysis: PMLs must support verification of important properties, e.g., absence of deadlocks, against process descriptions.

Supporting evolution: Software process improvement is an important issue in software engineering and has been the target of important industrial and research activities in the last years. For example, the SEI Capability Maturity Model (CMM [26]) defines a framework for assessing the level of maturity of a software process. This framework consists of five maturity levels. At the first level, software processes are chaotic and uncontrolled, activities

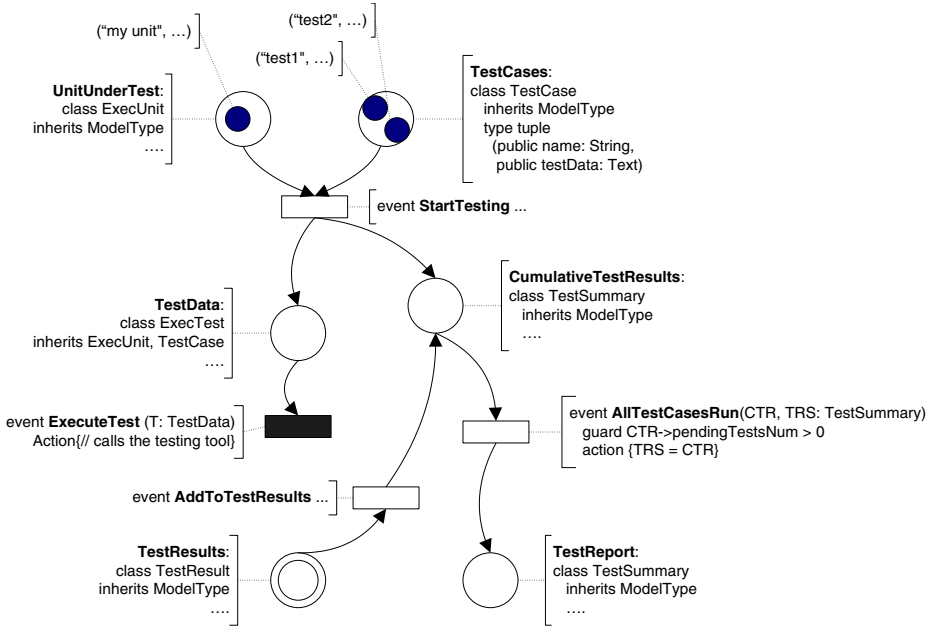


Fig. 11. Description of a generic unit test session in SLANG

are carried without any explicit guideline and there is no description of the process. The introduction of methods and technologies such as configuration management, quantitative measurement, quality control and process description, is expected to gradually increase the maturity of a software process and correspondingly its capability of dealing successfully with complex software projects. At level 5, software processes are continuously improved based on the experience and data accumulated over time. Mature software processes must support evolution of the the process descriptions. To this end PMLs must possess reflexive features allowing to modify process descriptions either off-line or on-the-fly.

Petri nets present several nice properties that make them particularly appealing as PML: They have a precise semantics; They provide an intuitive way of modeling concurrency and non-determinism; Their marking supports easily modeling of the process state, thus facilitating the representation of milestones and conditional choices; Their operational semantics allows to easily represent and analyze process enactment; Their intuitive visualization provides an excellent communication means; There exists a large body of theory that support analysis of many properties under different assumptions; There are many supporting tools.

Basic Petri nets have been extended and specialized in several ways to cover all requirements of a PML, by adding reflexivity, features for modeling process

artifacts, and mechanisms for managing tool integration. Extensions of Petri nets to cope with process modeling aspects are illustrated in the example of Figure 11 that shows a model of a generic unit test session in SLANG (the PML of the SPADE PSEE [19]). The availability of both the unit under test and a set of corresponding test cases triggers the testing activity that starts setting up the testing environment and starting the tracking of cumulative test results. The test results are incrementally cumulated while executing the test cases. When all test cases have been executed, a test report is generated.

