UNIVERSIDADE FEDERAL DE SANTA CATARINA

APLICAÇÃO DE ESTRATÉGIAS PARA CONFIGURAÇÃO INTELIGENTE DE ATIVIDADES EM UM SISTEMA DE GERENCIAMENTO DE WORKFLOW HOSPITALAR

Luiz Fernando da Silva Pereira

Florianópolis, outubro de 2005.

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE SISTEMAS DE INFORMAÇÃO

APLICAÇÃO DE ESTRATÉGIAS PARA CONFIGURAÇÃO INTELIGENTE DE ATIVIDADES EM UM SISTEMA DE GERENCIAMENTO DE WORKFLOW HOSPITALAR

Trabalho de conclusão de curso submetido à Universidade Federal de Santa Catarina como requisito parcial para a obtenção do grau de Bacharel em Sistemas de Informação

Luiz Fernando da Silva Pereira

Florianópolis, outubro de 2005.

LUIZ FERNANDO DA SILVA PEREIRA

APLICAÇÃO DE ESTRATÉGIAS PARA CONFIGURAÇÃO INTELIGENTE DE ATIVIDADES EM UM SISTEMA DE GERENCIAMENTO DE WORKFLOW HOSPITALAR

Trabalho de conclusão de curso apresentado como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação

____
Orientador: Prof. Dr. rer.nat. Aldo von Wangenheim

____
Coorientador: Bsc. Manassés Ribeiro

Banca examinadora

____
Prof. Dr. Mauro Roisenberg

____
Msc. Daniel Duarte Abdala

____
Bsc. Levi Ferreira

AGRADECIMENTOS

Aos meus pais, Antônio e Vera, por todo o carinho e incentivo, desde os primeiros passos da minha vida.

À minha namorada, Jeanifer, pelo amor e pelo companheirismo ao longo desses quase cinco anos ao meu lado.

Ao meu amigo e co-orientador, Manassés, pelo amparo e incentivo em todas as horas em que precisei.

Ao meu orientador, Prof. Aldo, pela confiança e pela oportunidade cedida.

Aos integrantes do Projeto Cyclops que se dispuseram a me ajudar nos momentos de dúvida.

Ao pessoal do Centro de Informações Toxicológicas de Santa Catarina (CIT/SC), pelas preciosas dicas e por me compreenderem e me incentivarem nessa reta final de TCC.

A todos os meus amigos que, de alguma forma, contribuíram para a conclusão bem sucedida deste curso ou que torceram para que isso acontecesse.

A Deus.

SUMÁRIO

1 INTRODUÇÃO 1

1.1 Contextualização do estudo 1
1.1.1 O controle de atividades em um ambiente hospitalar 1
1.1.2 Aplicação de Workflow no Ambiente hospitalar 3
1.1.3 Máquina de inferência e Tecnologia de *workflow* 4
1.1.4 O projeto Cyclops 4

1.2 Apresentação do problema 5
1.2.1 A Alocação Inteligente de Recursos 6
1.2.2 Justificativas 6

1.3 Estrutura do trabalho 7

2 OBJETIVOS 9

2.1 Objetivo Geral 9

2.2 Objetivos Específicos 10

3 FUNDAMENTAÇÃO TEÓRICA 13

3.1 Planejamento 13
3.1.1 Introdução 13
3.1.2 Definição 14
3.1.3 Características 15
3.1.4 Classificação 16

3.2 Sistemas de Gerenciamento de Workflow 19
3.2.1 Introdução 19
3.2.2 Definição 20
3.2.3 Características 21
3.2.4 Classificação 23
3.2.5 Algumas soluções existentes 24

4 CYCLOPS MEDICAL WORKFLOW MANAGEMENT 26

4.1 Visão Geral do sistema 26
4.1.1 O modelador de Processos (Workflow Designer) 26
4.1.2 O Cliente (Workflow Client) 28
4.1.3 O Servidor (Workflow Engine) 29

4.2 O Motor de Inferência 31
4.2.1 Visão geral 31
4.2.2 Metas e operadores: Implementação 35

4.3 Aspectos do desenvolvimento do trabalho 36
4.3.1 Metodologia adotada 36
4.3.2 Definição do Modelo de classes de *workflow* 37
4.3.3 O Controle de Agendamentos 41
4.3.4 O Algoritmo de Configuração de *Workflows* 42
4.3.5 Gerenciando as Alocações 46
4.3.6 O Desenvolvimento de Testes para Validação 47

5 RESULTADOS E DISCUSSÃO 50

5.1 Restrições do estudo 54

5.2 Sugestões para Trabalhos Futuros 55

REFERÊNCIAS 57

ANEXOS 59

ÍNDICE DE FIGURAS

Figura 1 – Idéia do Algoritmo de Configuração (RICHTER, 2001) ________ *16*
Figura 2 - Características de um Sistema de workflow (HOLLINGSWORTH, 2005) ________ *22*
Figura 3 - Tela principal do Cyclops Workflow Designer ________ *27*
Figura 4 - Tela principal do Cyclops Workflow Client ________ *28*
Figura 5 - Estrutura em camadas do Servidor de Workflow (RIBEIRO, 2005) ________ *30*
Figura 6 - Algoritmo contínuo da máquina de inferência ________ *34*
Figura 7 - Algoritmo de escalonamento de prioridades ________ *34*
Figura 8 - Seqüência de execução da redução de uma meta ________ *36*
Figura 9 - Diagrama de classes traduzindo o modelo de informações ________ *39*
Figura 10 - Particularidades dos recursos humanos e materiais ________ *40*
Figura 11 - Classes do pacote de agendamento ________ *42*
Figura 12 - Algoritmo de Configuração de Workflow ________ *43*
Figura 13 - Algoritmo Configuração de Atividades ________ *46*
Figura 14 - Tela do Visualizador de processos da máquina de inferência ________ *48*
Figura 15 - Atividade vista em detalhes através do visualizador de processos ________ *49*
Figura 16 – Diagrama de utilização de recursos ________ *49*
Figura 17 – Agendamentos de "Dr. Ribeiro" após 200 configurações ________ *51*
Figura 18 – Agendamentos de "Dr. Pereira" após 200 configurações ________ *51*
Figura 19 - Agendamentos de "Dr. Silva" após 200 configurações ________ *52*
Figura 20 - Agendamentos de "Dr. Ribeiro" após 1000 configurações ________ *53*
Figura 21 - Agendamentos de "Dr. Pereira" após 1000 configurações ________ *53*
Figura 22 - Agendamentos de "Dr. Silva" após 1000 configurações ________ *53*

RESUMO

Os ambientes hospitalares, de um modo geral, caracterizam-se pela dificuldade sistêmica ocasionada, principalmente, pela má utilização de recursos compartilhados. Esses recursos podem ser humanos (profissionais), materiais (aparelhos) ou físicos (instalações) e, comumente, apresentam disponibilidade limitada, apesar da alta demanda de utilização. Além disso, outra característica importante nesse tipo de ambiente é a natureza dinâmica da execução de seus processos, a qual requer agilidade e flexibilidade no seu gerenciamento, mas que, geralmente, não é contemplada de maneira adequada. Entretanto, medidas podem ser adotadas visando minimizar esses problemas. A utilização racionalizada dos recursos disponíveis, aliada ao gerenciamento adequado dos processos, apresenta-se como uma solução viável. A aplicação prática dessa solução é materializada pelo Projeto Cyclops, com o desenvolvimento de um Sistema de Gerenciamento de *Workflow*, dotado de um motor de inferência inteligente, e voltado à correta utilização de recursos de saúde. Nesse contexto, o presente trabalho constitui parte importante do sistema, e caracteriza-se por proporcionar uma maximização no aproveitamento dos recursos, minimizando desperdícios funcionais e temporais. Através do uso de conceitos de planejamento e configuração, este trabalho relata o desenvolvimento, assim como o resultado de experimentos realizados, os quais viabilizaram a validação da proposta de uma estratégia otimizada de implementação.

Palavras-chave: *Gerenciamento de Workflow, Motor de Inferência, Configuração, Planejamento, Sistemas de Workflow.*

ABSTRACT

Bad shared resource utilization causes difficulties in Hospital environments, when we think in a general manner. These resources may be human (professionals), material (machines) or physical (locations), and are often of limited availability, in spite of a high utilization demand. Besides this, another important feature in this kind of environment is the dynamic nature on its processes' execution, which requires an agile and flexible management, but, generally, is not correctly approached. However, these problems can be minimized by adopting some strategies. Reasonable resource utilization, together with an appropriated process management, appears as a possible solution. The practical appliance of this solution is concretized by The Cyclops Project, with the development of a Workflow Management System, which uses an intelligent Inference Engine and is destined to the correct health resource utilization. In this context, this work consists of an important part of the system, and its function is to promote good use of available resources, minimizing temporal and functional wasting. By means of the use of concepts of planning and configuration, this work presents the development and the results of carried out experiments, which made possible the validation of the proposal of an optimized strategy of implementation.

Keywords: *Workflow Management, Inference Engine, Configuration, Planning, Workflow Systems.*

1 INTRODUÇÃO

O ambiente hospitalar de um modo geral é, por natureza, um ambiente altamente dinâmico sob o ponto de vista dos seus processos do dia-a-dia. Visto que a saúde humana é um tema delicado, é grande a responsabilidade sobre todos os agentes que fazem parte da execução das atividades diárias de um hospital, pois um serviço eficiente deve ser obrigatoriamente prestado.

1.1 Contextualização do estudo

Gerenciar e controlar atividades de saúde não é uma tarefa simples, e até mesmo as grandes instituições desse ramo carecem da boa execução dessa prática. Essa dificuldade é visível em uma simples visita ao setor de clínica médica da maioria dos hospitais públicos no Brasil. Um estudo de caso realizado no Hospital Universitário da UFSC apresentado como proposta de projeto ao CNPq em 2004 pelo projeto Cyclops (CNPq, 2004), constatou essa estatística.

1.1.1 O controle de atividades em um ambiente hospitalar

Um paciente, ao ingressar no hospital, se depara com a execução de inúmeros processos. Em um simples tratamento de trauma, por exemplo, após dar entrada no pronto-socorro, o paciente, conforme diagnóstico inicial, é encaminhado ao setor de radiologia. Em seguida, passa por inúmeros exames

para, finalmente, ser lotado na enfermaria. Uma vez internado, o paciente é tratado e, eventualmente, liberado com alta.

Em cada um dos setores pelos quais o paciente é submetido, são preenchidos documentos e fichas das mais variadas naturezas, tais como cadastramento do paciente, prescrições médicas, observações clínicas, exames e laudos. A compilação de todos esses papéis resulta no prontuário do paciente, o qual é armazenado em grandes arquivos, fato este que dificulta sua localização e manuseio.

Tendo em vista a necessidade e o conseqüente interesse do Hospital Universitário da Universidade Federal de Santa Catarina em armazenar e gerenciar adequadamente as informações que compõem os processos hospitalares, tornou-se interessante a adoção de um sistema que gerencie as atividades de forma dinâmica.

Diante desse cenário, e considerando que quaisquer processos de atividades, sobretudo os de um ambiente hospitalar, requerem uma forma adequada de modelagem, uma boa representação e controle de processos é um ponto importante para a otimização da sua execução. A utilização de sistemas de gerenciamento de *Workflow*[1] apresenta, nesse contexto uma solução prática e viável.

---

[1] Termo em inglês que, em português, significa "Fluxo de trabalho"

1.1.2 Aplicação de Workflow no Ambiente hospitalar

Segundo a WfMC (*the Workflow Management Coalition*)[2], o termo *workflow* consiste na "facilitação ou automação de todo ou parte de um processo de negócios" (HOLLINGSWORTH, 1995). Com base nesse conceito, a filosofia de workflow propõe uma maneira de representar e manusear informações sobre as atividades dentro de um contexto de negócios de maneira simples, porém representativa, provendo, deste modo, uma maior capacidade de gerenciamento e reengenharia. Uma maneira adequada de se representarem os processos de uma instituição de saúde apresenta-se através de workflows dinâmicos, devido à potencial instabilidade de seus processos e recursos.

Em um exemplo, pode-se imaginar uma situação em que uma cirurgia está para ser realizada. Certamente, para a execução desta atividade, uma série de aparelhos e profissionais são previamente alocados, respeitando-se uma política de regras que leva em conta uma série de horários, logística, habilidades dos atores, características dos equipamentos, entre outras. Agora, suponha que, minutos antes do início do evento, ocorra um incidente qualquer, porém grave o suficiente para inviabilizar a realização da cirurgia naquelas circunstâncias. Com um sistema inteligente de gerenciamento de alocações em *workflows* dinâmicos, seria possível obter, em tempo hábil, uma nova configuração de todos os recursos mais propícios para a execução da mesma cirurgia, dada a nova situação.

---

[2] A WfMC é referência internacional em especificação e padronização de *workflow*

Essa alocação dinâmica e racionalizada de recursos é possível graças a um conjunto de elementos que viabilizam o processo de tomada de decisão específica ao problema.

1.1.3 Máquina de inferência e Tecnologia de *workflow*

As máquinas de inferência são pertinentes à vertente da Inteligência Artificial *simbólica*, e sua utilização é observada em sistemas especialistas[3]. Consistem em inferir, por meio de regras lógicas armazenadas em uma base de conhecimento, as soluções mais apropriadas dentre um conjunto de outras possíveis para um determinado problema ou situação (RUSSELL; NORVIG, 1995).

Esse conceito pode ser, por sua vez, inserido ao escopo do gerenciamento de *Workflow* dinâmico hospitalar, agindo, pois, como um motor de execução contínua, o qual permite o acompanhamento da execução das atividades. Esse motor de inferência vem a servir, também, de base estrutural para o desenvolvimento de um mecanismo capaz de realizar a configuração de atividades, isso é, fazer com que, dinamicamente, os recursos humanos (denominados atores) e materiais (denominados recursos propriamente ditos) sejam alocados dinamicamente, de maneira automatizada e eficiente.

1.1.4 O projeto Cyclops

O projeto Cyclops (*The Cyclops Project*) nasceu em 1992 como um projeto de longo prazo, tendo como fundadores o prof. Dr. Aldo von

[3] Sistemas inteligentes especializados em uma área específica, cujas atividades desempenhadas substituem, parcial ou totalmente, a função de um especialista humano (LUGER, 2004).

Wangenheim e o Prof. Dr. Michael M. Richter, na universidade de Kaiserslautern, Alemanha. Seu objetivo é desenvolver e colocar em prática, novos métodos, técnicas e ferramentas no campo de análises de imagens médicas usando técnicas advindas da Inteligência Artificial e da Visão Computacional.

Nesse contexto, cooperações com parceiros da medicina e da indústria alemã iniciaram-se em 1993. Em 1997, foi aceito como um novo projeto no quarto *Workshop of the German Brazilian Bilateral Program for Scientific and Technological Cooperation*, iniciando, então, suas atividades em 1998. Em meados de 2001, o projeto estendeu suas atividades para a área de gerenciamento de *workflow* e *PACS (Picture Archiving and Communication Systems)*.

Atualmente, o projeto desenvolve, além das áreas originalmente idealizadas, pesquisas direcionadas às tecnologias *Wireless* e *Wired* PACS, em conformidade com os padrões DICOM (*Digital Imaging and Communications in Medicine*) 3.0 (DICOM, 2004), além de aplicações de processamento de imagens direcionadas ao suporte de diagnósticos visuais e Tecnologias de Gerenciamento de workflow para unidades hospitalares.

1.2 Apresentação do problema

A distribuição dinâmica e racional de recursos concorrentemente compartilhados em atividades de processos hospitalares - que consiste em alocação, realocação, liberação e controle de agendas – caracteriza-se como uma tarefa complexa, que, para seres humanos, pode ser de solução inviável.

