Simulação de Modelos de Processo de Software com Máquinas de Estado Abstratas

Clayton Vieira Fraga Filho[1], José Luis Braga[1], Alcione de Paiva Oliveira[1], Vladimir Oliveira Di Iorio[1]

[1]Departamento de Informática – Universidade Federal de Viçosa (UFV).
Av. PH Rolfs s/n, Campus UFV - CEP 36570-000 - Viçosa - MG.

{clayton,zeluis,alcione,vladimir}@dpi.ufv.br

Resumo. *Este artigo apresenta a simulação de modelos de processos de software com máquinas de estado abstratas. Inicialmente é realizada a formalização dos elementos estruturais e comportamentais de processos de software, utilizando como referência o SPEMasm, adaptado do SPEM. O SPEMasm apresenta os elementos estruturais e comportamentais que permitem a simulação de processos de software. É realizado um estudo de caso onde um projeto é simulado na máquina de estado abstrata apresentada neste artigo, cujo modelo foi adaptado do OpenUp/Basic.*

Abstract. *This paper presents the simulation of software process models with abstract state machines. Initially a formalization of structural and behavioral processes of software is done, using as reference the SPEMasm, adapted as an extension to SPEM. The SPEMasm presents the structural and behavioral elements that allow simulations to be carried on. A case study is presented, where a small process adapted from OpenUP/Basic is simulated in the abstract state machine presented in this paper.*

1. Introdução

O software tem desempenhado um papel crítico e essencial na sociedade, qualquer produto ou serviço moderno tem como componente ou utiliza software em algum momento [FUGGETTA 2000]. Desde o final da década de 60 a comunidade de Engenharia de Software realiza pesquisas e desenvolve práticas para a especificação e desenvolvimento de software com foco na redução dos custos e de retrabalho necessários para o seu desenvolvimento e manutenção [BONA 2002]. Apesar dos esforços, a literatura técnica descreve como "Crise do Software" o fenômeno que reflete a incapacidade da indústria de software em atender satisfatoriamente às necessidades de um mercado consumidor cada vez mais exigente [PRESSMAN, 2001]. Somente nos Estados Unidos foram gastos U$250 bilhões com o desenvolvimento de software no ano de 2002, sendo que destes, U$38 bilhões foram perdidos em projetos mal sucedidos e U$17 bilhões foram acima do previsto nos projetos [STANDISH 2003].

Segundo [FUGGETTA 2000], um processo de desenvolvimento de software é definido como um conjunto de políticas, estruturas organizacionais, ferramentas, procedimentos e artefatos que são necessárias para especificar, projetar, desenvolver,

distribuir e manter um produto de software. Cada organização tem necessidade de processos de desenvolvimento de software especificados de acordo com a sua realidade e as suas necessidades. Contudo, não há garantia de que o uso de processos seja um remédio para todos os males da indústria de software, mas é um caminho promissor que já se mostrou efetivo em outras industriais, por exemplo, na indústria automobilística.

A adaptação de um processo para uma determinada empresa é uma atividade complexa, cercada de riscos e incertezas. Para saber se o processo funcionará, uma boa prática é colocá-lo em uso no cotidiano, e fazer ajustes na medida do necessário. O ideal é que o processo possa ser modelado e colocado em execução por simulação antes de ser implantado, possibilitando ao menos os que problemas mais visíveis sejam corrigidos antes da implantação no ambiente real, evitando custos desnecessários e riscos envolvidos. Contudo, a modelagem de processos de software não é tarefa simples, já que deve prover apoio automatizado para a sua execução (*enactment*) e acomodar atividades desempenhadas por seres humanos, que não podem ser automatizadas, como reuniões e negociações [REIS 2002]. A modelagem de processos de software e as ferramentas que a apóiam passaram a ser um objetivo perseguido pela indústria de desenvolvimento de software. A partir da modelagem, e com ferramentas adequadas de simulação do modelo de processo, é possível visualizar o processo em execução antes de ser colocado em prática, criando um laboratório para *design* de processos de valor inestimável para as empresas.

Para permitir a execução do processo (*process enactment*) em um ambiente de simulação adequado é necessário que o modelo de processo esteja formalizado em uma linguagem que tenha o poder expressivo suficiente para exibir as propriedades estáticas do processo e para representar suas propriedades dinâmicas. A partir dessa descrição do processo, torna-se possível interpretar os comandos da linguagem e proporcionar a desejada simulação do processo, em um ambiente que permita a interação com os analistas e especialistas em definição de processos de software.

Dentro deste contexto este artigo apresenta uma máquina de estado abstrata para simulação de modelos de processo de software, definidas a partir da formalização dos elementos do *SPEMasm,* adaptado de [OMG 2005]. No item 2 é apresentada a formalização de modelos de processos de software com máquinas de estado abstratas (*Abstract State Machine – ASM*), a partir do *SPEMasm*. O item 3 apresenta um estudo de caso realizado com a máquina de estado abstrata definida na linguagem *AsmetaL* e no item 4 a conclusão é apresentada.

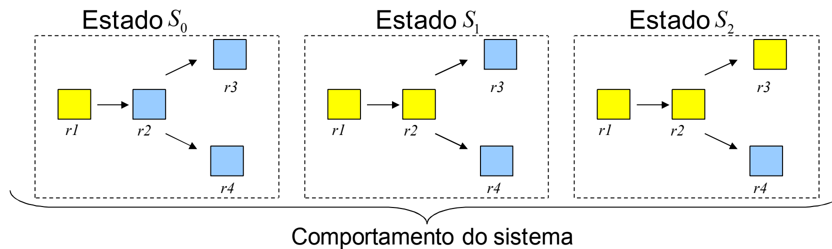
2. Formalização de Modelos de Processo de Software com Máquinas de Estado Abstratas

Máquinas de Estado Abstratas (*Abstract State Machines - ASM*) são um formalismo criado por Yuri Guerevich, originalmente conhecido por *Evolving Algebras*. Constituem um conceito expressivo e elegante para modelagem matemática de sistemas dinâmicos discretos [DI IORIO 2001]. Valente (1999) define que constituem um modelo computacional capaz de especificar qualquer algoritmo seqüencial em seu nível natural de abstração. Segundo Ober (2001), ASM tem sido utilizada para especificar a semântica de diversas linguagens de programação, como C++, Prolog, para descrever

especificações de hardware da IEEE como também para modelagem formal de algoritmos e descrição de arquitetura de hardware.