SLANG extends high-level Petri nets, using tokens to represent the process data. SLANG tokens are structured objects whose types are defined in the traditional object-oriented style, i.e., as a data structure that can be accessed through a set of exported operations. The net places are associated with a type (a.k.a. class) and they can contain only objects of the associated type. All SLANG types are organized in a type hierarchy as specified by the inheritance relation. The object oriented-paradigm makes it possible to describe the structure of software artifacts. For example, in the figure, the test cases are described as tuples with two fields: a string that identifies the test case by name and a text that describes the test data.

The transitions that in SLANG are called *events* are associated with guard predicates that control their execution, and actions that describes the effect of the firings on the tokens. A transition can fire if enabled by the associated guard predicate evaluated on the tokens in the input places. Its firing removes the tokens from the input places and produces tokens in the output places according to the associated action. For example, in the figure, the transition *AllTestCases-Run* is not enabled (guarded) until there are still test cases to execute. Its firing produces the test report from the incrementally generated test summary.

SLANG uses special *black transitions* and *user places* to integrate CASE tools. Black transitions send asynchronous messages to external tools as part of their action. User places (double circles) change their content as a result of an event that happens in the user environment. For example, in the figure, the event *ExecuteTest* is a black transition whose action calls an external testing tool for executing the test cases. After the execution of each test case, the external tool will produce a token in the user place *TestResults*, thus allowing the process to progress.

SLANG provides reflexive features for dynamically modifying a process description. Figure 12 shows a SLANG type hierarchy that includes both the predefined SLANG types and the user defined types for the previous examples. All types that participate to a process description derive from the predefined type *Token* that defines the set of properties common to all tokens. All user defined types inherit (directly or indirectly) from the predefined type *ModelType* which is a direct descendent of *Token*. The set of predefined types includes three additional types: *Activity*, *Meta type* and *Active copy* that allow to access the activity definitions, the user type definitions and the instantiated copies of a process during enactment, respectively. The specification of how to modify the process can be part of the process description itself.

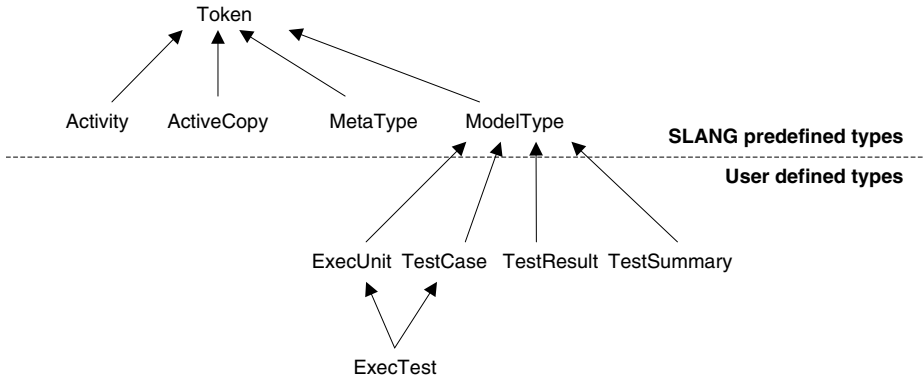


Fig. 12. A SLANG type hierarchy

Thirty years of research solved many problems in software process modeling, but some are still open: tolerability to inconsistencies and incompleteness, non-intrusiveness of PSEE, inconsistency management are some of the problems where Petri nets can find new applications.

6 Further Readings

The literature on software engineering and Petri nets is immense and finding a good compass is hard. Here we try to indicate some doors to access the enormous body of knowledge for identifying areas of common interest for software engineers and Petri net experts following the schema of this paper.

A good way for understanding problems and dimensions of software engineering is the volume “Future of Software Engineering” published in 2000 [27]. The introduction illustrates the many dimensions of the discipline, while the many papers present the current trends of the most important areas.