Sendo assim, o escopo do problema que o trabalho se propõe a resolver encontra-se exatamente na tarefa de dinamizar e racionalizar a utilização desses recursos, de modo a transcrever, da maneira mais natural possível, um modelo do seu funcionamento.

1.2.1 A Alocação Inteligente de Recursos

Em atividades ocorrentes no dia-a-dia de instituições cujo processo de trabalho é bastante dinâmico (em especial, ambientes hospitalares), é comum deparar-se com situações imprevistas, nas quais, mudanças repentinas acabam sendo necessárias.

Em se tratando de instituições públicas – onde a procura pelo atendimento é, em proporção, geralmente maior do que a quantidade de recursos disponíveis - o problema vai mais além. É necessário, pois, que os recursos disponíveis sejam aproveitados da melhor forma possível, de maneira que se possa ter a garantia de que, em determinado momento, os mesmos encontrem-se em sua máxima disponibilidade, dadas as eventuais necessidades.

1.2.2 Justificativas

O trabalho tem sua primeira justificativa fundamentada nas necessidades apresentadas pelo projeto Cyclops. O projeto *Medical Workflow Management* - subprojeto para o qual o presente trabalho é voltado - consiste na informatização com base na inovação, e empenha-se a desenvolver uma solução viável para o problema de gerenciamento de workflows dinâmicos. Para tanto, a contribuição

dada por este estudo ao projeto tem papel importante. O fato de se tratar de um sistema real, cuja implantação inicial se concretizará em um futuro próximo no hospital universitário da UFSC serve, também, como fator de motivação para o estudo.

A particular dificuldade apresentada nas instituições de saúde públicas no sentido de maximizar o uso de seus recursos configura-se como uma segunda justificativa. Espera-se que o gerenciamento racionalizado venha a acarretar saltos de qualidade baseados em uma maior agilidade no atendimento ao público, além de minimização dos custos em geral.

É interessante ressaltar que existem ferramentas aplicadas ao gerenciamento de w*orkflow* hospitalar, porém, poucas, ou talvez nenhuma provida de um núcleo capaz de dinamizar o processo de forma inteligente. Tal afirmação caracteriza uma terceira justificativa, pois apresenta a introdução de um novo conceito na área.

Sendo assim, os benefícios trazidos pelo projeto à sociedade são notáveis, servindo de ponto inicial para pesquisas futuras mais aprofundadas acerca do tema e proporcionando, uma vez posto em prática, resultados imediatos de melhoria em qualidade de serviço e atendimento na área da saúde.

1.3 Estrutura do trabalho

Nesta seção, é feita uma breve descrição da organização lógica deste trabalho. Sua estrutura é composta por cinco capítulos principais.

O capítulo introdutório contextualiza o estudo, apresentando o escopo do problema e justificando a solução adotada. O segundo capítulo comenta os objetivos a serem alcançados.

No terceiro capítulo, são contemplados os conceitos teóricos relevantes que fundamentaram o desenvolvimento. O quarto capítulo relata o desenvolvimento prático do trabalho, relatando, em detalhes, todas as etapas definidas nos objetivos.

For fim, no quinto capítulo, são apresentados os resultados, conclusões e propostas para discussão em produções futuras.

2 OBJETIVOS

Conforme visto no capítulo introdutório deste trabalho, a realidade do ambiente das instituições de saúde retrata uma carência importante no que se refere à quantidade de recursos disponíveis. Esses recursos - cuja natureza pode ser humana (médicos, enfermeiros, etc.) ou material (tomógrafos, salas de cirurgia, etc.) - caracterizam-se pela não possibilidade de compartilhamento e pela necessidade, todavia, de atendimento satisfatório à alta demanda de utilização de seus serviços.

2.1 Objetivo Geral

Os recursos disponíveis em ambientes clínico-hospitalares, apesar de escassos, podem ter sua utilização gerenciada através de ferramentas automatizadas, as quais proporcionam a obtenção de seu máximo aproveitamento.

Sendo assim, configura-se como objetivo principal da pesquisa, a extensão da máquina de inferência desenvolvida no Sistema *Cyclops Medical Workflow Management,* a fim de promover, de maneira automatizada e racionalizada, o gerenciamento das alocações de recursos entre as atividades de processos hospitalares. Essa idéia se materializa na forma de um mecanismo de configuração e controle de agendamentos.

2.2 Objetivos Específicos

O alcance do objetivo principal definido depende, na realidade, do alcance de alguns objetivos específicos, os quais retratam, em linhas gerais, o raciocínio traçado e seguido para a obtenção do êxito no desenvolvimento deste trabalho.

a) Criar um modelo computacional que represente, de maneira genérica, o cenário e a dinâmica de um processo de radiologia;

O ambiente hospitalar apresenta uma série de características peculiares, as quais devem ser refletidas no modelo de informações do Sistema, tanto estática quanto dinamicamente. A representação da estrutura informacional sob o aspecto estático exerce impacto direto na solução do problema, pois, é através dela que são representados os recursos a serem gerenciados. As características pertinentes aos recursos fundamentam critérios importantes de decisão no algoritmo de configuração de atividades. Por outro lado, a representação do aspecto dinâmico também desempenha papel importante. Ela permite a associação do conceito de *workflow* aos processos hospitalares, provendo a infra-estrutura necessária e viabilizando o desenvolvimento dentro de um escopo real de aplicação.

b) Definir e implementar mecanismos de controle de agendamentos;

Os recursos relevantes utilizados em unidades de saúde são, comumente, compartilhados de forma concorrente. Em outras palavras, de um modo geral, um profissional ou aparelho não é fisicamente capaz de realizar múltiplas tarefas ou estar em vários locais em um mesmo momento. Em contrapartida, a utilização racionalizada de recursos demanda disponibilidade em tempo hábil. Diante deste problema, é imprescindível a intervenção de um agente capaz de gerenciar agendamentos de recursos, apontando colisões temporais e indicando horários de ociosidade.

c) Aplicar critérios para alocação de recursos ao algoritmo de configuração de *workflows*;

Uma característica do mau gerenciamento da utilização de recursos concorrentemente compartilhados em um processo é o desperdício funcional. Considerando os recursos como entidades dotadas de funcionalidades que variam em número, pode-se afirmar que esse desperdício localiza-se, em parte, na falta de critérios de escolha em momentos onde há múltiplos recursos disponíveis. À medida que se racionaliza esse processo de escolha, maximiza-se a disponibilidade dos recursos em momentos posteriores, pois, dessa forma,

são poupados aqueles recursos com características excedentes quando há disponibilidade de outro mais restritamente especializado.

d) Desenvolver módulos para visualização, validação e aquisição de resultados.

A real otimização na utilização dos recursos através do desenvolvimento de todo o trabalho necessita ser comprovada por meio de algum método confiável. A fim de chegar a conclusões (esperadas e não esperadas) baseadas nos resultados obtidos, faz-se necessário o desenvolvimento de módulos que permitam a validação e a avaliação da efetividade do modelo proposto. Além disso, tal conjunto de módulos apresenta essencial importância para a continuidade do projeto *Cyclops Medical Workflow Management,* pois, baseando-se nos resultados aqui obtidos, é traçado o futuro rumo da pesquisa como um todo.

3 FUNDAMENTAÇÃO TEÓRICA

Embora envolva certo grau de novidade quanto à arquitetura e contexto de aplicação, o presente trabalho está sendo desenvolvido com fundamento em conceitos teóricos da literatura clássica da Inteligência Artificial e tecnologia de Workflow.

Este capítulo tem como objetivo relatar um levantamento desses conceitos, a fim de mostrar sua importância para o embasamento científico do trabalho proposto e, também, para iniciativas de trabalhos futuros.

3.1 Planejamento

Embora planejamento (em inglês, *planning*) e sistemas de Gerenciamento de *workflow* (*workflow management systems*) envolvam idéias e propósitos distintos, a junção de conceitos das duas partes pode gerar um resultado positivo. Esta seção apresenta algumas definições acerca de planejamento, com o intuito de mostrar a relação de conceitos existentes entre esta técnica e o sistema de gerenciamento de workflow médico desenvolvido neste trabalho.

3.1.1 Introdução

Resolver um problema tal como ele é no mundo real, consiste na maioria das vezes em uma tarefa inviável de ser resolvida por uma máquina, observado o infinito número de estados, variáveis e regras que compõem todo o universo

que nos cerca. Contudo, à medida que esse problema é transformado em algo específico, a complexidade cai consideravelmente, a ponto de tornar viável uma solução computacional (RUSSEL; NORVIG, 1995).

O planejamento entra nesse contexto como uma ferramenta de representação do conhecimento, com o intuito de simplificar e restringir a um universo específico o escopo de determinado problema, além de indicar uma solução otimizada para tal.

Levando-se em consideração que a solução computacional inteligente, na grande maioria das situações reais, consiste em uma tarefa não-trivial, o emprego de técnicas de planejamento é viável e amplamente utilizado na resolução de problemas específicos.

3.1.2 Definição

Segundo Doyle (1998), a resolução de um problema consiste no processo de se desenvolver uma seqüência de ações para que seja alcançada uma meta. Contudo, para que essa meta seja alcançada da melhor maneira, essas ações devem ser coordenadas por meio de um planejamento.

O autor diz, ainda, que planejamento consiste em decidir sobre o curso de uma ação antes de agir, sendo que um plano é definido pela representação desse curso, e consiste em uma seqüência de operadores, os quais levam ao alcance das metas propostas.

3.1.3 Características

Planejamento pode ser caracterizado como uma técnica que propicia a criação de táticas para a resolução de problemas que vislumbra momentos futuros ao estado inicial, com o objetivo de verificar a existência de ordens razoáveis de seqüências de ações (operadores) a fim de alcançar uma meta. Há inúmeros benefícios no emprego de técnicas de planejamento. Três deles podem ser destacados: a) redução no tempo de processamento da execução efetiva da tarefa; b) resolução de conflitos entre ações; e c) possibilidade de recuperação de erros por meio de *backtracking* (RUSSELL; NORVIG, 1995).

No entanto, a escolha das ações (operadores) que serão executadas depende de alguns fatores relativos ao estado corrente do universo do problema. Em outras palavras, para que seja definida a melhor seqüência de ações possível, é necessário que sejam avaliadas todas as características de cada uma delas para que, de acordo com a necessidade, sejam descartadas aquelas irrelevantes, assim como selecionada a mais apropriada. Essa escolha parametrizada do elemento apropriado dentre um conjunto de vários possíveis é desempenhada através de um mecanismo de configuração.

As técnicas de configuração caracterizam-se por buscas sucessivas a uma base de conhecimento extensa, porém finita e bem definida acerca de cada elemento constante no universo do problema (RICHTER, 2001). Cada um desses elementos possui um conjunto de atributos que visa atender ou não a um conjunto de requisitos. Com base em heurísticas (regras lógicas pré-determinadas), é eleito, então, o elemento que melhor atende à necessidade em

dada situação. A lógica genérica do processo de configuração pode ser entendida como um algoritmo recursivo, onde sua parada ocorre no momento onde, no estado final, todos os requisitos são satisfeitos (Figura 1).

Figura 1 – Idéia do Algoritmo de Configuração (RICHTER, 2001)

3.1.4 Classificação

O termo planejamento é um tanto quanto antigo. A evolução natural do conceito ao longo das décadas propiciou o surgimento de inúmeras técnicas e ferramentas, que foram agregando novos atributos aos sistemas de planejamento.

Tais sistemas podem ser divididos em duas classes principais: os sistemas de planejamento hierárquico e os sistemas de planejamento não-hierárquico. Paralelamente a essas duas classes, os sistemas de planejamento podem, ainda, ser classificados quanto à linearidade da execução em mais duas grandes classes: lineares e não-lineares (DOYLE, 1998).

3.1.4.1 Planejamento Hierárquico

Os sistemas de planejamento hierárquico (também conhecidos por *hierarquical planning system*s) propiciam a geração de planos que permitem o alcance das metas em níveis crescentes de refinamento, isto é, de maneira hierárquica. No planejamento hierárquico, o plano resultante tem a estrutura de uma árvore, onde cada meta pode conter outras metas mais específicas, que resolvem um fragmento menor e mais especializado do problema. (RUSSELL; NORVIG, 1995).

Por exemplo, o plano {Ir(Mercado), Comprar(Leite), Comprar(Pão), Ir(Casa)} poderia ser decomposto em tarefas mais específicas. A ação Ir(Mercado) poderia derivar as ações {Sair(Casa), Dirigir(Carro), Entrar(Mercado), Estacionar(Carro)}. Do mesmo modo, a ação Dirigir(Carro) poderia ainda ser decomposta em mais uma série de sub-ações, tais como {Abrir(PortaDoCarro), Entrar(Carro), Ligar(Carro), Arrancar(Carro), (...)} até que o detalhamento seja suficiente para representar uma solução coerente.

Nesse tipo de sistema, é possível a abstração de tarefas que são irrelevantes em um dado momento. O objetivo do planejamento hierárquico é prover uma visão abstrata e macroscópica, simplificando o processo de busca e raciocínio ao representar tarefas mais vagas, ao invés de percorrer cada uma das metas específicas a serem alcançadas no problema.

3.1.4.2 Planejamento Não-hierárquico

Nos sistemas de planejamento não-hierárquico (n*on-hierarchical Planning*), não há distinção entre metas de alto nível e metas específicas, ou seja, não há o conceito de derivação (especialização) de tarefas. Assim, todas as metas a alcançar são as mais específicas possíveis. Esse tipo de sistema é pouco usado atualmente, visto que há uma série de desvantagens no fato de não haver essa relação de hierarquia e de não ser possível a abstração de partes quando conveniente.

3.1.4.3 Planejamento Linear

No contexto do planejamento, linearidade significa a não-possibilidade de existirem múltiplos caminhos de execução das tarefas. Conseqüentemente, as tarefas contidas em planos lineares têm, em sua execução, uma forte relação de sincronia temporal, visto que duas ou mais tarefas não podem de maneira alguma ser executadas simultaneamente.

Em planejadores deste tipo, as metas são alcançadas por meio de uma única seqüência de processamento. Essa seqüência é caracterizada pela ausência de bifurcações que representem, além da execução paralela de tarefas, algum tipo de desvio baseado em critérios de decisão (NILSSON; FIKES, 1970).

3.1.4.4 Planejamento Não-linear

Ao contrário dos sistemas de planejamento linear, os sistemas de planejamento não-linear permitem que a execução flua percorrendo mais de um

caminho, de maneira simultânea ou não. Nos planos resultantes desses sistemas, existem momentos em que não há dependência temporal entre as tarefas. Isto é vantajoso, pois permite a execução de maneira assíncrona, o que possibilita uma representação mais ampla e realista dos processos que ocorrem no mundo real.

3.2 Sistemas de Gerenciamento de Workflow

O objetivo desta seção é definir o termo sistema de *workflow* de acordo com os padrões publicados *no Workflow Reference Model* (HOLLINGSWORTH, 1995), a fim de demonstrar a contextualização da estrutura do sistema desenvolvido em um padrão internacional.

3.2.1 Introdução

O crescente investimento no estudo e pesquisa na tecnologia de *workflow* retrata a necessidade de redução do custo na forma como os negócios são executados e de mais agilidade no desenvolvimento de novos serviços e produtos.

Essas necessidades são supridas pela tecnologia de *workflow* através de Metodologias e ferramentas que dão suporte à modelagem, abstração e execução dos processos de uma organização. Sua principal característica é a combinação das atividades humanas com as atividades da máquina, através de ferramentas que propiciam suporte à execução das atividades.