A execução de um programa ASM consiste no disparo de regras de transição que fazem com que a máquina mude de estados sucessivamente, até endereçar o estado final. Uma regra de transição de ASM tem a forma de um programa de uma linguagem imperativa, a principal diferença é a ausência de iteração explícita, já que o conceito está implícito na execução da máquina [DI IORIO 2001].

A Figura 1 apresenta um diagrama que exibe o comportamento de um programa ASM, dado pela mudança de estados.



Fonte: Elaborado pelo autor

Figura 1 - Representação gráfica dos estados a partir de uma computação.

No estado S_0 uma função, representada pelos nós, tem seu valor modificado pelas regras r_n se a condição para sua atualização (mudança de valor) for verdadeira, como a seguir:

if *Condition* ***then*** *Updates*

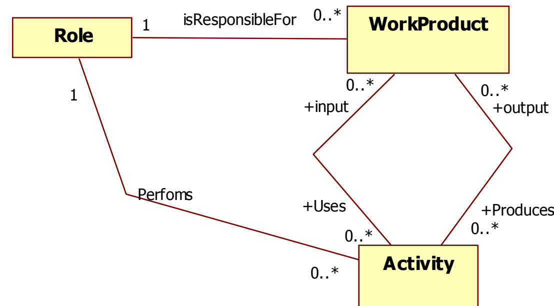
O disparo das regras é realizado de forma paralela, desde que as condições para a execução da regra sejam satisfeitas no estado corrente para as funções apresentadas no vocabulário, já que um programa ASM pode ser executado para cada agente definido. Os agentes executam de forma concorrente seus programas ASM e interagem globalmente compartilhando os endereços dos estados (OBER, 2001). Neste artigo, agentes são representados pelos atores do processo (*ProcessRole*), que desempenham funções, tais como Analistas de Sistemas, Programadores, Gerente de Projetos, Arquitetos de Software, dentre outros [OMG 2005].

Outros estados são alcançados de acordo com a verificação realizada por cada regra, até que o estado final seja alcançado e nenhuma condição seja verdadeira para o início de nenhuma outra regra presente no vocabulário, finalizando assim a execução da ASM.

2.1. *SPEMasm*: uma adaptação do SPEM para simulação em máquinas de estado abstratas

Desenvolvido pelo *Object Management Group* (OMG), o *Software Process Engineering Metamodel* (SPEM) é um meta-modelo que pode ser usado para descrever um processo concreto ou uma família de processos de desenvolvimento de software relacionados. É classificada como uma linguagem de modelagem de processo não-executável, pois a execução (*enactment*) do processo não faz parte do seu escopo [OMG 2005].

A relação básica entre os principais conceitos do SPEM é apresentada na Figura 2.



Fonte: [OMG 2005]

Figura 2 – Relação entre os conceitos de processo

Os atores são responsáveis pela realização das atividades (*Activity*) e assumem papéis (*Role*) no processo. A responsabilidade sobre os produtos de trabalho (*WorkProduct*) é dos atores que ao executarem atividades, podem utilizar tais produtos de trabalho como entrada ou saída para sua realização, indicando, por exemplo, que um documento ou diagrama é produzido em uma atividade. Kruchten (2003) enfatiza que o *Role* não é apenas um indivíduo ou um cargo, mas a responsabilidade e o comportamento que um indivíduo ou um grupo de indivíduos têm que assumir durante o processo. O comportamento é definido de acordo com as atividades que o ator desenvolve, já a responsabilidade é definida de acordo com os produtos de trabalho (*WorkProduct*) mantidos pelo ator, ao desempenhar determinado papel. O papel é desempenhado de acordo com a necessidade no momento em que o projeto acontece e de acordo com o que foi pré-estabelecido nas reuniões de planejamento. Pode ser necessário que um indivíduo atue como Analista de sistemas em um momento e em outro, como Homologador de software, ou Revisor de qualidade.

A realização de algumas atividades demanda informações produzidas em outra atividade do processo, e podem utilizar outros produtos de trabalho, produzidos em atividades anteriores do processo de desenvolvimento.

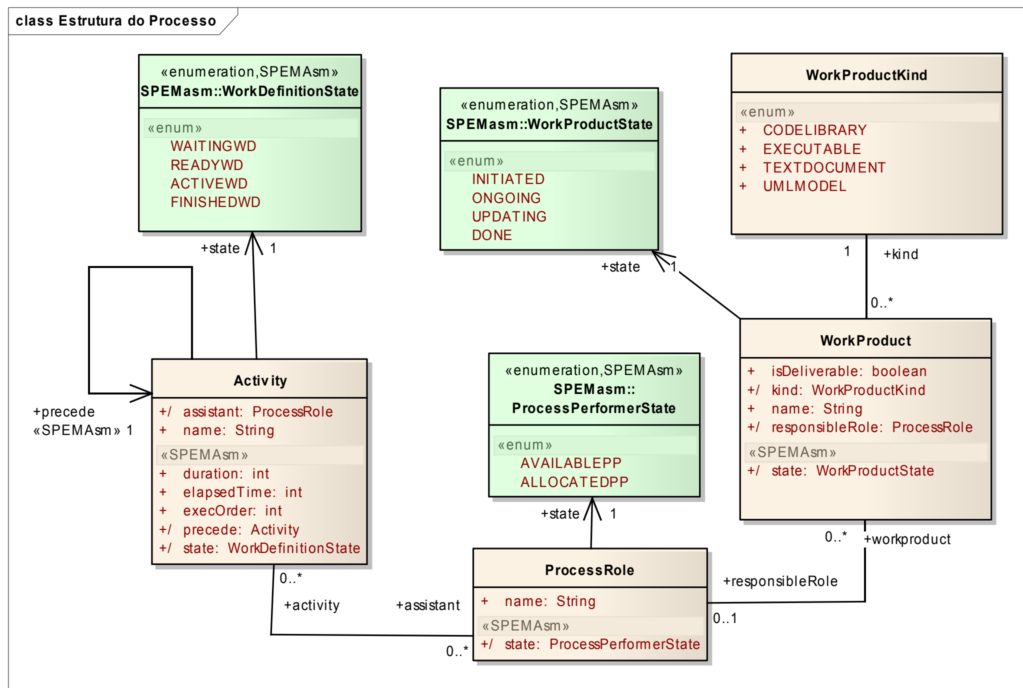
O SPEM é organizado em pacotes cuja divisão é feita em SPEM_*Foundations,* que trata blocos de construção baseados na UML 1.4 e o SPEM_*Extensions*, que descreve a semântica e os construtores para a representação de processos de software [OMG, 2005]. Neste artigo os pacotes *ProcessStructure* (Estrutura do processo) e *ProcessLifeCycle* (Ciclo de Vida do Processo) são considerados, já que representam os principais elementos para modelagem de processo de software.