Modeling languages have been widely studied and it is difficult to identify a good survey. Interested readers can find a general overview in software engineering handbooks, e.g., in [28], [29] or [30]. A good survey of formal methods is given by Wing [31], while an interesting discussion on the role of formal methods in software engineering is proposed by Saiedian [32]. The different models that comprise the UML approach are illustrated in many book, e.g., [21].

A comprehensive overview of Petri nets is given in [33]. Colored Petri nets together with a sample of industrial applications are presented in Jensen’s books [34–36]. The volume edited by Agha and De Cindio discusses the use of Petri nets in the object oriented framework [37].

Approaches for mapping various notations to Petri nets have been proposed by many authors: [38–42]. Rule based mappings have been proposed by Paige [43] and Baresi et al. [44].

Stochastic Petri nets are presented in the books by Bause and Kritzinger and by Ajmone Marsan et al. [45, 46]. Timed Petri nets are discussed in the classic

paper by Merlin and Farber [9], while high-level timed Petri nets (ER nets, in the paper) and weak and strong time semantics have been introduced by Ghezzi et al. [11]. Time Petri nets are overviewed also in the book by Nissanke [12]. Time reachability analysis is discussed by Berthomieu and Diaz [10] for timed Petri nets, and by Ghezzi et al. for high-level timed Petri nets [47].

The term “software process” was first proposed by Osterwiel in his seminal paper [24]. Various PSEE are discussed in many papers, e.g., [48, 49, 19, 18]) The possibility of using Petri nets as a PML for describing workflows of business processes and many results related to the use of Petri nets for this purpose have been described in Chapter 1² of this book.

7 Conclusions

The meeting of problem- and solution-oriented disciplines can lead to important progresses in both areas. Petri nets provide an excellent means for modeling concurrent aspects and have been extended in many ways to cope with many problems. Petri nets have been successfully applied many times to several software engineering problems. However, the two disciplines do not go through a period of particularly strong cross fertilization. This paper tried to overview some aspects of software engineering, pointing to aspects where Petri nets have been or can be proposed as solutions to critical problems. We hope to have provided few ideas to foster new fruitful collaborations between the two disciplines.

Acknowledgment

This paper springs from a course held at the Advanced Course on Petri Nets (ACPN) 2003. A preliminary version of this material has been presented as a tutorial at the 24th International Conference on Application and Theory of Petri Nets (ICATPN 2003) held in Eindhoven in June 2003 jointly with Gregor Engels, who greatly contributed to the preparation of the tutorial and the organization of the content. We would like to thank Gregor for the invaluable preliminary work that helped shaping the course and the derived paper.

References

1. Young, M.: Neat models of messy problems: Notes on the interplay between solution- and problem-centered disciplines, and more particularly on the interaction between Petri net research and software engineering research. *International Journal of Computer Systems Science and Engineering* **16** (2001)
2. Oswald, H., Esser, R., Mattmann, R.: An environment for specifying and executing hierarchical petri nets. In: *Proceedings of the 12th International Conference on Software Engineering*. (1990) 164–172

² This is a reference to the Chapter of this book authored by van der Aalst and related to workflow modeling.