3.2.2 Definição

Segundo Hollingsworth (1995), entende-se por *workflow* “a facilitação ou automatização computadorizada de um processo de negócio, como um todo ou em parte”, o qual troca documentos, informações, e atividades de trabalhos de diferentes participantes, ou seja, é a execução coordenada de múltiplas tarefas (e.g. e-mail, um formulário), afim de que sejam tomadas ações, de acordo com um conjunto de regras e procedimentos.

SUNY (1997) define workflow como algo cujo objetivo é aumentar a eficiência de processos de negócio, tanto os críticos quanto os eventuais, e a efetividade das pessoas que trabalham em conjunto para executá-los.

Analisando-se essas definições de diferentes autores, é possível perceber que há uma convergência de opiniões acerca do tema, no que diz respeito ao fato de *workflow* consistir em uma seqüência de ações, as quais levam à facilitação e otimização de um processo. Percebe-se, conseqüentemente, que workflow está fortemente associado ao conceito de plano, cuja definição é mostrada na seção “Planejamento (*Planning*)”. Na verdade, workflow nada mais é do que um plano resultante de um processamento prévio, realizado por um especialista humano ou por um sistema de planejamento computadorizado.

Por sua vez, observando-se os conceitos de workflow apresentados, um *Workflow Management System* (WfMS) é um sistema que define, cria e gerencia a execução de *workflow* através do uso de software, o qual é capaz de interpretar a definição do processo, interagindo com os participantes do *workflow*

e, onde requerido, solicitando o uso de ferramentas de tecnologia de informação e suas aplicações.

3.2.3 Características

Um processo de negócios pode ter um ciclo de vida que varia de minutos até meses, podendo ser implementado de várias maneiras, usando uma variada infra-estrutura de TI e de comunicações. Além disso, podem operar em um ambiente que varia de uma pequena rede local na empresa até longas distâncias, internacionalmente. Contudo o Modelo de Referência da WfMC, com o intuito de acomodar de forma compatível todo o universo de técnicas de implementação e ambientes de aplicação, definiu um conjunto de características, que fornecem a base arquitetônica para o desenvolvimento de sistemas dessa tecnologia.

Basicamente, todos os sistemas de *workflow* devem prover suporte em três áreas principais: as funções de tempo de construção (*build-time functions*), responsáveis por definir e modelar o processo dos *workflows*; as funções de controle em tempo de execução (*run-time control functions*), que gerenciam e executam todo o processo em um ambiente operacional; e finalmente, as interações com os usuários e aplicações auxiliares. A Figura 2, extraída do modelo de referência da WfMC, ilustra essa estrutura básica:

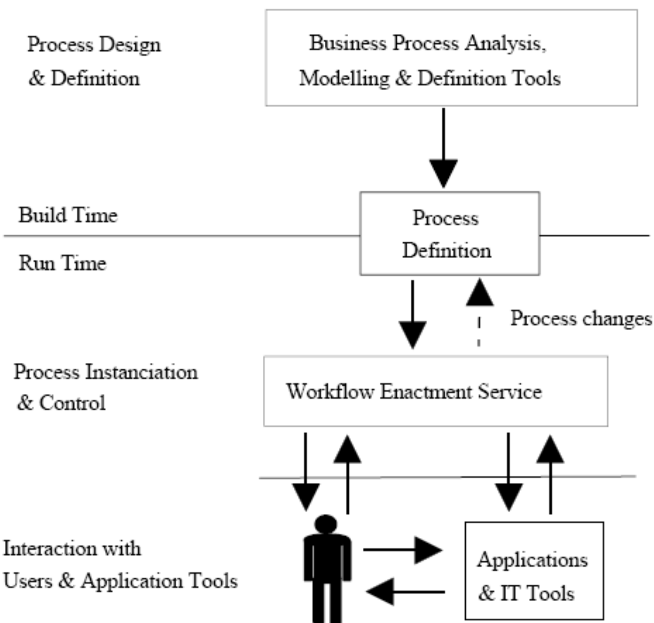

Figura 2 - Características de um Sistema de *workflow* (HOLLINGSWORTH, 2005)

3.2.3.1 Funções de Tempo de Construção (*build-time functions*)

Esta parte preocupa-se com a definição e modelagem do processo de *workflow* e suas atividades. Os processos são traduzidos do mundo real para um modo formal e não-ambíguo, o qual seja compreensível por uma máquina e compatível com o restante do sistema. As definições de processos (*process definitions*) resultantes destas funções podem estar representadas de maneira gráfica, textual ou na forma de linguagens formais.

3.2.3.2 Funções de Controle de Processo em Tempo de Execução *(run-time control functions*)

São responsáveis por interpretar as definições de processo durante o tempo de execução através da ligação dos processos à execução do mundo

real. A principal parte é o *workflow engine* (motor de *workflow*) o qual é o responsável pela criação, distribuição, remoção e execução de atividades. Preocupam-se com o gerenciamento de processos de *workflow* em um ambiente operacional e com o seqüenciamento das atividades para serem manuseadas como parte de cada processo.

3.2.3.3 Interação de Atividades em Tempo de Execução (*run-time activity interactions*)

Esta parte tem o controle entre as diferentes atividades e seus participantes, ou seja, interage em tempo de execução com os usuários e ferramentas para processamento dos vários passos das atividades. Geralmente, essas funções agem em conjunto com ferramentas de TI (tais como ferramentas para preenchimento de formulários, por exemplo), e são necessárias para que os usuários possam acompanhar o processo, bem como intervir na execução, a fim de alimentar o sistema com novos dados, etc.

3.2.4 Classificação

Com o passar do tempo, a evolução os Sistemas de Gerenciamento de *Workflow* fez com que sua aplicação se expandisse às mais diversas áreas. Com isso, uma série de classificações acerca desses sistemas veio à tona, podendo ser aplicadas sob diversos aspectos, tais como a tecnologia e estrutura e complexidade dos processos.

Quanto à tecnologia e estrutura, destacam-se os sistemas centrados em e-mails, onde é predominante o uso de ferramentas *standalone*, que direcionam documentos aos participantes por meio de sistemas de e-mail; os centrados no gerenciamento de documentos, os quais se caracterizam por serem utilizados por meios de *software* de prateleira; e os centrados em atividades ou processos, onde o objetivo é dinamizar as tarefas, reduzindo gastos através da ação integrada com outros sistemas.

Quanto à complexidade dos processos, sistemas administrativos, *Ad Hoc*, Colaborativos e de Produção podem ser citados.

3.2.5 Algumas soluções existentes

Esta seção visa apresentar alguns exemplos de aplicações práticas existentes, a fim de proporcionar critérios de comparação entre as mesmas e o Sistema proposto pelo projeto Cyclops.

3.2.5.1 TriGSflow

Trata-se da tese de PhD de Stefan Rausch-Schott, e consiste em um sistema para gerenciamento de *workflow* que oferece flexibilidade no sentido de suportar regularmente mudanças de solicitações e condições. O sistema é baseado em um banco de dados orientado a objetos, em regras de Evento-Condição-Ação (ECA) integradas, e em papéis que auxiliam a evolução do objeto.

O autor aborda workflow management system sob o aspecto transacional, empregando um modelo genérico de workflow centrado em atividades,

caracterizado por vários gêneros de *workflow:* pastas, itens históricos, e itens de trabalho.

A solução engloba, também, aspectos administrativos e de comunicação, o que permite uma representação expansível da estrutura organizacional aliada à comunicação assíncrona de informações entre os agentes participantes.

3.2.5.2 Vortex

Propõe um paradigma de programação para a especificação de atividades de tomada de decisão, aplicadas a *workflow*. Nesse sistema, *workflows* são modelados declarativamente. Direciona o foco no conhecimento de como os objetos de entrada são processados dentro de uma organização e adota a arquitetura de caixa-preta, onde os módulos computam valores de atributo.

Utilizando esse sistema, cabe ao programador a especificação das condições sob tarefas a serem executadas.Essa abordagem é útil em *workflows* onde não se sabe precisamente o número exato de passos a serem executados.

4 CYCLOPS MEDICAL WORKFLOW MANAGEMENT

Como mencionado anteriormente, o motor de inferência e, conseqüentemente, a parte referente ao presente trabalho, constitui parte importante de um sistema mais abrangente, composto de vários subsistemas que funcionam de forma integrada, com base em padrões internacionais que garantem a compatibilidade na comunicação de informações.

4.1 Visão Geral do sistema

O *Medical Workflow Management System* adota a arquitetura cliente-servidor, onde vários clientes fazem requisições via rede (sobre o protocolo TCP/IP) a um servidor central, o qual fornece serviços de resposta.

A estrutura é composta por três sistemas principais: um modelador de Processos (Workflow Designer), que permite a abstração gráfica dos *workflows*; o servidor de *workflow* (Workflow Server), que é responsável por todo o processamento de controle e execução do sistema; e um Cliente para acesso (Workflow Client), cuja finalidade é fornecer serviços aos usuários finais, permitindo-lhes a operação mais amigável do sistema.

4.1.1 O modelador de Processos (Workflow Designer)

Essa ferramenta corresponde às funções desempenhadas no tempo de construção dos workflows a serem posteriormente alocados e gerenciados. Sua

interface gráfica permite que o especialista defina os processos, bem como seu fluxo de execução, arrastando ícones e interligando-os com setas.

A ferramenta inclui o conceito de *Macros*, que nada mais são do que *workflows* dentro de *workflows*. A vantagem do uso de *macros* é a geração de planos hierárquicos, o que possibilita a abstração de complexidade e uma melhor visão do processo em níveis mais altos (Figura 3).

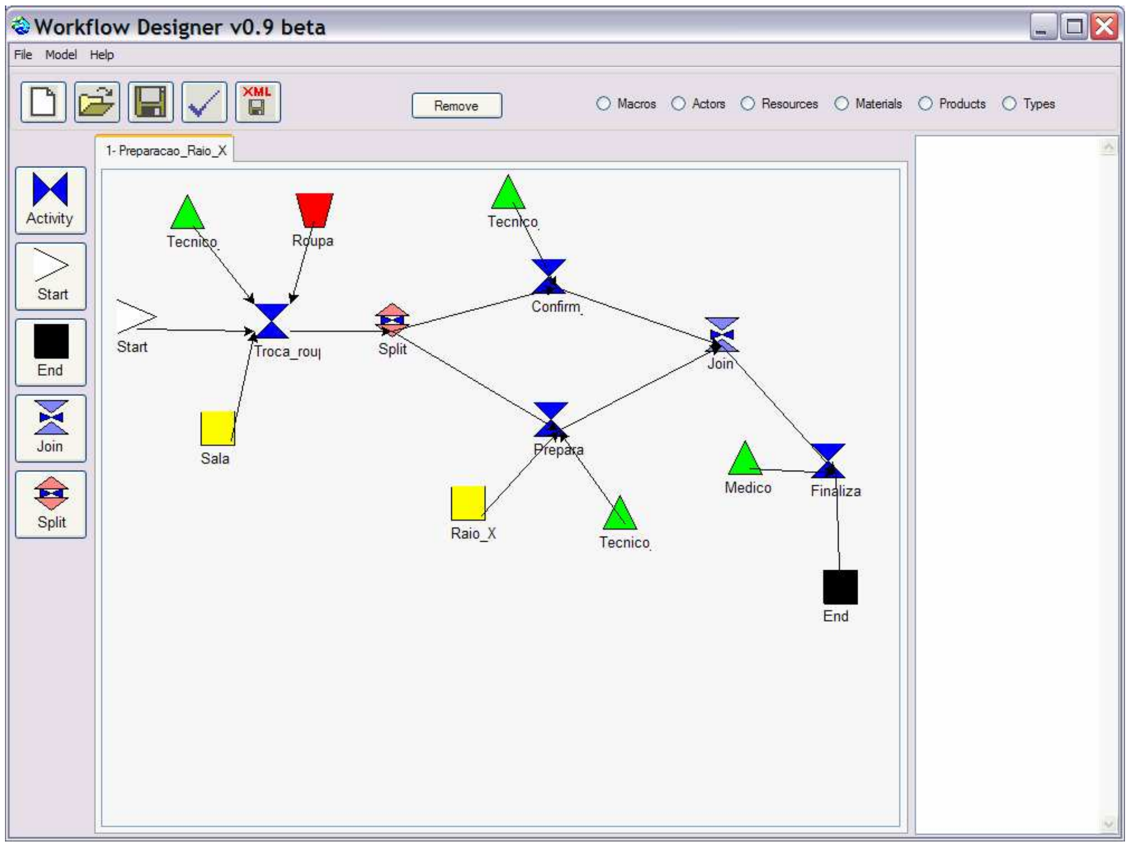

Figura 3 - Tela principal do *Cyclops Workflow Designer*

Como se pode perceber através da análise da Figura 3, cada categoria de informação relevante ao modelo de workflow corresponde a um ícone específico. Cada *workflow* modelado através dessa ferramenta pode ser exportado no

formato xml, a fim de ser compatível com o restante das aplicações do sistema e, particularmente, com o motor de inferência.

4.1.2 O Cliente (Workflow Client)

Consiste em uma ferramenta através da qual os usuários têm acesso ao sistema. Os *Workflow Clients* constituem aplicações terminais, distribuídas entre os setores administrativos, e são responsáveis por solicitar serviços ao Servidor central.

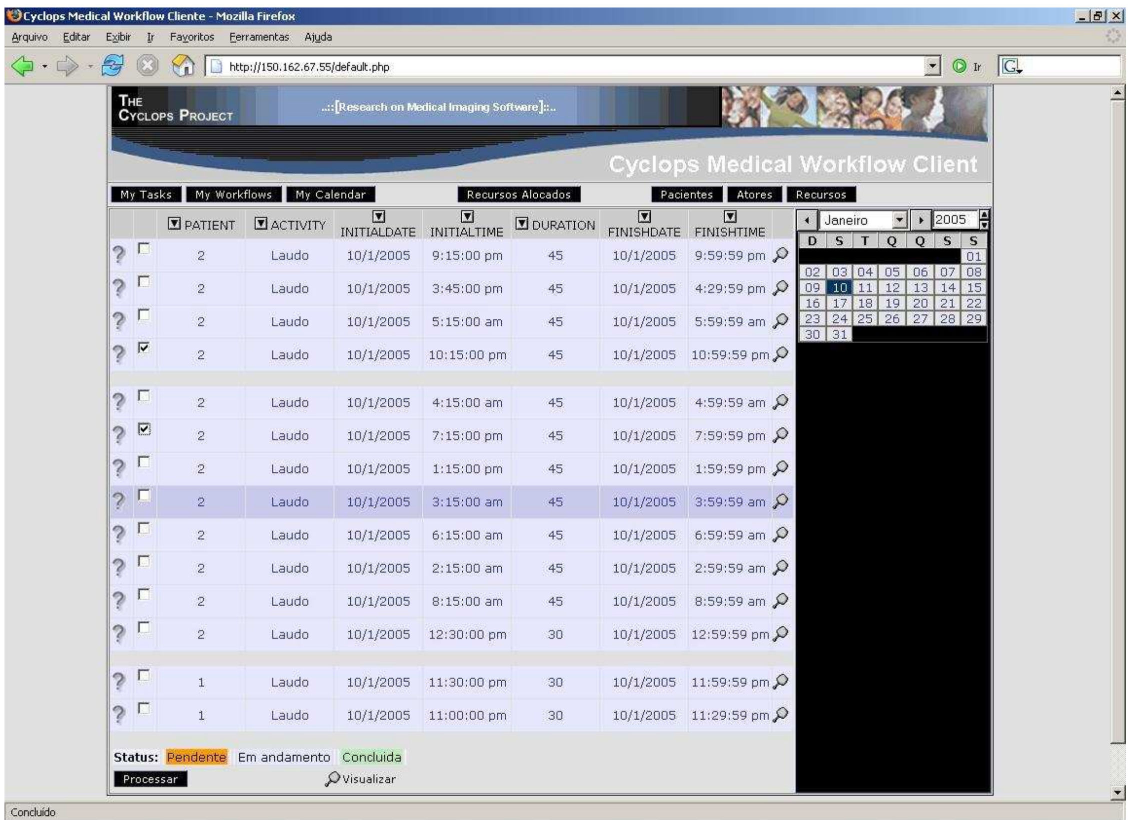

Figura 4 - Tela principal do Cyclops Workflow Client

O módulo deve atuar, também, como ferramenta gráfica, permitindo que os usuários humanos possam acompanhar e gerenciar o andamento dos

processos. Esse acompanhamento se dá graças às funções de visualização de processos, recursos e agendas (Figura 4).