O meta-modelo SPEM [OMG 2005] foi estendido para o meta-modelo denominado *SPEMasm* com o objetivo de permitir a simulação de processos de software em Máquinas de Estado Abstratas. A extensão visa especificamente:

- Adicionar comportamento ao SPEM;
- Permitir que o SPEM seja instanciado para projetos de software, estendendo assim a proposta da OMG (2005);
- Possibilitar sua abstração para Máquinas de Estado Abstratas, de forma que não seja necessária uma infra-estrutura tecnológica para a execução do processo.

O pacote estrutural do SPEM, versão 1.1, fornece um conjunto genérico de estruturas que permitem a representação de qualquer processo de software [OMG, 2005]. A composição das estruturas objetiva principalmente mostrar a relação entre elementos do modelo, como também fornece descrições básicas de atributos destes elementos.

O elemento *Activity* apresenta apenas atributos para a descrição de como uma atividade é relacionada aos demais elementos do pacote estrutural. A adaptação realizada no pacote estrutural no *SPEMasm* para permitir a simulação de modelos de processo de software é apresentada na Figura 3.



Fonte: Adaptado de [OMG 2005]

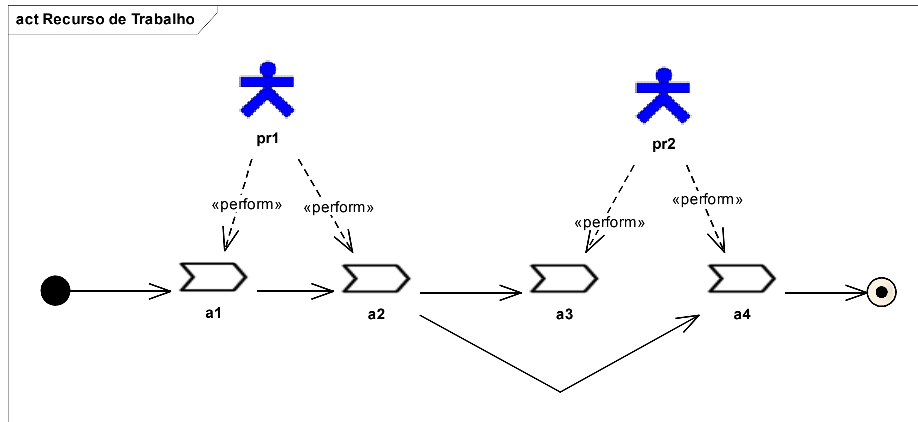
Figura 3 – Pacote Estrutural de Processo do *SPEMasm*

Foram adicionados ao elemento *Activity* os atributos: *duration, elapsedTime, execOrder, precede* e *state*. Tais atributos são estereotipados no modelo com <<*SPEMasm*>> para identificar as adições realizadas. O propósito do atributo *duration* é permitir que o gerente do processo informe a duração estimada da atividade. A duração deve ser informada em unidades inteiras de tempo, como dias, horas ou semanas. Como o foco não é a execução em ambiente de produção, valores não discretos podem ser abstraídos ou arredondados. Importante salientar que a duração das atividades não é modificada durante a simulação, portanto deve ser estimada fora do modelo e informada.

O atributo *elapsedTime* tem como objetivo permitir que a máquina de estados abstrata proposta neste artigo possa simular o tempo decorrido de uma atividade, assim que ela for iniciada pelo fluxo de trabalho, que é descrito pelos atributos *execOrder* e *precede*. A ordem de execução das atividades deve ser definida pelo atributo *precede*, sendo assim, uma atividade *a1*, precede *a2* se: *precede*(*a2*) := *a1*. O atributo *execOrder*

é definido pela máquina de estado abstrata de acordo com a precedência padrão informada pelo atributo *precede,* que aqui é denominada *precedência por fluxo de atividade*, ou ainda por precedências de *recurso de trabalho* e de *produto de trabalho*.

A ASM definida considera 3 tipos de precedência: precedência por fluxo de atividades, precedência por recursos de trabalho e precedência por produtos de trabalho. A precedência por recurso de trabalho é avaliada quando em uma rede de atividades, a precedência de fluxo de atividade permite a execução da atividade *a4* em paralelo à atividade *a3*, porém o ator *pr2* (*ProcessRole*) é o mesmo para as ambas as atividades, conforme apresentado na Figura 4.