3. Duri, S., Buy, U., Devarapalli, R., Shatz, S.: Application and experimental evaluation of state space reduction methods for deadlock analysis in ada. *ACM Transactions on Software Engineering and Methodology* **3** (1994) 340–380
4. Jahnke, J., Schafer, W., Zundorf, A.: Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications. In: *Proceedings of the European Conference on Software Engineering 1997*, Springer Verlag (1997)
5. Bastide, R., Palanque, P.: A Petri net based environment for the design of event-driven interfaces. *Lecture Notes in Computer Science* **935** (1995)
6. Leveson, N., Stolzy, J.: Safety analysis using petri nets. *IEEE Transactions on Software Engineering* **SE-13** (1987) 386–397 19 refs.
7. Azema, P., Juandele, G., Sanchis, E., Montbernard, M.: Specification and verification of distributed systems using PROLOG interpreted Petri nets. In: *Proceedings of the 7th International Conference on Software Engineering*, IEEE Computer Society Press (1984) 510–519
8. Suzuki, T., Shatz, S.M., Murata, T.: A protocol modeling and verification approach based on a specification language and petri nets. *IEEE Transactions on Software Engineering* **16** (1990) 523–536
9. Merlin, P., Faber, D.J.: Recoverability of communication protocols. *IEEE Transactions on Communication* **24** (1976) 1036–1043
10. Berthomieu, B., Diaz, M.: Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering* **17** (1991) 259–273
11. Ghezzi, C., Mandrioli, D., Morasca, S., Pezzè, M.: A unified High-Level Petri Net formalism for time-critical systems. *IEEE Transactions on Software Engineering* **17** (1991) 160–172
12. Nissanke, N.: *Realtime Systems*. International series in computer science. Prentice Hall (1997)
13. T.D.C., L., A., G.: Synchronisation and storage models for multimedia objects. *IEEE Journal on Selected Areas in Communications* **8** (1990) 413–427
14. Wahl, T., Rothermel, K.: Representing time in multimedia systems. In: *Proceedings of the International Conference on Multimedia Computing and Systems*, Boston, USA. (1994)
15. Willrich, R., Saqui-Sannes, P.D., Senac, P., Diaz, M.: Multimedia authoring with hierarchical timed stream Petri nets and Java. *Multimedia Tools and Applications* **16** (2002) 7–27
16. Engels, G., Sauer, S.: Object-Oriented Modeling of Multimedia Applications. In: *Handbook of Software Engineering and Knowledge Engineering* (S. K. Chang ed.). Volume 2. World Scientific (2002) 21–52
17. Vazirgiannis, M.: Interactive multimedia documents: modeling, authoring, and implementation experiences. Volume 1564 of *Lecture Notes in Computer Science*. Springer-Verlag (1999)
18. Emmerich, W., Gruhn, V.: FUNSOFT Nets: a Petri-Net based Software Process Modeling Language. In Ghezzi, C., Roman, G., eds.: *Proceedings of the 6th ACM/IEEE Int. Workshop on Software Specification and Design (IWSSD)*, Como, Italy, IEEE Computer Society Press (1991) 175–184
19. Bandinelli, S., Fuggetta, A., Ghezzi, C., Lavazza, L.: SPADE: An environment for software process analysis, design, and enactment. In Nuseibeh, B., Finkelstein, A., Kramer, J., eds.: *Software Process Modelling and Technology*. John Wiley and Sons (1994) 33–70