4.1.3 O Servidor (Workflow Engine)

Este componente fornece as funções de tempo de execução (*Run-time control functions*), constituindo o centro de controle de todo o sistema. A modelagem da arquitetura de informações do servidor foi inspirada na documentação do padrão DICOM 3.0 (DICOM, 2004), que define a estrutura de um servidor de processamento e de como devem funcionar a comunicação, as associações e os serviços aos clientes. O padrão DICOM apóia-se em padronizações estabelecidas pelos organismos internacionais ISO, ANSI, IEEE, dentre outros.

A estrutura do modelo pode ser entendida como uma pilha de quatro camadas, cujo fluxo de dados atua de baixo para cima. A primeira camada do modelo consiste em um canal de comunicação, que nada mais é do que um protocolo de camada de aplicação, atuando sobre o protocolo TCP/IP. Logo acima, estão as camadas de apresentação de dados e serviços de associações. A camada do elemento de serviços de mensagem vem logo a seguir, fornecendo serviços diretamente para a camada mais interna: a camada da máquina de inferência do modelo, que se localiza no topo da pilha. O modelo de camadas, que compõe o servidor é representado pela Figura 5.

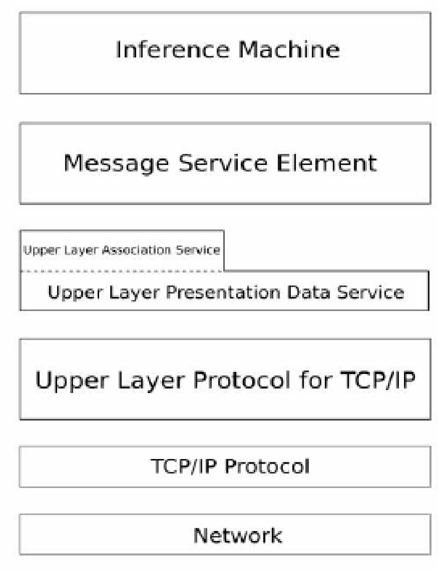

Figura 5 - Estrutura em camadas do Servidor de Workflow (RIBEIRO, 2005)

A camada *Upper Layer Protocol for TCP/IP* atua logo acima da camada de rede TCP/IP, sendo a camada mais externa e de mais baixo nível do sistema. É responsável por fazer contato com o sistema Operacional, a fim de viabilizar, através de *Socket*, a comunicação entre os clientes e o servidor.

Internamente, a camada é dividida em dois componentes: *SocketProvider*, que permite que os clientes conectem-se e enviem suas mensagens ao servidor; e *SocketUser*, que fornece serviços que permitem que o servidor envie aos clientes as respostas das requisições.

A subcamada de *Upper Layer Association Service* provê serviços de associação, ou seja, reserva no servidor uma porta de comunicação individual para cada cliente conectado. Além disso, ela é o elo entre o cliente e as camadas mais internas do sistema. A camada *Upper Layer Presentation Data*

Service é responsável pelo gerenciamento das associações, validando ou não as requisições através de verificações.

Logo acima, na camada denominada *Message Service Element*, as requisições são identificadas e direcionadas aos objetos incumbidos de resolvê-las. Esse reconhecimento se dá devido ao emprego de um protocolo denominado Service-Object Pair (SOP), onde consta o serviço a ser executado e o objeto associado a ele.

Por fim, o topo da estrutura é constituído pelo motor (ou máquina) de inferência. Este é o componente que dispara os módulos de configuração desenvolvidos no trabalho, sendo o centro principal de processamento de todo o sistema. O funcionamento global da máquina de inferência, bem como o relato de todo o desenvolvimento do mecanismo de gerenciamento de alocações é descrito nas seções subseqüentes desta monografia.

4.2 O Motor de Inferência

4.2.1 Visão geral

O motor ou máquina de Inferência (*Inference Engine* ou, ainda, *Inference Machine*) constitui o coração do sistema de workflow em desenvolvimento. É o núcleo do processamento, para onde as informações contidas nas requisições dos clientes são destinadas e eventualmente tratadas, a fim de produzir o devido resultado.

A implementação do seu módulo principal foi realizada sob um algoritmo relativamente simples, baseado no conceito de metas reduzidas por operadores,

onde as metas (*goals*) são os objetivos a serem alcançados (tarefas a serem concluídas) e os seus respectivos operadores (*operators*) representam a maneira como essas metas serão alcançadas.

Uma meta deriva um conjunto (vazio ou não) de outras metas. Essa relação é recursiva, isto é, as metas derivadas de uma meta podem derivar outras metas e assim sucessivamente. Sendo assim, uma meta é alcançada (reduzida) quando, por meio de um operador, sua execução é concluída e não há mais metas-filhas a reduzir.

Observando-se o procedimento acima, é possível visualizar um conceito aplicado ao mecanismo básico de inferência de sistemas clássicos de Planejamento Hierárquico (*Hierarquical Planning*). Embora realize inferência direcionada a outro fim (sistemas de planejamento utilizam inferência para tomada de decisões, baseando-se em regras), a máquina de inferência proposta utiliza o mesmo MÉTODO de execução, o qual se caracteriza pela execução hierárquica de ações através de metas/operadores.

4.2.1.1 Estrutura Básica

O motor mantém um repositório, onde são armazenadas todas as metas, assim como seus respectivos operadores de redução. Além disso, nela, são mantidas duas filas principais: a fila de metas a executar (*executionTo*) e a fila de metas em execução (*executionIn*). A fila de metas *a executar* é, na verdade, um vetor de filas, cujos índices correspondem a níveis de prioridade. Já a fila de metas *em execução* consiste em uma fila em seu conceito “clássico”, ou seja, o único critério de saída é a ordem de chegada.

4.2.1.2 Funcionamento Básico

Conforme foi comentado anteriormente, cada meta é programada na máquina de inferência contendo um nível de prioridade. Esse nível de prioridade pode variar de 1 a 100 sendo que, para cada um desses níveis, é mantida uma fila específica. Quando uma meta é requisitada à máquina de inferência, ela é inserida na fila correspondente ao seu nível de prioridade, para que seja posteriormente reduzida. Além de possuir nível de prioridade, uma meta pode, também, conter dependências (metas prioritárias), de modo que a mesma só será escalonada para execução após todas as suas dependências serem resolvidas.

A inserção de uma meta na fila de metas *a executar* indica que a mesma está pronta para a execução, porém, sob dependência da sua respectiva prioridade para ser ativada. Quando uma meta é inserida na fila de metas *em execução*, isto significa o início da sua redução. Tal meta permanece nessa fila até que o processo de redução seja totalmente concluído. Na prática, sob a ótica da fila de execução, a redução de uma meta se dá pela substituição da mesma pelas suas metas-filhas.

Quando em atividade, a máquina de inferência mantém um *looping* constante, cuja execução ocorre em paralelo ao restante do processamento do sistema, e que é responsável por varrer a fila de metas *a executar* a fim de realizar o escalonamento. O escalonamento leva em consideração a prioridade, as dependências e a ordem de chegada. Esse laço de repetição contínua pode

ser mais claramente entendido observando-se o pseudocódigo da Figura 6, o qual resume o algoritmo de forma simplificada:

```
[Método execute]

Enquanto maquinaAtiva faça:
       melhorPrioridade := encontrePrioridadeParaExecutar;
       se encontrouPrioridade então:
             melhorMeta := encontreMetaNaPrioridade(melhorPrioridade);

             se encontrouMelhorMeta então:
                   reduzaMeta(melhorMeta);
             fim-se;
       fim-se;
fim-faça;
```

Figura 6 - Algoritmo contínuo da máquina de inferência

Como se pode perceber, a primeira ação executada no algoritmo é encontrar a prioridade mais adequada. Para isto, é realizado um cálculo numérico simples, que leva em conta o valor (número inteiro de 1 a 100) e o tamanho (número de metas contidas) do vetor correspondente a cada prioridade (Figura 7).

```
[Método encontrePrioridadeParaExecutar]

para cada elemento de listaDePrioridades faça:
      valorCalculado := priordade.numMetas * prioridade.valor / 100;
      listaValoresCalculados.adicione(valorCalculado);
fim-faça;
retorne listaValoresCalculados.maiorValor;
```

Figura 7 - Algoritmo de escalonamento de prioridades

Observando o quadro, percebe-se que a “melhor prioridade” corresponde àquela cuja relação do valor com o número de metas respectivas resulta em um maior número. Desta maneira, quando necessário, metas com prioridades

menores podem ser reduzidas em detrimento de metas mais prioritárias, o que minimiza o problema de gargalos.

O passo final para definir a preferência de execução em cada iteração do algoritmo leva em conta o número de dependências que, por ventura, as metas possam ter de outras metas. Dentre as metas contidas na prioridade selecionada, é escalonada a primeira cujas dependências são inexistentes, ou já foram totalmente resolvidas.

4.2.2 Metas e operadores: Implementação

No sistema proposto, as metas e operadores constituem um conjunto determinado em tempo de desenvolvimento. Eles são empregados em diversas funcionalidades, que vão desde o carregamento dos *workflows* criados na ferramenta de modelagem, até serviços de alocação, realocação, execução, ativação e desativação da máquina de inferência.

Cada meta corresponde a um objeto de uma classe que estende "*Goal*". *Goal* é uma classe abstrata que oferece atributos e métodos que permitem que suas subclasses conheçam seus respectivos operadores de redução, efetuem tal processo (método *reduce()*), comuniquem-se com a máquina de inferência, gerem metas-filhas, conheçam suas dependências, entre outras características. O resultado da redução de uma meta depende do estado do sistema e dos parâmetros de inicialização passados no momento de sua criação.

Paralelamente às metas, cada operador de redução apresenta-se implementado no sistema como instância de uma subclasse da classe abstrata denominada "*Operator*". Os objetos das subclasses de *Operator* são os

elementos capazes de - através da execução de um método denominado *eval()*, que é chamado pelo objeto-meta através do método *reduce()* - reduzir de fato sua respectiva meta, bem como executar as ações necessárias para que se produza o resultado. Além disso, são os operadores os responsáveis pela validação dos parâmetros de inicialização de suas metas associadas (Figura 8).

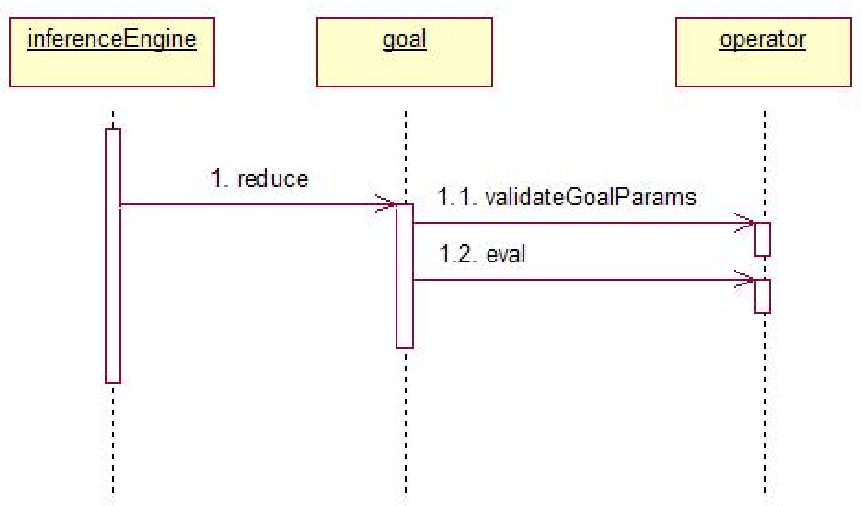

Figura 8 - Seqüência de execução da redução de uma meta

4.3 Aspectos do desenvolvimento do trabalho

4.3.1 Metodologia adotada

O intuito desta seção é descrever e justificar as técnicas e ferramentas utilizadas em toda a extensão do trabalho, justificando sua necessidade e relevância.

4.3.1.1 Ferramentas e Modelo de Desenvolvimento Utilizados

A linha de raciocínio seguida pelo projeto *Cyclops,* por envolver, em sua maioria, projetos de pesquisa, demanda a adoção de um modelo de desenvolvimento incremental, com base na criação de protótipos. Os sistemas iniciais desenvolvidos adotam a linguagem Smalltalk como ferramenta para prototipação. O uso desta linguagem se deve ao fato de a mesma ser de relativamente fácil aprendizado, além de ser oferecer facilidades relacionadas à realização de testes de unidade (ABDALA; WANGENHEIM, 2002).

O desenvolvimento do presente seguiu o raciocínio do projeto, constituindo parte do primeiro protótipo funcional do sistema e tendo *Smalltalk* como linguagem para desenvolvimento, através do ambiente Cincom VisualWorks não-comercial, em sua versão 7.2.

4.3.2 Definição do Modelo de classes de *workflow*

Para a criação do modelo de informações de representação dos workflows mantidos pelo Servidor, usaram-se, como base, as mesmas entidades abstraídas no modelador gráfico (Tabela 1), salva alguma diferença situada na existência de algumas classes auxiliares.

Tabela 1 - Entidades que compõem *workflows* no sistema

Start	- Entidade única em um dado fluxo, que sinaliza onde é o seu início.
Activity	- Entidade principal em um fluxo, que representa uma tarefa a ser executada.
Join	- Entidade de junção de fluxos concorrentes.
Split	- Entidade de separação de um fluxo em vários.
End	- Entidade que indica o fim de um processo.
Actor	- Entidade que representa recursos humanos capazes de participar da realização das tarefas às quais estão associadas. Ex.: Médico.
Resource	- Entidade que representa recursos materiais compartilhados envolvidos da realização das tarefas às quais estão associadas. Ex.: Sala, tomógrafo.
Material	- Entidade que representa recursos materiais que não são compartilhados por várias atividades. Parte-se do princípio que estes estejam sempre disponíveis. Ex.: Bisturis, luvas, seringas.

Como o servidor não dispunha de um módulo de *parsing*, capaz de converter o xml gerado em seu modelo local de objetos, fez-se necessário, para o início das atividades do trabalho, o desenvolvimento desse módulo. O diagrama de classes, representado pela figura, apresenta o modelo conceitual das entidades básicas que compõem um *workflow*. Note a existência de uma classe denominada *WorkflowLoader*, associada à classe *Workflow*. É ela a responsável pela leitura, *parsing* e carregamento do xml para o modelo de objetos proposto.