Fonte: Elaborado pelo autor

Figura 4 – Precedência de Recurso de Trabalho

No exemplo da Figura 4 a precedência por fluxo de atividade, que é definida pelo modelo, é dada pela seqüência:

$$precede(a1) := \emptyset,\ precede(a2) := a1,\ precede(a3) := a2,\ precede(a4) := a2$$

Pela regra de precedência de fluxo de atividade, *a3* e *a4* podem ser executadas em paralelo, e ASM define inicialmente os seguintes valores para *execOrder:*

$$execOrder(a1) := 1,\ execOrder(a2) := 2,\ execOrder(a3) := 3,\ execOrder(a4) := 3$$

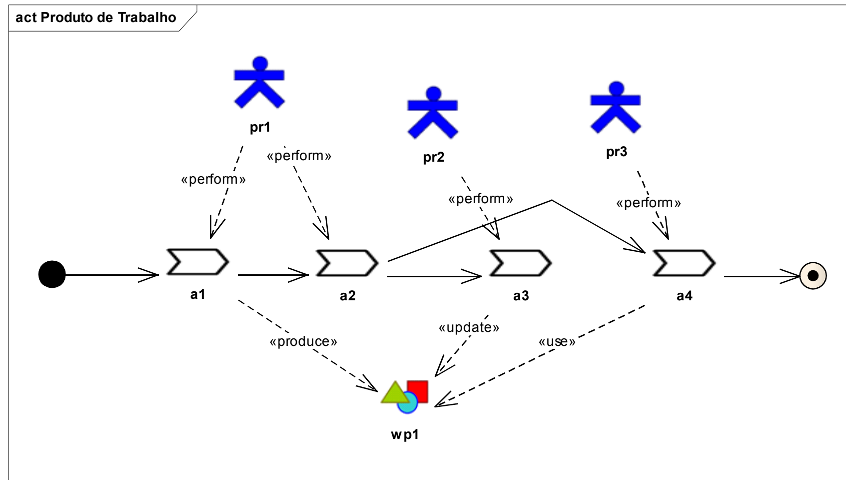
Antes que o trabalho das atividades seja iniciado, a ASM avalia cada recurso de trabalho e quais atividades esse recurso realiza (*perform*), e redefine a ordem de execução, como também a precedência:

$$execOrder(a1) := 1,\ execOrder(a2) := 2,\ execOrder(a3) := 3,\ execOrder(a4) := 4$$

E redefine a precedência para:

$$precede(a1) := \emptyset,\ precede(a2) := a1,\ precede(a3) := a2,\ precede(a4) := a3$$

A precedência por produto de trabalho também é verificada pela ASM, quando um produto de trabalho produzido por uma atividade *a1* (*produce*), é utilizado (*use*) na atividade *a4*, mas é atualizado (*update*) pela atividade *a3* como é apresentado na Figura 5.



Fonte: Elaborado pelo autor

Figura 5 – Precedência de Produto de Trabalho

Na precedência de produtos de trabalho, a ASM avalia o uso (*use*) e a atualização (*update*) dos produtos de trabalho. Neste caso, o *execOrder* é modificado para 4, já que a atividade *a3* atualiza um produto de trabalho que é utilizado pela atividade *a4,* conforme apresentado a seguir:

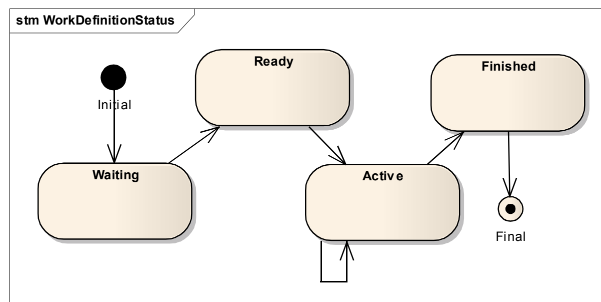
execOrder(a1):= 1, execOrder(a2):= 2, execOrder (a3):= 3, execOrder(a4):= 4

E redefine a precedência para:

precede(a1):= ∅, *precede*(a2):= a1, *precede*(a3):= a2, *precede*(a4):= a3

As regras que verificam as precedências de produtos de trabalho e recursos de trabalho são executadas em paralelo pela máquina, portanto não são mutuamente exclusivas.

Finalmente, o atributo *state* de *Activity* é utilizado para controlar o ciclo de vida de definições de trabalho (*WorkDefinition*), durante a simulação do processo pela ASM. Os estados possíveis para uma definição de trabalho são apresentados no elemento *WorkDefinitionState* e a transição de estados para a atividade é demonstrada na Figura 6.



Fonte: Adaptado de (REIS, 2003)

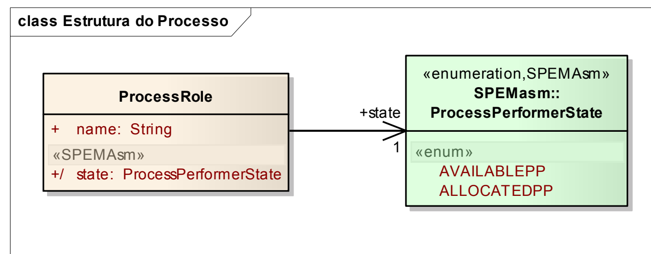
Figura 6 – Estados possíveis para uma atividade.

Conforme adaptação realizada na transição proposta por Reis (2003), a atividade pode assumir os seguintes estados, ou seja, o atributo *state* pode assumir os seguintes valores:

- –***Waiting***: aguardando que o agente (*ProcessRole*) inicie o trabalho;
- –***Ready***: pronta para iniciar o trabalho, pois foi colocada na fila de execução pelo agente, considerando a precedência.
- –***Active***: a atividade está em execução, e permanece, até que o tempo decorrido (*elapsedTime*) seja igual ao tempo estimado(*duration*) para a atividade.
- –***Finished***: o trabalho da atividade foi finalizado, o recurso de trabalho (*ProcessRole*) foi desalocado e os produtos de trabalho atualizados ou produzidos nesta atividade foram finalizados.

Neste artigo, optou-se por adaptar a transição de estados proposta por Reis (2003) e eventuais Falhas, Cancelamentos e Pausas não são considerados para as atividades, pois podem requerer ação humana, que pode ser simulada por agentes inteligentes, ou modelos dinâmicos, não cobertos pela proposta deste trabalho. Contudo a abordagem orientada a objetos utilizada pelo SPEM permite que novos elementos sejam incorporados, como também adicionadas regras de transição para mudança dos estados na especificação ASM descrita.