20. Florin, G., Natkin, S.: Generalization of queueing network product form solutions to stochastic Petri nets. *IEEE Transactions on Software Engineering* **17** (1991) 99–107
21. Booch, G., Rumbaugh, J., Jacobson, I.: *The Unified Modeling Language User Guide*. 1 edn. Addison-Wesley, Reading, Massachusetts, USA (1999)
22. Ramchandani, C.: Analysis of asynchronous concurrent systems by timed Petri nets. Technical Report Project MAC Tech. Rep. 120, Massachusetts Institute of Technology (1974)
23. Sifakis, J.: Use of Petri nets for performance evaluation. *Acta Cybernetica* **4** (1978) 185–202
24. Osterweil, L.: Software processes are software too. In: *Proceedings of the 9th International Conference on Software Engineering*, IEEE Computer Society Press (1987) 2–13
25. Ambriola, V., Conradi, R., Fuggetta, A.: Assessing Process-Centered Software Engineering environments. *ACM Transactions on Software Engineering and Methodology* **6** (1997) 283–328
26. Paulk, M., Curtis, B., Chrissis, M., Weber, C.: Capability maturity model, version 1.1. *IEEE Software* **10** (1993)
27. Finkelstein, A., ed.: *The future of software engineering (part of the Proceedings of the 22th International Conference on Software Engineering)*. ACM Press, NY (2000)
28. Ghezzi, C., Jazayeri, M., Mandrioli, D.: *Fundamentals of Software Engineering*. 2 edn. Prentice Hall, Englewood Cliffs (1999)
29. Sommerville, I.: *Software Engineering*. 6th edn. Addison-Wesley (2001)
30. van Vliet, H.: *Software Engineering: Principles and Practice*. John Wiley & Sons, Chichester (1993)
31. Wing, J.: A specifier's introduction to formal methods. *Computer* **23** (1990) 8, 10–22, 24
32. Saiedian, H.: An invitation to formal methods. *IEEE Computer* **29** (1996) 16–30
33. Murata, T.: Petri nets: properties, analysis, and applications. In: *Proceedings of the IEEE*. Volume 77. (1989) 541–580
34. Jensen, K.: *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Uses*, vol. 1: Basic Concepts. EATCS Monographs on Theoretical Computer Science. Springer-Verlag (1992)
35. Jensen, K.: *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Uses*, vol. 2: Analysis Methods. EATCS Monographs on Theoretical Computer Science. Springer-Verlag (1995)
36. Jensen, K., ed.: *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Uses*, vol. 3: Practical Uses. Number 1217 in *Lecture Notes in Computer Science*. Springer-Verlag (1997)
37. Agha, G., Cindio, F.D., Rozenberg, G.: Concurrent object-oriented programming and Petri Nets: advances in Petri Nets. Volume 2001. Springer-Verlag Inc., New York, NY, USA (2001)
38. France, R.: Semantically extended data flow diagrams: A formal specification tool. *IEEE Transactions on Software Engineering* **18** (1992) 329–346
39. Fencott, P., Galloway, A., Lockyer, M., O'Brien, S.: Formalising the semantics of Ward/Mellor SA/RT essential models using a process algebra. *Lecture Notes in Computer Science* **873** (1994)

40. Petersohn, C., Huizing, C., Peleska, J., de Roever, W.: Formal semantics for Ward and Mellor's transformation schemas and its application to fault tolerant systems. *International Journal of Computer Systems Science and Engineering* **13** (1998) 131–136
41. Shi, L., Nixon, P.: An improved translation of SA/RT specification model to high-level timed Petri nets. In Gaudel, J.W.M.C., ed.: *FME '96: Industrial Benefit and Advances in Formal Methods*. Volume 1051 of LNCS., Springer Verlag (1996) 518–537
42. Richter, G., Maffeo, B.: Toward a rigorous interpretation of ESMIL-extended systems modeling language. *IEEE Transactions on Software Engineering* **19** (1993) 165–180
43. Paige, R.F.: A meta-method for formal method integration. *Lecture Notes in Computer Science* **1313** (1997)
44. Baresi, L., Orso, A., Pezzè, M.: Introducing formal specification methods in industrial practice. In: *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, NY, ACM (1997) 56–67
45. Bause, F., Kritzinger, P.: *Stochastic Petri Nets - An Introduction to the Theory*. Advanced Studies in Computer Science. Vieweg Verlag (1996)
46. Marsan, M.A., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: *Modelling with Generalized Stochastic Petri Nets*. Wiley Series in Parallel Computing. John Wiley and Sons (1995)
47. Ghezzi, C., Morasca, S., Pezzè, M.: Validating timing requirements of time basic net specifications. *Journal of Systems and Software* **27** (1994) 97–117
48. Montangero, C., Ambriola, V.: OIKOS: Constructing process-centred SDEs. In A. Finkelstein, J. Kramer, B.N., ed.: *Software Process Modelling and Technology*. John Wiley and Sons (1994) 131–151
49. Conradi, R., Hagaseth, M., Larsen, J., Nguyen, M., Munch, B., Westby, P., Zhu, W., Jaccheri, M., Liu, C.: EPOS: Object-oriented cooperative process modeling. In Nuseibeh, B., Finkelstein, A., Kramer, J., eds.: *Software Process Modelling and Technology*. John Wiley and Sons (1994) 33–70