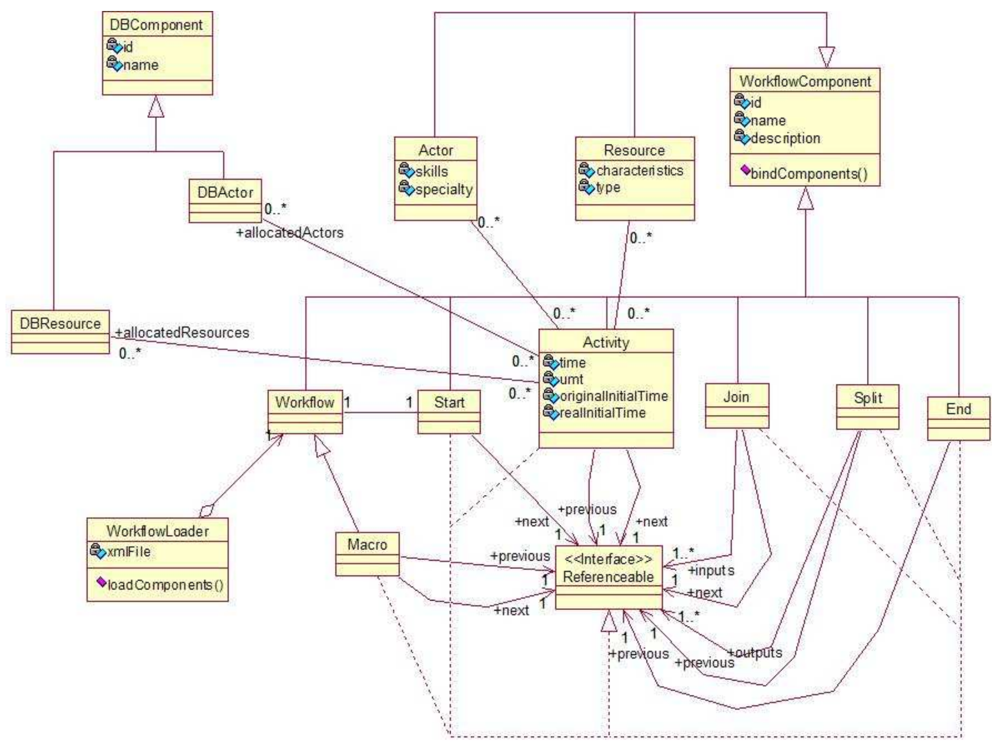

Figura 9 - Diagrama de classes traduzindo o modelo de informações

O diagrama de classes ilustrado na Figura 9 permite a visualização de duas classes que, à primeira vista, aparentam acusar ambigüidades no modelo. Tratam-se das classes *DBActor* e *DBResource*. Entretanto, há uma diferença conceitual significativa entre essas classes e as classes *Actor* e *Resource*. As

classes *Actor* e *Resource* configuram-se como subclasses de *WorkflowComponent*. Na verdade, elas representam apenas conceitos abstratos relacionados a especificações de perfis de atores em workflows ainda não instanciados - os denominados *workflows abstratos* (no sistema de *workflow* médico proposto, a instanciação de um *workflow* se dá no momento em que, dado um horário inicial, o mesmo é associado a um paciente existente na base de dados).

As atividades de um *workflow* mantêm, todavia, de acordo com o modelo, associações com objetos da classe *DBActor* e *DBResource*. Essas associações são extremamente importantes, pois informam quais recursos reais (cadastrados na base de dados) estão alocados, tendo como parâmetro, os respectivos objetos da classe *Actor* ou *Resource*.

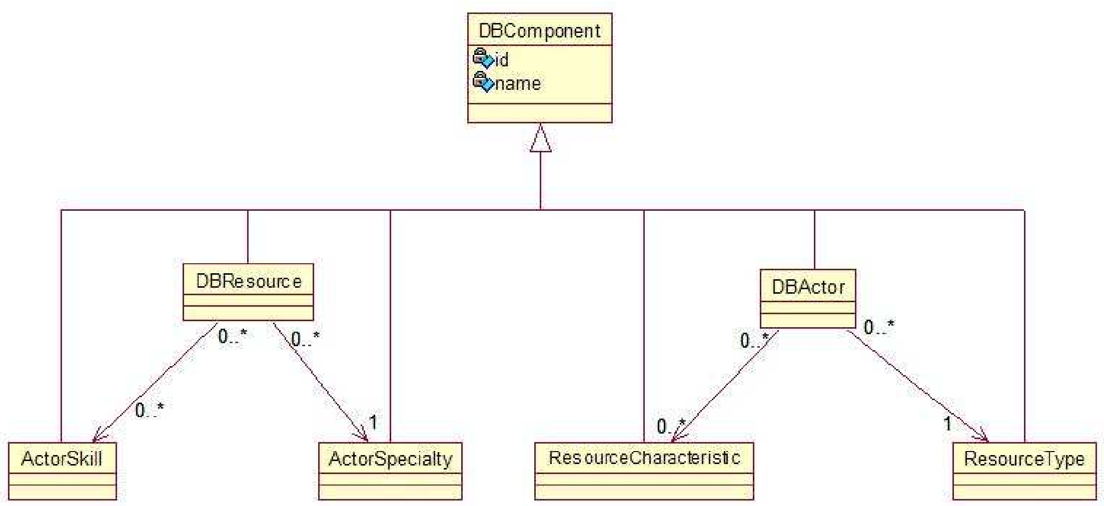

Figura 10 - Particularidades dos recursos humanos e materiais

Os atores e recursos cadastrados no sistema possuem, ainda, uma particularidade que, para melhor entendimento do conceito global, foi ocultada do modelo representado anteriormente. Eles dispõem de características

específicas que tornam possível que a máquina de inferência tenha parâmetros atemporais de comparação entre eles. Em outras palavras, esses objetos possuem “habilidades”, que podem variar de um para outro (Figura 10).

Um recurso humano, representado por objetos da classe *DBActor* possui uma especialidade (*ActorSpecialty*), bem como um conjunto de habilidades (*ActorSkills*); já os recursos materiais (*DBResource*) caracterizam-se por um tipo (*ResourceType*) e um conjunto de características (*ResourceCharacteristics*).

4.3.3 O Controle de Agendamentos

Um dos requisitos os quais o presente trabalho visa preencher para alcançar seu objetivo principal diz respeito ao gerenciamento inteligente dos recursos, com relação à sua disponibilidade temporal, especificamente.

Para tanto, fez-se necessário o desenvolvimento de um conjunto de artefatos de software que possibilite a realização de um controle adequado dos agendamentos.

A máquina de inferência proposta dispõe de um módulo genérico, dedicado ao controle automatizado dos agendamentos de recursos. Objetos deste módulo podem ser facilmente agregados a qualquer objeto, e são responsáveis por evitar que dois recursos sejam alocados à mesma atividade em horários conflitantes, além de indicar o horário mais adequado, caso o período desejado esteja totalmente preenchido.

As classes desenvolvidas no pacote de agendamento fornecem, também, métodos para testes de disponibilidade, que avaliam se um recurso está disponível ou não em dado período de tempo.

Vale destacar que o uso deste pacote de classes, cujas classes são demonstradas na Figura 11, é imprescindível para o mecanismo de alocação inteligente de recursos, pois, graças a ele, os problemas ocasionados por colisões e pelo surgimento de janelas (lacunas) de tempo ocioso ocasionado pelo mau gerenciamento temporal são resolvidos.

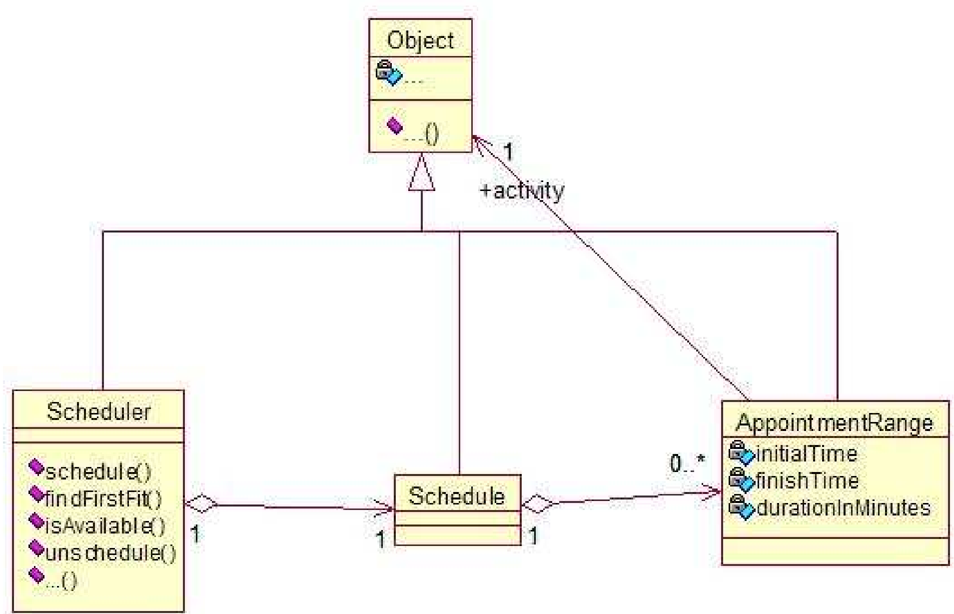

Figura 11 - Classes do pacote de agendamento

4.3.4 O Algoritmo de Configuração de *Workflows*

No sistema proposto, os recursos são alocados aos *workflows* por meio de alguns tipos específicos de operadores, correspondentes a metas de, basicamente, duas classes relevantes: metas de alocação em nível de *workflow* (*WorkflowConfigurationGoal*) e metas de alocação em nível de atividade (*ActivityConfigurationGoal*). A primeira delas é necessária para que o fluxo de execução do processo (o qual, geralmente, configura-se como uma árvore de

dados por apresentar hierarquias) seja percorrido. A segunda classe de metas de alocação é responsável por promover, para cada atividade isoladamente, a alocação de fato.

Sendo assim, uma meta da classe *WorkflowConfigurationGoal* é responsável por identificar cada atividade ao longo do workflow, derivando, para cada uma delas, sua respectiva meta de alocação de recursos (*ActivityConfigurationGoal*). O algoritmo que percorre o fluxo do workflow possui, naturalmente, uma característica recursiva de execução que lhe dá a flexibilidade necessária para a resolução do problema.

```
[Método percorra(umWorkflowComponent)]

se umWorkflowComponent é uma Activity, então:
      reduzaMeta(ActivityConfigurationGoal(umWorkflowComponent));
      percorra (umWorkflowComponent.proximo);
Senão:
      percorra (umWorkflowComponent.proximo);
fim-se;
```

Figura 12 - Algoritmo de Configuração de Workflow

Naturalmente, o algoritmo real implementado no sistema apresenta uma complexidade maior do que tal apresentado no pseudocódigo simplificado da Figura 12, pois trata inúmeras situações peculiares de cada tipo de componente (p.e., tratar múltiplos fluxos de execução originários de um *Split*), além de outras situações internas de programação. Entretanto, ele é suficiente para demonstrar a essência da lógica recursiva aplicada.

4.3.4.1 Os Critérios de Decisão para a Alocação

As metas de configuração de atividades (*ActivityConfigurationGoal*) são, por sua vez, as que desempenham a alocação de maneira efetiva. Consiste, basicamente, em uma sucessão de buscas à base de dados com o intuito de filtrar os atores e recursos que melhor se adequam às necessidades apresentadas.

Em um exemplo inicial, pode-se imaginar uma situação onde se tem um exame radiológico contrastado a ser realizado, cujo risco de ocorrência de reações alérgicas demanda a utilização de um carrinho para reanimação cardiovascular e que o médico executor, além de ser especialista em radiologia, disponha de experiência no tratamento de uma possível reação anafilática provocada pelo contraste. A primeira ação (etapa 1) exercida pelo algoritmo de configuração seria buscar os aparelhos de raios-X e carrinhos de reanimação disponíveis, bem como selecionar, dentre todos os médicos radiologistas, aqueles que possuem experiência no tratamento emergencial de reações ao contraste.

É possível notar que, até o momento, o algoritmo pode ter encontrado mais de um aparelho de reanimação, mais de um aparelho de raios-X e, também, mais de um médico cujas características coincidem com as dos perfil necessário. Escolher um entre os selecionados de maneira aleatória seria uma solução possível. Avancemos, entretanto, mais um passo com relação ao exemplo citado.

Suponha, agora, que no mesmo horário inicial do exame radiológico, deseja-se realizar, também, uma sessão de acupuntura. Suponha-se, ainda, que um dos médicos radiologistas candidatos a realizar o suposto exame seja, dentre todos eles, o único habilitado em medicina chinesa. Caso, coincidentemente, esse médico fosse o "sorteado" para execução do exame, a sessão de acupuntura teria de ser cancelada ou transferida para outro horário, pois, seu único possível executor não estaria disponível naquele momento. Essa situação apresenta, claramente, um problema de mau aproveitamento dos recursos que, no entanto, é resolvido pelo algoritmo de configuração disparado pelo motor de inferência.

Após selecionar, embora sem critérios mais específicos, para cada perfil requerido, todos os recursos correspondentes, o algoritmo parte para sua segunda etapa (etapa 2). A continuação do algoritmo consiste em um laço recursivo e divide-se em outras duas partes. Inicialmente (etapa 2.1), são selecionados do conjunto somente os recursos disponíveis para alocação durante o período de execução da atividade. Esse procedimento gera um conjunto possivelmente reduzido, descartando os recursos aptos, porém indisponíveis.

Por fim (etapa 2.2), aplica-se o critério de decisão, que consiste em alocar, para cada perfil, o recurso que possua o menor número de atributos possível, já que todos os candidatos selecionados até o momento são plenamente aptos para a atividade, e qualquer atributo excedente caracteriza desperdício de recurso.

Como, obviamente, todos os recursos alocados para a atividade devem trabalhar simultaneamente, o algoritmo verifica se todos eles foram alocados para o horário em questão. Caso algum recurso não tenha sido alocado (devido a alguma indisponibilidade de tempo), é definido para a atividade um novo horário inicial, posterior ao atual e, com base nele, é repetida a etapa 2. O processo se repete até que seja possível a alocação de todos os recursos necessários (Figura 13).

```
[Método configureAtividade(atividade, horaInicial)]

      [etapa1]
            aptos := selecioneAptos();

      [etapa2(horaInicial)]
            [etapa 2.1]
                  disponiveis := selecioneDisponiveis(horaInicial,aptos);
                  maisAptos := selecioneIdeais(disponiveis);
            [etapa 2.2]
                  Se encontrouTodos então:
                        AloqueRecursos(maisAptos);
                  Senão:
                        novaHoraInicial := encontreNovaHoraInicial();
                        etapa2(novaHoraInicial);
                  fim-se;
```

Figura 13 - Algoritmo Configuração de Atividades

4.3.5 Gerenciando as Alocações

É considerável a contribuição para a melhoria nos processos das instituições hospitalares graças à alocação racional e dinâmica de recursos promovida pelo motor de inferência proposto. Contudo, sua flexibilidade não se restringe apenas a esse ponto. O sistema permite que, de acordo com as necessidades, os recursos já alocados a determinado *workflow* possam ser liberados, a fim de serem disponibilizados para alocação em outros *workflows* que, por ventura, sejam mais urgentes ou prioritários. Deste modo, é possível

obter a flexibilidade necessária para que a ação de um ser humano compense as limitações de uma máquina.

4.3.6 O Desenvolvimento de Testes para Validação

O produto resultante do conjunto de artefatos desenvolvidos neste trabalho, por se tratar de *software* residente no lado servidor do sistema, apresenta a característica peculiar de rodar de maneira "silenciosa" para o usuário. Embora o sistema *Workflow Client* - cuja função é intermediar o usuário final com o servidor - faça parte do sistema de gerenciamento na íntegra, o mesmo encontra-se, no momento, em fase de desenvolvimento, fato este que inviabiliza sua utilização.

No entanto, a fim de validar o modelo proposto e, também, de verificar e conferir o alcance dos objetivos, foi desenvolvida uma ferramenta que possibilita a realização de baterias de teste. Essa ferramenta é composta, na verdade, por um conjunto de módulos. Os três primeiros módulos consistem em interfaces gráficas, sendo que o primeiro possibilita visualização e gerenciamento simplificado das configurações de *workflows*. O segundo permite a visualização dos agendamentos. O terceiro módulo, por sua vez, permite ações como carregamentos de *workflows* gerados pelo modelador (*Workflow Designer*) e instanciações (associação de um *workflow* a um paciente para alocação em dado horário inicial). Já o quarto módulo é responsável por realizar simulações de situações reais e consiste em disparar, automaticamente, grandes números de metas de alocações de workflow em horários aleatórios dentro de certo período.