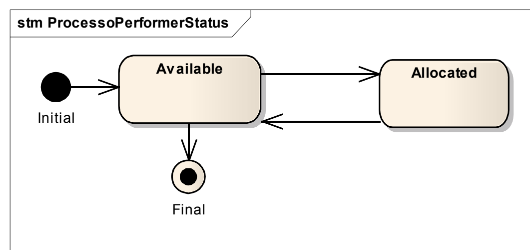
A adaptação realizada no elemento *ProcessRole* permite que seu estado ao longo da simulação de um processo seja atualizado pela ASM. A Figura 7 apresenta o elemento *ProcessRole*, e seus possíveis estados.



Fonte: Adaptado de (OMG, 2005)

Figura 7 – Elemento *ProcessRole* adaptado

Os estados para um recurso de trabalho foram adaptados de Reis (2003), e são apresentados na Figura 8.



Fonte: Adaptado de (REIS, 2003)

Figura 8 – Estados de um *ProcessRole*.

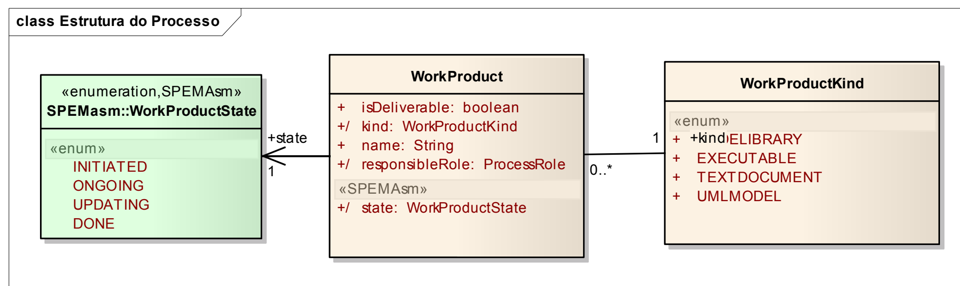
Um ator pode assumir os seguintes estados:

- – ***Available***:o ator está disponível para realizar uma atividade;

– ***Allocated***:o ator está alocado em uma atividade e não pode realizar outra atividade.

No diagrama apresentado na Figura 8, o estado inicial do ator é disponível (*Available*) e logo que ele inicia o trabalho em uma atividade, é alocado (*Allocated*), voltando a ficar disponível assim que o trabalho é finalizado.

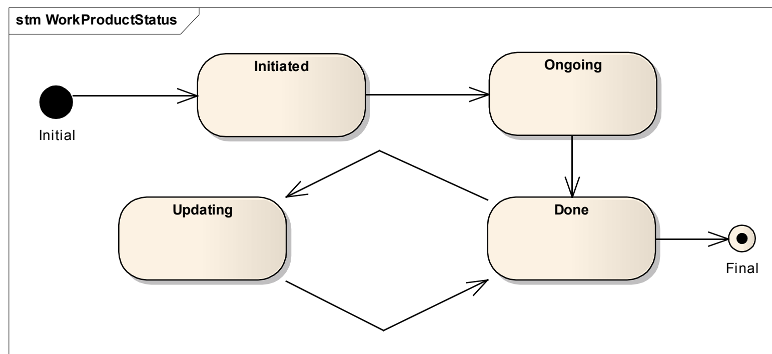
O elemento *WorkProduct* foi adaptado para que os estados pelos quais um produto de trabalho passa durante a instanciação de um processo de desenvolvimento de software sejam considerados. Figura 9 apresenta o *WorkProduct*.



Fonte: Adaptado de (OMG, 2005)

Figura 9 – Elemento *WorkProduct*

O atributo *state* foi adicionado e seu ciclo de vida é apresentado na Figura 10.



Fonte: Adaptado de (REIS, 2003)

Figura 10 – Estados de um *WorkProduct*.

Um produto de trabalho pode assumir os seguintes estados:

– ***Initiated***: o produto de trabalho é iniciado assim que a ASM entra em execução;

– ***Ongoing***: assim que o produto de trabalho começa a ser produzido ou atualizado por uma atividade, ele está em andamento, ou em processo de produção;

– ***Done***: o produto de trabalho é concluído assim que a atividade termina de produzí-lo ou atualizá-lo;

– ***Updating***: Depois que um produto de trabalho é produzido ou atualizado por outras atividades, uma atividade pode atualizá-lo.

2.2. Máquinas de Estado Abstratas: adicionando comportamento ao *SPEMasm*

O comportamento do processo é simulado utilizando-se regras da ASM. Cada estado possível para cada domínio é representado pelos domínios estáticos enumerados, associados aos domínios abstratos dos pacotes *ProcessStructure* e *ProcessLifeCycle* do *SPEMasm*. Neste trabalho os elementos do *SPEMasm* são formalizados em ASM segundo a Tabela 1.

Tabela 1 – Formalização do *SPEMasm* para ASM

Elemento	*SPEMasm*		ASM
Estrutura	*ProcessStructure*	*WorkDefitinitionState, WorkProductState, ProcessPerformState, WorkProductKind*	Domínio estático enumerado pré-definido
		Activity, WorkProduct	Domínios abstratos
		ProcessRole	Agente
		Atributos de *Activity*, *WorkProduct* e *ProcessRole*	Funções estáticas e dinâmicas
		Associações	Funções estáticas
		Instâncias	Funções estáticas
	ProcessLifeCycle	*Iteration, Phase*	Domínios abstratos
		Precondition e Goal	Funções dinâmicas
		Associações	Funções estáticas
Comportamento	*Diagramas de Estados*	*WorkDefitinitionState* para *Activity, Phase e Iteration,*	Regras de Transição
		WorkProductState, ProcessPerformState,	Regras de Transição
		Restrições	Axiomas (Predicados)

Adaptado de (PARK *et al*, 2007)