O módulo de visualização de processos da máquina de inferência, como dito anteriormente, emula, de maneira simplificada, o funcionamento do cliente de *workflow*, possibilitando o acompanhamento das atividades de cada um dos *workflows* instanciados, bem como a visualização das metas em execução, recursos alocados, horário previsto e real de alocação, etc.

A interface (Figura 14) mostra todos os recursos humanos (*Actors*) e materiais (*Resources*) cadastrados na base de dados, sendo possível, por meio de um duplo-clique, a visualização dos agendamentos efetuados para cada um deles, dentro do período desejado.

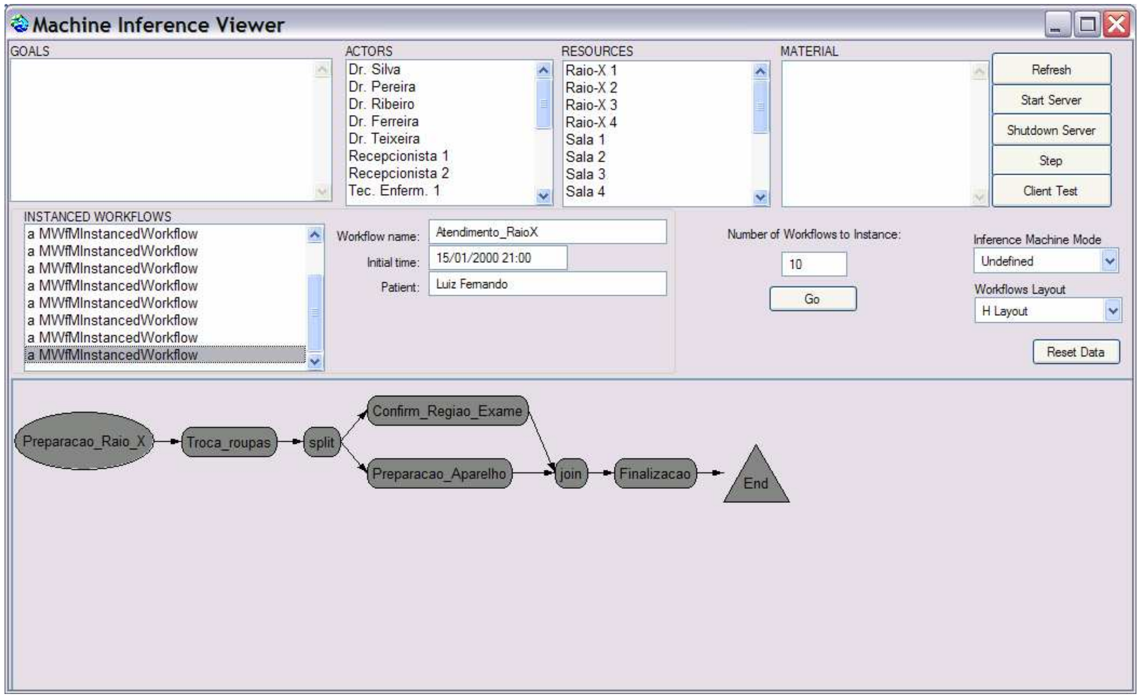

Figura 14 - Tela do Visualizador de processos da máquina de inferência

O visualizador de processos permite, ainda, que cada atividade seja visualizada em mais detalhes, mostrando informações como recursos alocados, perfis desejados, entre outros (Figura 15). O exemplo ilustrado pela Figura 16

mostra alguns agendamentos de atividades alocadas para um recurso humano (Ator), que representa o médico “Dr. Ribeiro”.

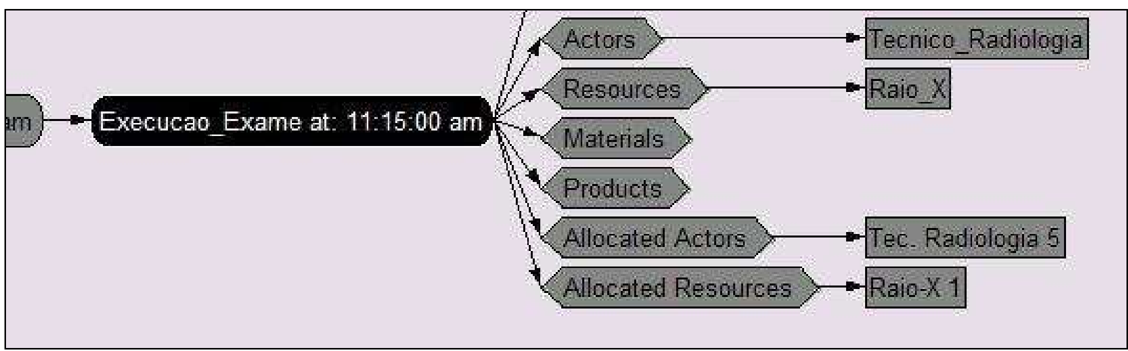

Figura 15 - Atividade vista em detalhes através do visualizador de processos

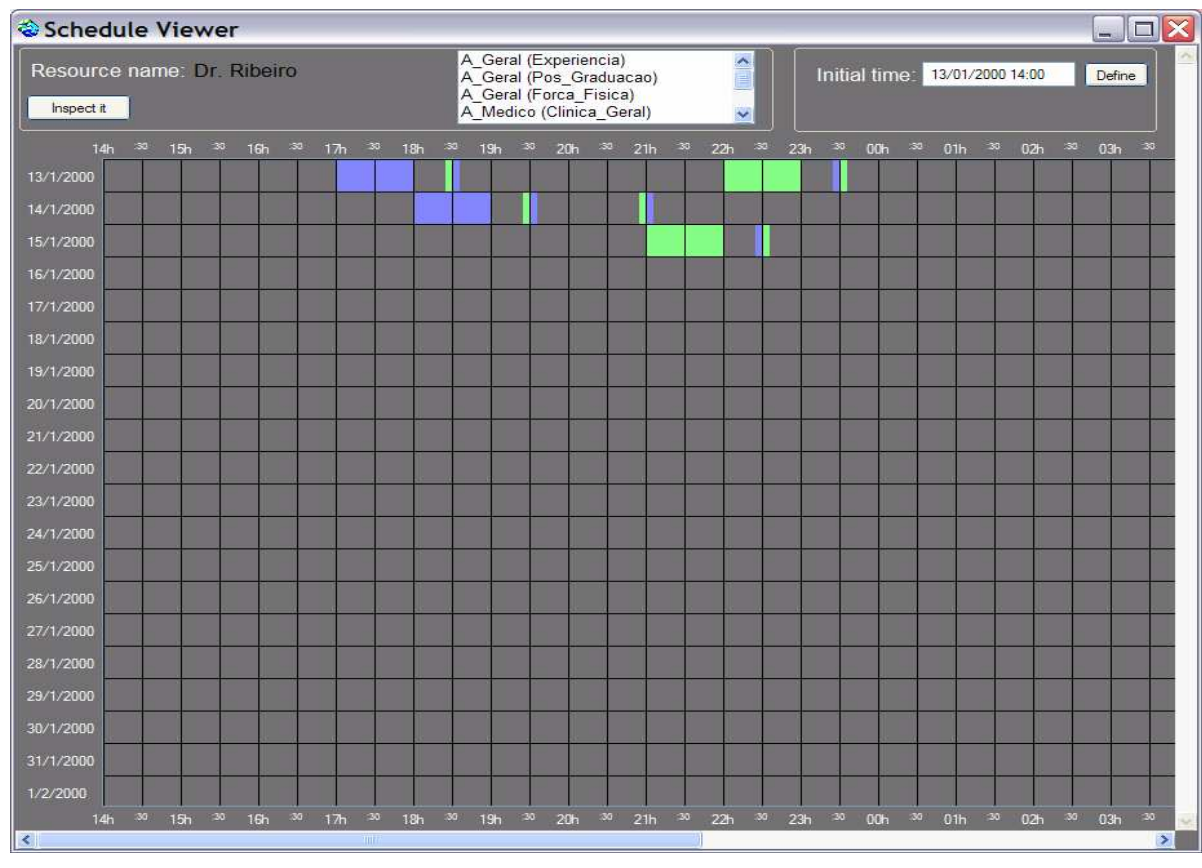

Figura 16 – Diagrama de utilização de recursos

5 RESULTADOS E DISCUSSÃO

Os testes e simulações realizados por meio das ferramentas desenvolvidas proporcionaram uma visão ampla e diferenciada, a qual permitiu que algumas constatações fossem feitas. Essas constatações, por sua vez, convergem para algumas conclusões sobre os resultados obtidos com o desenvolvimento do trabalho relatado.

Em um primeiro teste realizado, que se caracterizou pela realização de 200 configurações de workflows representando exames de Raio-X, cujo horário inicial variou, aleatoriamente, entre 01/10/2005 00:00h e 09/10/2005 23:59h, observou-se uma otimização generalizada no aproveitamento dos recursos, baseando-se nos critérios já citados e justificados neste trabalho. Para comprovar esse fato, basta observar as alocações referentes à atividade de laudo, cuja duração é de 30 minutos[4]. Essas alocações são mostradas na Figura 17, Figura 18 e Figura 19.

No exemplo, dentre os médicos cadastrados na clínica, o ator de nome "Dr. Ribeiro", graças ao menor número de características excedentes às necessárias, configura-se como o mais apto para realização do laudo, seguido por "Dr. Pereira" e "Dr. Silva", respectivamente. Os atores de nome "Dr. Ferreira" e "Dr. Teixeira", por não possuírem todas as características necessárias

[4] No visualizador de agendamentos, cada alocação é representada por uma única cor (verde ou azul). A alternância de cores tem a finalidade de distinguir uma alocação de outra.

(Experiência e Radiologia) para realização da atividade, foram descartados pelo algoritmo. Observando os agendamentos de cada recurso em particular, percebeu-se que, para os recursos mais aptos, há um maior número de alocações. Com isso, concluiu-se que o sistema proposto proporciona, de fato, uma maior disponibilidade por parte dos recursos que possuem um número maior de características excedentes.

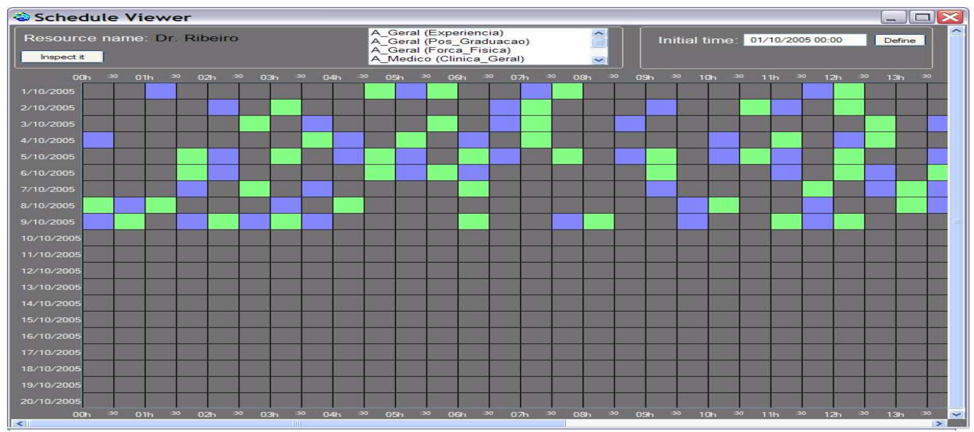

Figura 17 – Agendamentos de "Dr. Ribeiro" após 200 configurações

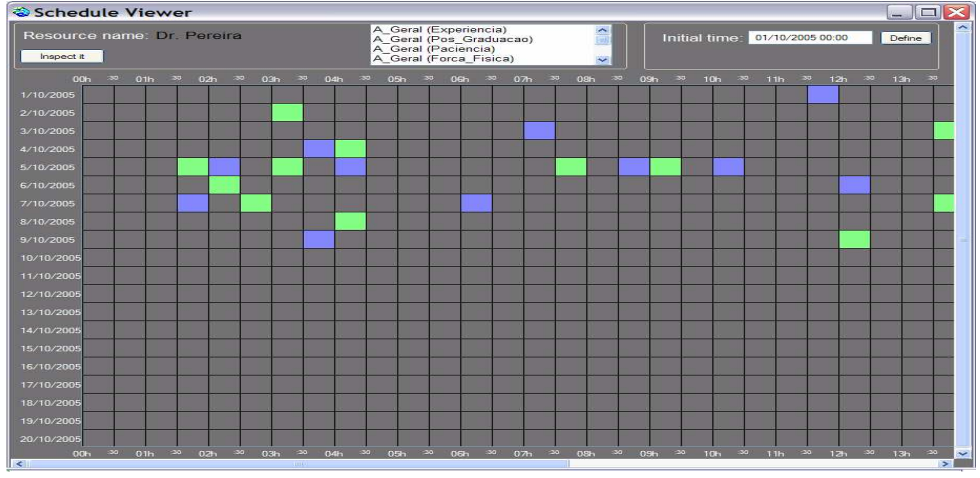

Figura 18 – Agendamentos de "Dr. Pereira" após 200 configurações

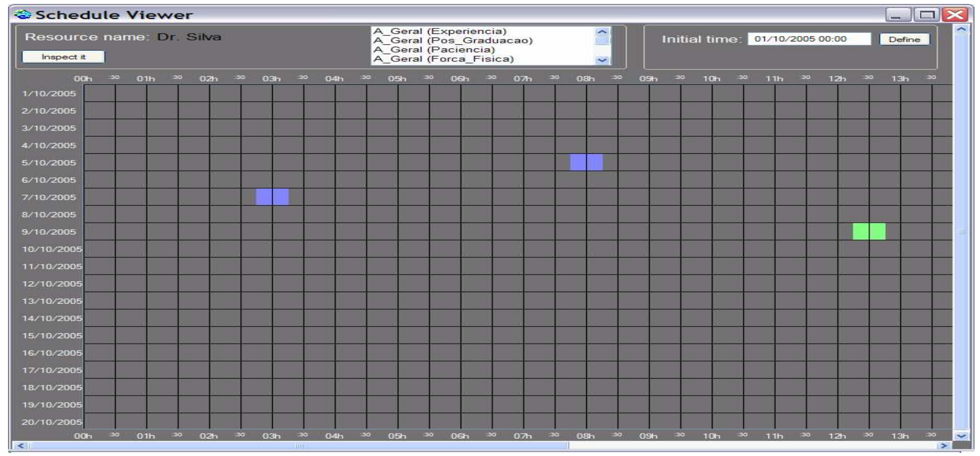

Figura 19 - Agendamentos de "Dr. Silva" após 200 configurações

Após a realização de mais uma bateria de 800 alocações (totalizando 1000), foi possível comprovar a ação do algoritmo de configuração sob um outro aspecto. Observando as imagens, constata-se que "Dr. Ribeiro" encontra-se com sua agenda quase totalmente lotada no período no qual as alocações aleatórias foram disparadas. Vale ressaltar que "Dr. Pereira" e, principalmente, "Dr. Silva" encontram-se com mais horários livres. Isto significa que, além da melhoria na disponibilidade dos recursos, houve, também, uma otimização do aproveitamento do tempo disponível, minimizando desperdícios em cada um dos recursos no experimento realizado.

Figura 20 - Agendamentos de "Dr. Ribeiro" após 1000 configurações

Figura 21 - Agendamentos de “Dr. Pereira” após 1000 configurações

Figura 22 - Agendamentos de "Dr. Silva" após 1000 configurações

Após realizados os testes de configuração em massa, cujos resultados podem ser vistos através da Figura 20, Figura 21 e Figura 22, foi possível perceber que, após saturadas as agendas dos recursos, a indisponibilidade causou atrasos muitas vezes intoleráveis (chegando à dimensão de dias) no horário inicial de algumas atividades. Esses atrasos, em determinados processos críticos, tornam inviável a execução real dos mesmos. Apesar de a máquina de inferência ser capaz de resolver este problema através da desalocação e realocação de workflows menos prioritários (necessitando, todavia, da interação manual humana no processo), há métodos mais eficazes para a resolução totalmente automatizada deste problema, os quais poderiam ser aplicados.

Outro ponto crítico constatado no modelo em questão reside no fato de que o algoritmo de configuração, por ser constituído de buscas sucessivas à base de dados, consiste em algo cuja complexidade cresce exponencialmente, à medida que aumenta a quantidade, bem como a indisponibilidade dos recursos. Em casos extremos, onde o universo da base de conhecimento é muito extenso e há uma saturação nos agendamentos, o tempo de resposta da máquina de inferência pode ser insatisfatório.

5.1 Restrições do estudo

O presente trabalho encontrou-se restrito ao desenvolvimento de módulos de software, configurando parte específica do motor de inferência para gerenciamento de *Workflow* médico. Embora tenha demandado inúmeras tarefas adicionais, as quais tornaram viável sua validação, o trabalho limitou-se,

basicamente, à representação de workflows e recursos no sistema, assim como todo o mecanismo de configuração pertinente.

É importante ressaltar que o sistema de gerenciamento na sua íntegra é considerado completo e possivelmente implantado após a conclusão de duas dissertações de mestrado, cujo desenvolvimento depende dos resultados aqui obtidos.

Por tratar-se de um trabalho de pesquisa em fase inicial, o desenvolvimento do sistema deu-se por prototipação. Também por este motivo, não coube a este trabalho preocupar-se com um modelo conceitual completo e perfeitamente fiel ao modelo de informações da tecnologia de workflow. Pretendeu-se, todavia, criar um cenário que torne possível a representação do fluxo de execução e dos recursos, e que viabilize a aplicação de métodos para promover a alocação inteligente dos mesmos.

Os testes de validação, desempenho e obtenção de resultados das partes desenvolvidas foram realizados em laboratório, tentando, todavia, aproximar-se da realidade dos processos de uma clínica radiológica.

5.2 Sugestões para Trabalhos Futuros

As limitações e resultados constatados através da validação deste trabalho permitem que sejam levantados alguns pontos, os quais podem ser explorados em trabalhos de pesquisas futuras, em prol do desenvolvimento dos conhecimentos relacionados ao tema de gerenciamento de *workflow* em geral.

Deixa-se como proposta a expansão do modelo proposto, de modo a efetuar a aplicação de alguma técnica viável que permita que, em situações de

saturação de agendas, o sistema indique os melhores horários cujos atrasos entre execuções de atividades são nulos ou estão dentro de certo limite de tolerância estipulado. Isto fará com que o modelo proposto apresente, nesses casos, resultados mais próximos à realidade do processo.

Além disso, são válidos os eventuais esforços futuros no sentido de desenvolver o algoritmo de configuração para maximização de performance, a fim de obter menores tempos de resposta.

REFERÊNCIAS

ABDALA, Daniel Duarte; WANGENHEIM, Aldo Von. **Conhecendo Smalltalk.** 1. ed. Florianópolis: Visual Books, 2002.

BRASIL. Resolução nº 1638, de 10 de julho de 2002. **Conselho Federal de Medicina,** Brasília, DF, 2002. Disponível em <http://www.portalmedico.org.br/>. Acesso em: 5 dez. 2004.

CNPq. Proposta de financiamento de projeto nº 507241/2004-5. **Cyclops Hospital Management System - Desenvolvimento e Validação de um Modelo de Sistema de Workflow Management Inteligente para Hospitais.** Florianópolis, 2004. Disponível em: <http://www.cnpq.br/>. Acesso em: 15 mar. 2005.

DICOM HOMEPAGE. Disponível em: <http://medical.nema.org/>. Acesso em: 6 dez. 2004.

DOYLE, P. **AI QUAL SUMMARY: PLANNING.** Site de Disciplina de Inteligência Artificial da Universidade de Darthmouth, 1998. Disponível em: <http://www.cs.dartmouth.edu/>. Acesso em: 28 mai. 2005.

FABER, Kerstin et al. **Modelling of Radiological Examinations with POKMAT, a Process Oriented Knowledge Management Tool**. In: *Conference on AI in Medicine in Europe*, 8., 2001, [S.l]. Artificial Intelligence Medicine. Londres: Springer-Verlag, 2001. p. 409 - 412.

FERREIRA, Levi; RIBEIRO, Manassés. **Gerenciamento de Processos Hospitalares Utilizando Tecnologia de Workflow.** Florianópolis: UFSC, 2003.

HOLLINGSWORTH, David. **The Workflow Reference Model.** Hampshire: Workflow Management Coalition, 1995. Disponível em: <http://www.wfmc.org>. Acesso em: 5 dez. 2004.

LUGER, George F. **Inteligência Artificial - Estruturas e Estratégias para a Solução de Problemas Complexos**. 4. ed. [S. l.], Bookman, 2004.

NILSSON, Nils J.; FIKES, Richard E. **STRIPS - A new approach to the application of theorem proving to problem solving**. 2. ed. California: Stanford Research Institute, 1971. Disponível em: <http://ai.stanford.edu/>. Acesso em: 10 jan. 2005.

RIBEIRO, Manassés. **Cyclops Workflow Server - Uma Proposta para um modelo de um servidor de fluxo de atividades médico-hospitalar**. Florianópolis, SC: Projeto Cyclops, 2005.

RAUSCH-SCHOTT, Stefan. **TRIGSflow-Workflow Management Based on Active Object-Oriented Database Systems and Extended Transaction Mechanisms.** 1997. Tese (Doutorado) - Institute Of Applied Computer Science, Johannes Kepler University, Linz, Áustria, 1997.

RICHTER, Michael M. **Knowledge Based Systems Introduction.** University of Kaiserslautern, 2001. Apresentação de slides.

RUSSELL, Stuart; NORVIG, Peter. **Artificial Intelligence - A Modern Approach.** 2. ed. [S.I.]. Prentice Hall, 1995.

SUNY - Center for Technology in Government. **An Introduction to Workflow Management Systems.** University of Albany, 1997.

THE CYCLOPS HOMEPAGE. Disponível em: <http://cyclops.telemedicina.ufsc.br>. Acesso em: 6 dez. 2004.

ANEXOS

ANEXO A – ESPECIFICAÇÃO DE REQUISITOS DO SISTEMA

Legenda:
F – Requisito funcional
NF – Requisito não-funcional associado

F1 Registrar Workflows Abstratos	**Oculto (x)**			
Descrição: O sistema deve ser capaz de persistir uma estrutura de dados, representando workflows abstratos (não associados a recursos reais).				
Requisitos Não Funcionais				
Nome	**Restrição**	**Categoria**	**Desejável**	**Permanente**
NF1.1 Efetuar Parsing de conversão.	Os workflows abstratos modelados através de uma ferramenta externa devem ser carregados no sistema através da tradução de arquivos XML.	Compatibilidade	**( )**	**(x)**
NF1.2 Efetuar validação sintática	Os arquivos XML devem ser analisados a fim de evitar erros de grafia nos arquivos-fonte.	Compatibilidade	**( )**	**(x)**

F2 Instanciar Workflows	**Oculto ( )**
Descrição: O sistema deve Associar Workflows Abstratos a pacientes registrados, caracterizando, assim, um workflow real configurável.	

F3 Gerenciar agendamentos de Recursos	**Oculto (x)**			
Descrição: O sistema deve manter, para cada recurso cadastrado, uma agenda de gerenciamento automático, a fim de evitar colisões de horários e sugerir, dado um determinado momento, os próximos horários de disponibilidade.				
Requisitos Não Funcionais				
Nome	**Restrição**	**Categoria**	**Desejável**	**Permanente**
NF3.1 Visualização dinâmica de agendamentos	Deve ser possível que cada recurso disponha de uma tela para visualização de seus agendamentos, cuja atualização ocorra em tempo real.	Interface	**(x)**	**( )**

F4 Representar Planos hierárquicos não-lineares.	**Oculto ( )**			
Descrição: O sistema deve permitir a representação de planos hierárquicos (sub-workflows recursivos, isto é, onde um workflow contenha outros workflows em níveis mais baixos de refinamento) e não-lineares (capazes de contemplar a representação de atividades de execução paralela).				
Requisitos Não Funcionais				
Nome	**Restrição**	**Categoria**	**Desejável**	**Permanente**
NF4.1 Visualização do fluxo de execução.	Deve ser possível a representação gráfica dos workflows instanciados no sistema, que permita a visualização de informações pertinentes à alocação, bem como interação com o sistema.	Interface	**(x)**	**(x)**

F5 Efetuar Configuração de atividades	**Oculto (x)**			
Descrição: O sistema deve ser capaz de percorrer workflows e, para cada uma de suas atividades, alocar automaticamente os mais apropriados recursos com horário disponível.				
Requisitos Não Funcionais				
Nome	**Restrição**	**Categoria**	**Desejável**	**Permanente**
NF5.1 Mecanismo de disparo de alocações sucessivas aleatórias	Deve ser possível, por parte do sistema, a simulação automatizada de configurações. Deve ser dado pelo usuário o número de workflows a serem configurados. As configurações devem ocorrer com horários iniciais aleatórios, dentro de um certo período, variando randomicamente entre todos os workflows abstratos cadastrados.	Interface	**(x)**	**( )**

F6 Efetuar dealocação de recursos	**Oculto (x)**			
Descrição: O sistema deve ser capaz de desfazer configurações efetuadas, a partir da atividade desejada, no fluxo de execução. Com isso, deseja-se liberar recursos em situações inesperadas, para que sejam utilizados em possíveis realocações.				
Requisitos Não Funcionais				
Nome	**Restrição**	**Categoria**	**Desejável**	**Permanente**
NF6.1 Escolha do ponto de dealocação	Deve ser possível, graficamente, a dealocação através da seleção da atividade desejada.	Interface	**(x)**	**( )**

ANEXO B – ARTIGO SOBRE O TRABALHO

Cyclops Medical Workflow Engine - Aplicando Configuração em um Sistema de Gerenciamento de Workflow Hospitalar

Luiz Fernando da Silva Pereira
Universidade Federal de Santa Catarina
Sistemas de Informação
luizfsp@inf.ufsc.br

Resumo

Os ambientes hospitalares, de um modo geral, caracterizam-se pela dificuldade ocasionada pela má utilização de recursos compartilhados, os quais apresentam disponibilidade limitada, apesar da alta demanda de utilização. Outra característica nesses ambientes é a dinamicidade da execução de seus processos, a qual requer agilidade no gerenciamento, porém, geralmente, não é contemplada adequadamente. Entretanto, visando minimizar esses problemas, medidas podem ser adotadas. A utilização racionalizada dos recursos disponíveis, aliada ao gerenciamento adequado dos processos, apresenta-se como uma solução viável. Uma aplicação prática dessa solução é apresentada pelo Projeto Cyclops, com o desenvolvimento de um Sistema de Gerenciamento de Workflow, dotado de um motor de inferência inteligente, que visa otimizar a utilização de recursos de saúde. Nesse contexto, o trabalho descrito neste artigo constitui parte importante do sistema, e caracteriza-se por proporcionar a comprovação de uma maximização no aproveitamento dos recursos, minimizando desperdícios funcionais e temporais. Através do uso de conceitos de planejamento e configuração, o trabalho relata o desenvolvimento, assim como o resultado de experimentos realizados, os quais viabilizaram a validação da proposta de uma estratégia de implementação em ocasiões futuras.

1. Introdução

Os processos de um ambiente hospitalar são caracterizados pela sua natureza dinâmica. É grande a responsabilidade sobre todos os agentes que fazem parte da execução das suas atividades diárias, visto que a saúde humana é um tema delicado e serviços de qualidade devem ou, pelo menos, deveriam ser prestados nessa área de atuação.

Gerenciar e controlar atividades de saúde não é uma tarefa simples. Essa dificuldade é visível em uma visita ao setor de clínica médica da maioria dos hospitais públicos no Brasil. Tendo em vista a necessidade, tornou-se interessante a adoção de um sistema que gerencie as atividades de forma dinâmica.

Diante desse cenário, e considerando que quaisquer processos de atividades, sobretudo os de um ambiente hospitalar, requerem uma forma adequada de modelagem, uma boa representação e controle de processos é um ponto importante para a otimização da sua execução. A utilização de sistemas de gerenciamento de Workflow apresenta, nesse contexto uma solução prática e viável.

Segundo a WfMC (the Workflow Management Coalition), o termo workflow (em português, "fluxo de trabalho") visa a automação de um processo de negócios. Com base nesse conceito, a filosofia de workflow propõe uma maneira simples, porém adequada de representar e manusear informações sobre as atividades em um contexto de negócios, favorecendo o gerenciamento e a reengenharia.

Em um exemplo, pode-se imaginar uma situação em que uma cirurgia está para ser realizada. Certamente, para a execução desta atividade, uma série de aparelhos e profissionais foram previamente alocados, respeitando-se uma política de regras que leva em conta uma série fatores, como horários, logística, habilidades dos atores, características dos equipamentos, entre outras. Agora, suponha que, minutos antes do início do evento, ocorra um incidente qualquer, porém grave o suficiente para inviabilizar a realização da cirurgia naquelas circunstâncias. Com um sistema inteligente de gerenciamento de

alocações em workflows dinâmicos, seria possível obter, em tempo hábil, uma nova configuração de todos os recursos mais propícios para a execução da mesma cirurgia, utilizando o tempo da melhor maneira possível, dada a nova situação.

Essa alocação dinâmica e racionalizada de recursos é possível graças a um conjunto de elementos que viabilizam o processo de tomada de decisão específica ao problema.

2. Teoria do Mecanismo de Configuração

A escolha dos recursos que serão agendados para cada atividade do processo depende de alguns fatores relativos ao estado corrente do universo do problema. Em outras palavras, para que seja definida a melhor seqüência de ações possível, é necessário que sejam avaliadas todas as características de cada uma delas para que, de acordo com a necessidade, sejam descartadas aquelas irrelevantes, assim como selecionada a mais apropriada. Essa escolha parametrizada do elemento apropriado dentre um conjunto de vários possíveis é desempenhada através de um mecanismo de configuração.

As técnicas de configuração caracterizam-se por buscas sucessivas a uma base de conhecimento extensa, porém finita e bem definida acerca de cada elemento constante no universo do problema (RICHTER, 2001). Cada um desses elementos possui um conjunto de atributos que visa atender ou não a um conjunto de requisitos. Com base em heurísticas (regras lógicas pré-determinadas), é eleito, então, o elemento que melhor atende à necessidade em dada situação. A lógica genérica do processo de configuração pode ser entendida como um fluxo recursivo, onde sua parada ocorre no momento onde, no estado final, todos os requisitos são satisfeitos (Figura 23).

Figura 23 - Idéia do Algoritmo de Configuração (RICHTER, 2001)

3. Aspectos de implementação

A solução desenvolvida, sob o ponto de vista da essência do seu funcionamento, é caracterizada por três funções distintas: gerenciar as configurações, controlar os agendamentos e aplicar os critérios para seleção de recursos.