Para as definições de trabalho (*WorkDefinition)*, por exemplo, atividades, iterações e fases, o ciclo de vida apresentado na Figura 6 é formalizando como:

enum domain *WORKDEFINITIONSTATE = { WAITINGWD | READYWD | ACTIVEWD | FINISHEDWD }*

Conforme apresentado na Figura 10, todo produto de trabalho que esteja sendo atualizado por uma atividade, é concluído quando a atividade é finalizada. A regra formal que estabelece tal definição é apresentada no Quadro 1.

```
MonitoreUpdatingWorkProducts ≡
  ∀ w ∈ WORKPRODUCT | State(w) = UPDATING ∧ isNotEmpty(UpdatedBy (w)) :
     if State(FIRST(sequence(UpdatedBy (w)))= FINISHEDWD then
        State(w) := DONE
     Endif
```

Fonte: Elaborado pelo autor

Quadro 1 – Regra *MonitoreUpdatingWorkProducts*

Toda atividade tem sua ordem de execução (*ExecOrder*) definida pela função *DefineDefaultOrders* apresentada no Quadro 2.

```
DefineDefaultOrders ≡
      ∀ a ∈ ACTIVITY
       if IsUndef(ExecOrder(a)) then
            ExecOrder (a) := ExecOrder (Precede(a))  + 1
       endif
```

Fonte: Elaborado pelo autor

Quadro 2 – Regra *DefineDefaultOrders*

A regra *MonitoreActivitiesWithUseWP* descrita no Quadro 3 estabelece que quando uma atividade a_{n+1} use um produto de trabalho que seja produzido por uma

atividade a_n da rede de atividades, mas sua ordem de execução seja menor que a ordem de execução da atividade a_n, a nova precedência de a_{n+1}, passa a ser a_n, e sua ordem de execução defina por *ExecOrder(*a_n*)+1,*descrita na seção 2.1.

```
MonitoreActivitiesWithUseWP ≡
∀ a ∈ Activity :
    if isNotEmpty(set Use(a)) then
        let (act = FIRST(sequence (ProducedBy (FIRST(sequence (Use (a)))))) in
                if ExecOrder (act) >= ExecOrder(a) then
                        Precede(a) := act, ExecOrder(a) := ExecOrder (act) +1
                endif
        endlet
    endif
```

Fonte: Elaborado pelo autor

Quadro 3 – Regra *MonitoreActivitiesWithUseWP*

Outra avaliação feita pela ASM é dada pela situação quando a função *Update* de uma atividade é estabelecida na regra *DefineWorkDefinitionParameters.* Neste caso a precedência da atividade cuja função *Update* é definida pode ser modificada, conforme a regra *MonitoreActivitiesWithUpdateWP* descrita no Quadro 4.

```
MonitoreActivitiesWithUpdateWP ≡
∀ a ∈ Activity :
    if isNotEmpty(set Update(a)) then
        let (act = FIRST(sequence (ProducedBy (FIRST(sequence (Update (a)))))) in
                if ExecOrder (act) >= ExecOrder(a) then
                        Precede(a) := act, ExecOrder(a) := ExecOrder (act) + 1
                endif
        endlet
    endif
```

Fonte: Elaborado pelo autor

Quadro 4 – Regra *MonitoreActivitiesWithUpdateWP*

A regra *MonitoreActivitiesWithUpdateWP* estabelece que quando uma atividade a_n atualize um produto de trabalho que seja utilizado por uma atividade a_{n+1} da rede de atividades, e a ordem de execução da atividade a_n seja maior que à ordem de execução da atividade a_{n-1} que inicialmente produziu o produto de trabalho, a nova precedência de a_{n+1}, passa a ser a_n, e sua ordem de execução defina por *ExecOrder (*a_n*)+1.*

Neste artigo é definido que um ator pode desempenhar somente uma atividade de cada vez, portanto não há compartilhamento de tempo para a realização de atividades. Dado um conjunto de atividades $a_1 ... a_n$ que determinado ator deve desempenhar no projeto, o ator deve realizar a atividade a_2 caso tenha concluído a atividade a_1. A regra *MonitoreActivitiesPerformed* estabelece tal definição e é apresentada no Quadro 5.

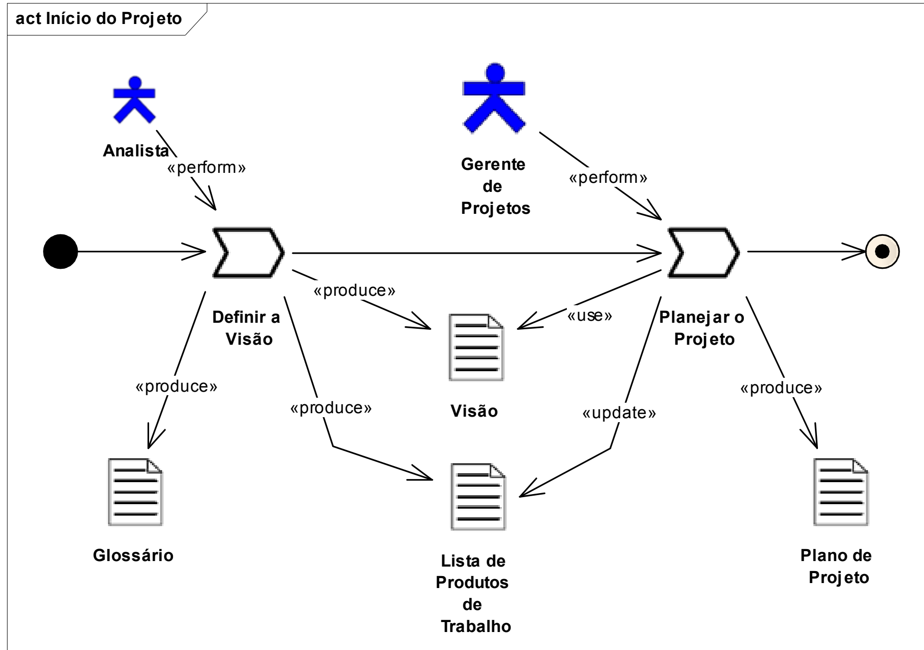
```
MonitoreActivitiesPerformed ≡
∀ a ∈ Activity | sequence Activity(FIRST(sequence(Assistant(a))) ⊃ a :
        let (act = FIRST(sequence (Activity(self)))) in
            if act ≠ a ∧ ExecOrder (act) = ExecOrder(a) ∧ Assistant (act) = Assistant(a) then
                Precede(a) := act, ExecOrder(a) := ExecOrder (act) + 1
            endif
        endlet
    endif
```

Fonte: Elaborado pelo autor

Quadro 5 – Regra *MonitoreActivitiesPerformed*

3. Estudo de Caso: simulação de modelos de processos de software

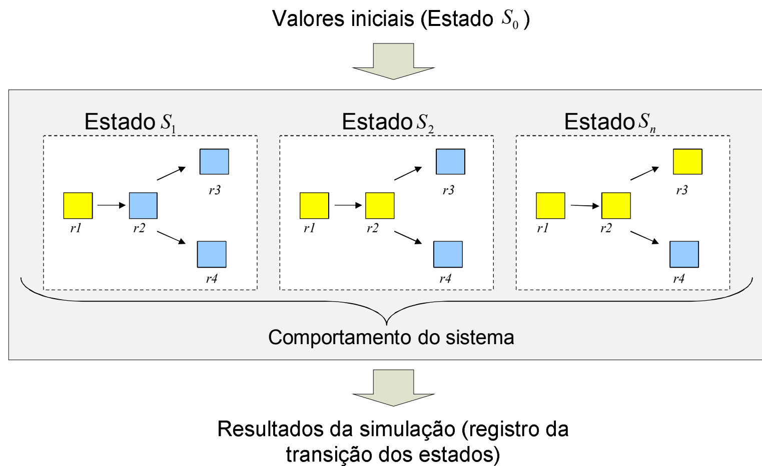
O estudo de caso utiliza como modelo de processo uma adaptação realizada no *OpenUP/Basic* [KROLL 2006] apresentada parcialmente na Figura 11.



Fonte: Adaptado de (IBM, 2008)

Figura 11 – Iteração *Início do Projeto*

A máquina de estado abstrata descrita neste artigo representa a estrutura do processo (*ProcessStructure*) como também o comportamento por meio das regras de transição parcialmente apresentadas na seção 2 utilizando a linguagem *AsmetaL* [ASMETA 2007]. Além da estrutura e do comportamento é necessário que o programa *ASM* receba valores (instâncias) para um projeto. O esquema de funcionamento da ASM é apresentado na Figura 12.



Fonte: Elaborado pelo autor

Figura 12 – Funcionamento da máquina de estado abstrata.

Os valores iniciais (estado S_0) com as instâncias são utilizados como entrada para a simulação, e são modificados pelas regras de transição, gerando um registro (*log*) de todos os estados desde o inicial (estado S_0) até o estado final (estado S_n).

A simulação é executada no *Eclipse*[1] com o plugin *Asmee*[2] [ASMETA 2007] e utiliza informações de duração de um projeto interno como entrada (estado S_0) para produzir o registro da transição de estados de todos os elementos do projeto.

O registro (*log)* de cada transição permite ao Engenheiro de Processo identificar eventuais erros na alocação de pessoal, na determinação de prazos para as atividades e na dependência de produtos de trabalho para a realização de atividades no projeto.

O modelo de processo adaptado foi instanciado para um projeto denominado *Estudo de Caso*, cuja cronograma é apresentado na Figura 13.

Id	Nome da tarefa	Duração	Início	Término	Predecessora	Nomes dos recursos
1	**Estudo de Caso**	**14 dias**	**Qua 2/1/08**	**Seg 21/1/08**		
2	**Concepção**	**10 dias**	**Qua 2/1/08**	**Ter 15/1/08**		
3	**Início do Projeto**	**6 dias**	**Qua 2/1/08**	**Qua 9/1/08**		
4	a1 - Definir a Visão	3 dias	Qua 2/1/08	Sex 4/1/08		Analista
5	a2 - Planejar o Projeto	3 dias	Seg 7/1/08	Qua 9/1/08	4	Gerente de Projeto
6	**Planejar e Gerenciar a Iteração**	**4 dias**	**Qui 10/1/08**	**Ter 15/1/08**		
7	a3 - Planejar a Iteração	2 dias	Qui 10/1/08	Sex 11/1/08	5	Gerente de Projeto
8	a4 - Gerenciar a Iteração	2 dias	Seg 14/1/08	Ter 15/1/08	7	Gerente de Projeto
9	**Elaboração**	**6 dias**	**Seg 14/1/08**	**Seg 21/1/08**		
10	**Requisitos e Arquitetura**	**6 dias**	**Seg 14/1/08**	**Seg 21/1/08**		
11	a5 - Levantar e detalhar requisitos	6 dias	Seg 14/1/08	Seg 21/1/08	7	Analista
12	a6 - Detalhar a arquitetura	5 dias	Seg 14/1/08	Sex 18/1/08	7	Arquiteto de Software

Fonte: Elaborado pelo autor

Figura 13 – Informações do projeto Estudo de Caso.

Na Figura 13, as linhas 2 e 9 representam as fases do modelo, composta por iterações, apresentadas nas linhas 3, 6 e 10. As iterações *Início do Projeto* e *Planejar e Gerenciar a Iteração* compõem a fase *Concepção*, conforme apresentado no cronograma. No projeto proposto a unidade de tempo utilizada é o *dia,* e a data inicial do projeto foi estabelecida para o dia 02/01/2008. A determinação da data de término é dada pela soma da data inicial da atividade com a duração em dias, no entanto a máquina de estado abstrata proposta não realiza cálculo de datas. As precedências de atividades são apresentadas nas linhas 5, 7, 8, 11 e 12. Por exemplo, a atividade *Planejar o Projeto* inicia-se após o término da atividade *Definir a Visão.*

A mudança da precedência é realizada no estado S_2 apresentado na Tabela 2.

Tabela 2 – Registro dos estados da precedência.

Estado	*precede(a1)*	*precede(a2)*	*precede(a3)*	*precede(a4)*	*precede(a5)*	*precede(a6)*
S0	undef	a1	a2	a3	a3	a3
S1	undef	a1	a2	a3	a3	a3
S2	undef	a1	a2	a3	a3	a5
S3	undef	a1	a2	a3	a3	a5

Fonte: Elaborado pelo autor

No estado S_2 ocorre a mudança da precedência da atividade *a6*, que anteriormente tinha como predecessora a atividade *Gerenciar a Iteração* (*a3*) e após a execução da regra *MonitoreActivitiesWithUpdateWP* modifica sua predecessora para

[1] http://www.eclipse.org/

[2] http://asmeta.sourceforge.net/userdoc/asmee.html

atividade *Levantar e Detalhar Requisitos* (*a5*), já que *a5* atualiza o documento produzido na atividade *a3*, que é utilizado pela atividade *Detalhar a Arquitetura* (*a6*).

A ordem de execução da atividade *Detalhar a arquitetura* (*a6*) é modificada pela regra *MonitoreActivitiesWithUpdateWP* no estado, conforme apresentado na Tabela 3.

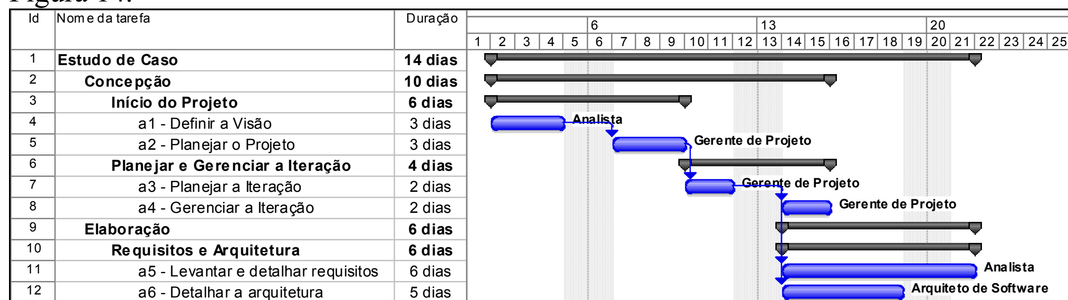
Tabela 3 – Registro dos estados da ordem de execução.

Estado	*execOrder(a1)*	*execOrder(a2)*	*execOrder(a3)*	*execOrder(a4)*	*execOrder(a5)*	*execOrder(a6)*
S0	1	2	Undef	Undef	Undef	undef
S1	1	2	3	Undef	Undef	undef
S2	1	2	3	4	4	5

Fonte: Elaborado pelo autor

As tabelas 2 e 3 foram formatadas para apresentação neste trabalho a partir do registro (*log*) em formato texto separado por vírgula, produzido pelo plugin *Asmee* utilizado para executar a simulação.

A ordem de execução é informada apenas para a primeira atividade da cadeia de atividades de um projeto, no entanto, considerando a precedência declarada na Figura 13 é possível definir que a atividade *Detalhar a arquitetura* (*a6*) seria executada em paralelo com as atividades *a5* e *a4*, conforme demonstrado no gráfico de *Gantt* na Figura 14.



Fonte: Elaborado pelo autor

Figura 14 – Gráfico de Gantt do projeto

4. Conclusões

Este artigo demonstrou o uso de Máquinas de Estado Abstratas (ASM) como uma alternativa viável para simulação de modelos de processo de desenvolvimento de software. A ASM definida permite a expansão para considerar outros fatores não cobertos neste trabalho, através da adição de elementos nos modelos estrutural e comportamental no *SPEMAsm* e regras de transição na ASM. Muitos são os métodos e técnicas definidos na literatura técnica que solucionam (ou propõem soluções) para tais fatores, e estes também podem ser incluídos no ASM.

O estudo de caso conduzido demonstrou que a definição e o conhecimento acerca das regras de funcionamento de um processo de desenvolvimento de software são essenciais no momento da alocação de recursos e dão estabelecimento da rede de atividades. O estudo de caso foi importante ainda para demonstrar que o modelo de processo deve ser modificado e revisto constantemente já que se configura a cada projeto de maneira ímpar, e o tempo disponível para avaliar os modelos de processo e propor melhorias é curto, sobretudo nas organizações onde não há melhoria processo e modelos de maturidade estabelecidos.

Referências

Bona, Cristina. **Avaliação de Processos de Software: Um estudo de caso em XP e ICONIX**. Dissertação de Mestrado - Universidade Federal de Santa Catarina, Florianópolis, 2002.

Fuggetta, Alfonso. **Software process: a roadmap.** In: Proceedings of the Conference on the Future of Software Engineering. Limerick, Ireland, June 04 - 11, 2000. ICSE '00. ACM Press, New York, NY, 25-34.

Standish Group. **Latest Standish Group CHAOS Report Shows Project Success Rates Have Improved by 50%**. 25 de Março de 2003. Disponível em http://www.standishgroup.com/press/article.php?id=2. Consultado em 15 de Junho de 2007.

Pressman, Roger S. **Software Engineering: A Practitioner's Approach**, 5. ed., McGraw-Hill, Boston, 2001.

Reis, Carla Alessandra. **Uma Abordagem Flexível para Execução de Processos de Software Evolutivos.** 2003. 267 f. Tese (Doutorado) - Curso de Ciência da Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2003.

Di Iorio, Vladimir Oliveira. **Avaliação Parcial de Máquinas de Estado Abstratas.** 2001. 163 f. Tese (Doutorado) - Curso de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, 2001. Cap. 2.

Valente, M. T. O; Bigonha, R. S.; Maia, M. A.; Loureiro, A. A. F. **Aplicação de ASM na Especificação de Sistemas Móveis**. II Workshop de Métodos Formais (WMF), p. 60-69. 1999.

OMG. **Software Process Engineering Metamodel Specification (SPEM) - version 1.1**. 2005. Disponível em: < www.omg.org/docs/formal/05-01-06.pdf >.

Park, SeungHun et al. **Deriving Software Process Simulation Model from SPEM-based Software Process Model**. APSEC 2007: 382-389

Kroll, Per; MACISAAC, Bruce. **Agility and Discipline Made Easy**: Practices from OpenUP and RUP: Addison-wesley, 2006. 448 p. (Professional).