3.1. Gerenciando as configurações

No sistema proposto, os recursos são alocados aos *workflows* por meio de ações (denominados “operadores”), correspondentes a objetivos (“metas”) de, basicamente, dois tipos: metas de alocação em nível de processo (“*Workflow Configuration Goal”*) e metas de alocação em nível de atividade (“*Activity Configuration Goal”*). A primeira delas é necessária para que o fluxo de execução do processo seja percorrido. A segunda classe de metas de alocação é responsável por promover, para cada atividade isoladamente, o agendamento dos recursos.

Sendo assim, uma meta do tipo “*Workflow Configuration Goal”* é responsável por identificar cada atividade ao longo do fluxo, derivando, para cada uma delas, sua respectiva meta de alocação de recursos (“*Activity Configuration Goal”*). O algoritmo que percorre o fluxo possui, naturalmente, uma característica recursiva de execução que lhe dá a flexibilidade necessária para a resolução do problema.

```
[Método percorra(umWorkflowComponent)]

se umWorkflowComponent é uma Activity, então:
     reduzaMeta(ActivityConfigurationGoal(umWorkflowComponent));
     percorra (umWorkflowComponent.proximo);
Senão:
     percorra (umWorkflowComponent.proximo);
fim-se;
```

Figura 24 - Algoritmo de Configuração de *Workflow*

3.2. Controlando Agendamentos

Um dos requisitos da solução apresentada diz respeito ao gerenciamento inteligente dos recursos, com relação à sua disponibilidade temporal. Para tanto, fez-se necessário o desenvolvimento de um conjunto de artefatos de software que possibilite a realização de um controle adequado dos agendamentos.

O sistema dispõe de um módulo genérico, dedicado ao controle automatizado dos agendamentos de recursos. Este módulo é responsável por evitar que dois ou mais recursos sejam alocados à mesma atividade em horários conflitantes, além de indicar o horário mais adequado, caso o período desejado esteja totalmente preenchido.

O pacote de controle de agendamentos fornece, também, recursos para testes de disponibilidade, que avaliam se um recurso está disponível ou não em dado período de tempo.

Vale destacar que o uso deste pacote é imprescindível para o mecanismo de alocação inteligente de recursos, pois, graças a ele, os problemas ocasionados por colisões e pelo surgimento de janelas (lacunas) de tempo ocioso ocasionado pelo mau gerenciamento temporal são resolvidos.

3.2 Aplicando critérios de decisão para alocação

A meta de configuração de atividades (“*Activity Configuration Goal”*) é, por sua vez, a que desempenha o agendamento de maneira efetiva. Consiste, basicamente, em uma sucessão de buscas à base de dados com o intuito de filtrar os atores e recursos que melhor se enquadram às necessidades apresentadas.

Em um exemplo inicial, pode-se imaginar uma situação onde se tem um exame radiológico contrastado a ser realizado, cujo risco de ocorrência de reações alérgicas demanda a utilização de um carrinho para reanimação cardiovascular e que o médico executor, além de ser especialista em radiologia, disponha de experiência no tratamento de uma possível reação anafilática provocada pelo contraste. A primeira ação (etapa 1) exercida pelo algoritmo de configuração seria buscar os aparelhos de raios-X e carrinhos de reanimação disponíveis, bem como selecionar, dentre todos os médicos radiologistas, aqueles que possuem experiência no tratamento emergencial de reações ao contraste.

É possível notar que, até o momento, o sistema pode ter encontrado mais de um aparelho de reanimação, mais de um aparelho de raios-X e, também, mais de um médico cujas características coincidem com as do perfil necessário. Escolher um entre os selecionados de maneira aleatória seria uma solução possível. Avancemos, entretanto, mais um passo com relação ao exemplo citado.

Suponha, agora, que no mesmo horário inicial do exame radiológico, deseja-se realizar, também, uma sessão de acupuntura. Suponha-se, ainda, que um dos médicos radiologistas candidatos a realizar o suposto exame seja, dentre todos eles, o único habilitado em medicina chinesa. Caso, coincidentemente, esse médico fosse o “sorteado” para execução do exame, a sessão de acupuntura teria de ser cancelada ou transferida para outro horário, pois, seu único possível executor não estaria disponível naquele momento. Essa situação apresenta um problema de mau aproveitamento dos recursos que, no entanto, é resolvido pelo algoritmo de configuração disparado pelo sistema proposto.

Após selecionar, embora sem critérios mais específicos, para cada perfil requerido, todos os recursos correspondentes, o algoritmo parte para sua segunda etapa (etapa 2). A continuação do algoritmo consiste em um laço recursivo e divide-se em outras duas partes. Inicialmente (etapa 2.1), são selecionados do conjunto somente os recursos disponíveis para alocação durante o período de execução da atividade. Esse procedimento gera um conjunto possivelmente reduzido, descartando os recursos aptos, porém indisponíveis.

Por fim (etapa 2.2), aplica-se o critério de decisão, que consiste em alocar, para cada perfil, o recurso que possua o menor número de atributos possível, já que todos os candidatos selecionados até o momento são plenamente aptos para a atividade, e qualquer atributo excedente caracteriza desperdício de recurso.

Como, obviamente, todos os recursos alocados para a atividade devem trabalhar simultaneamente, o algoritmo verifica se todos eles foram alocados para o horário em questão. Caso algum recurso não tenha sido alocado (devido a alguma indisponibilidade de tempo), é definido para a atividade um novo horário inicial,

posterior ao atual e, com base nele, é repetida a etapa 2. O processo se repete até que seja possível a alocação de todos os recursos necessários (Figura 13).

```
[Método configureAtividade(atividade, horaInicial)]

      [etapa1]
            aptos := selecioneAptos();

      [etapa2(horaInicial)]
            [etapa 2.1]
                  disponiveis := selecioneDisponiveis(horaInicial,aptos);
                  maisAptos := selecioneIdeais(disponiveis);
            [etapa 2.2]
                  Se encontrouTodos então:
                        AloqueRecursos(maisAptos);
                  Senão:
                        novaHoraInicial := encontreNovaHoraInicial();
                        etapa2(novaHoraInicial);
                  fim-se;
```

Figura 25 - Algoritmo Configuração de Atividades

4. Resultados

Através do desenvolvimento e realização de alguns, foi possível realizar algumas constatações, as quais convergem para algumas conclusões sobre os resultados obtidos com o desenvolvimento do trabalho relatado.

Em um primeiro teste realizado, caracterizado pela realização de 200 configurações de *workflows* representando exames de Raio-X, cujo horário inicial variou, aleatoriamente, entre 01/10/2005 00:00h e 09/10/2005 23:59h, observou-se uma otimização generalizada no aproveitamento dos recursos, baseando-se nos critérios já citados e justificados neste trabalho. Para comprovar esse fato, basta observar as alocações referentes à atividade de laudo, cuja duração é de 30 minutos. Essas alocações são mostradas na Figura 17, Figura 18 e Figura 19.

No exemplo, dentre os médicos cadastrados na clínica, o ator de nome “Dr. Ribeiro”, graças ao menor número de características excedentes às necessárias, configura-se como o mais apto para realização do laudo, seguido por “Dr. Pereira” e “Dr. Silva”, respectivamente. Os atores de nome “Dr. Ferreira” e “Dr. Teixeira”, por não possuírem todas as características necessárias (Experiência e Radiologia) para realização da atividade, foram descartados pelo algoritmo. Observando os agendamentos de cada recurso em particular, percebeu-se que, para os recursos mais aptos, há um maior número de alocações. Com isso, concluiu-se que o sistema proposto proporciona, de fato, uma maior disponibilidade por parte dos recursos que possuem um número maior de características excedentes.

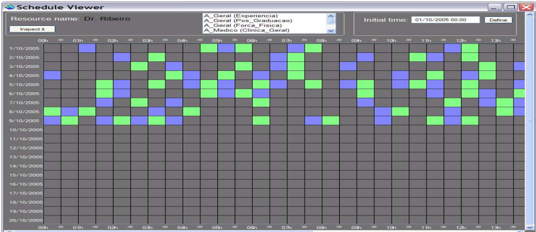

Figura 26 – Agendamentos de “Dr. Ribeiro” após 200 configurações

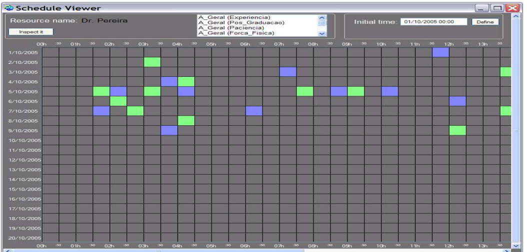

Figura 27 – Agendamentos de “Dr. Pereira” após 200 configurações

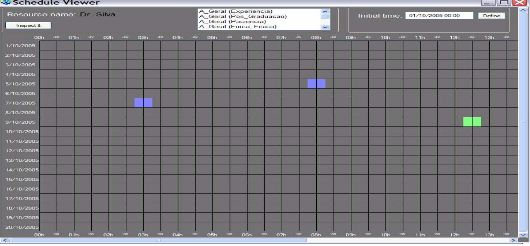

Figura 28 - Agendamentos de “Dr. Silva” após 200 configurações

Após a realização de mais uma bateria de 800 alocações (totalizando 1000), foi possível comprovar a ação do algoritmo de configuração sob um outro aspecto. Observando as imagens, constata-se que “Dr. Ribeiro” encontra-se com sua agenda quase totalmente lotada no período no qual as alocações aleatórias foram disparadas. Vale ressaltar que “Dr. Pereira” e, principalmente, “Dr. Silva” encontram-se com mais horários livres. Isto significa que, além da melhoria na disponibilidade dos recursos, houve, também, uma otimização do aproveitamento do tempo disponível, minimizando desperdícios em cada um dos recursos no experimento realizado.

Figura 29 - Agendamentos de "Dr. Ribeiro" após 1000 configurações

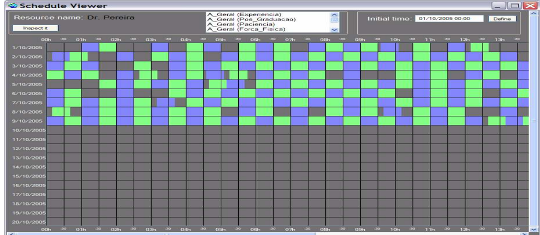

Figura 30 - Agendamentos de "Dr. Pereira" após 1000 configurações

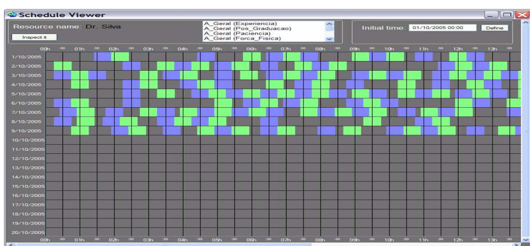

Figura 31 - Agendamentos de "Dr. Silva" após 1000 configurações

Após realizados os testes de configuração em massa, cujos resultados podem ser vistos através da Figura 20, Figura 21 e Figura 22, foi possível perceber que, após saturadas as agendas dos recursos, a indisponibilidade causou atrasos muitas vezes intoleráveis (chegando à dimensão de dias) no horário inicial de algumas atividades. Esses atrasos, em determinados processos críticos, tornam inviável a execução real dos mesmos. Apesar de o sistema apresentado ser capaz de resolver este problema através da desalocação e realocação de workflows menos prioritários (necessitando, todavia, da interação manual humana no processo), há métodos mais eficazes para a resolução totalmente automatizada deste problema, os quais poderiam ser aplicados.

Outro ponto crítico constatado no modelo em questão reside no fato de que o algoritmo de configuração, por ser constituído de buscas sucessivas à base de dados, consiste em algo cuja complexidade cresce exponencialmente, à medida que aumenta a quantidade, bem como a indisponibilidade dos recursos. Em casos extremos, onde o universo da base de conhecimento é muito extenso e há uma saturação nos agendamentos, o tempo de resposta da máquina de inferência pode ser insatisfatório.

7. Referências

RIBEIRO, Manassés. **Cyclops Workflow Server - Uma Proposta para um modelo de um servidor de fluxo de atividades médico-hospitalar**. Florianópolis, SC: Projeto Cyclops, 2005.

RICHTER, Michael M. **Knowledge Based Systems Introduction.** University of Kaiserslautern, 2001. Apresentação de slides.

ANEXO B – CÓDIGO-FONTE

```
Smalltalk defineClass: #MWfMWorkflowComponent
	superclass: #{Core.Object}
	indexedType: #none
	private: false
	instanceVariableNames: 'id name description parentComponent '
	classInstanceVariableNames: ''
	imports: ''
	category: 'CyclopsMWfM_Workflow_Data_Structure'

METHOD bindComponents: parent
"Must be implemented in the subclasses"

METHOD isParentOf: aWorkflowComponent
	| result parent |

	result := false.
	parent := aWorkflowComponent parent.
	[parent isNil] whileFalse:[
		result := (parent = self).
		parent := parent parent.
	].
	^result.

METHOD isSonOf: aWorkflowComponent
	| result parent |

	result := false.
	parent := self parent.
	[parent isNil] whileFalse:[
		result := (parent = aWorkflowComponent).
		parent := parent parent.
	].
	^result.

“-----------------------------------------------------------------------------------------------------“
Smalltalk defineClass: #MWfMWorkflow
	superclass: #{Smalltalk.MWfMWorkflowComponent}
	indexedType: #none
	private: false
```

```
	instanceVariableNames: 'version starts ends actors resources materials
products xmlFile activitys joins splits macros instancedWorkflow '
	classInstanceVariableNames: ''
	imports: ''
	category: 'CyclopsMWfM_Workflow_Data_Structure'
```

METHOD bindComponents: parent

```
	self starts do:[ :each|
		each bindComponents: self.
	].
	self activitys do:[ :each|
		each bindComponents: self.
	].
	self actors do:[ :each |
		each bindComponents: self.
	].
	self resources do:[ :each |
		each bindComponents: self.
	].
	self materials do:[ :each |
		each bindComponents: self.
	].
	self products do:[ :each |
		each bindComponents: self.
	].
	self macros do:[ :each |
		each bindComponents: self.
	].
	self splits do:[ :each |
		each bindComponents: self.
	].
	self joins do:[ :each |
		each bindComponents: self.
	].
	self ends do:[ :each |
		each bindComponents: self.
	].
```

METHOD cloneExecutionFlow

```
"Copies the workflow components (activities, splits, joins, macros etc) as new
objects into a new MWfMWorkflow object."

	| newWorkflow |

	newWorkflow := self dcopy.
	newWorkflow bindComponents: nil.
```

```
	^newWorkflow.


"----------------------------------------------------------------------------------------------------------"
Smalltalk defineClass: #MWfMActivity
	superclass: #{Smalltalk.MWfMWorkflowComponent}
	indexedType: #none
	private: false
	instanceVariableNames: 'time umt previous next actors resources
materials products workflow allocatedActors allocatedResources
originalInitialTime realInitialTime '
	classInstanceVariableNames: ''
	imports: ''
	category: 'CyclopsMWfM_Workflow_Data_Structure'
```

METHOD bindComponents: parent

```
	| reference |
	"BIND PREVIOUS"
	reference := self previous.
	(reference isKindOf: MWfMWorkflowComponent) ifTrue: [reference :=
reference id.].

	self previous: (parent activitys at: (reference) ifAbsent: []).
	(self previous = nil) ifTrue: [
	self previous: (parent joins at: (reference) ifAbsent: []).
	(self previous = nil) ifTrue: [
	self previous: (parent products at: (reference) ifAbsent: []).
	(self previous = nil) ifTrue: [
	self previous: (parent splits at: (reference) ifAbsent: []).
	(self previous = nil) ifTrue: [
	self previous: (parent macros at: (reference) ifAbsent: []).
	(self previous = nil) ifTrue: [
	self previous: (parent starts at: (reference) ifAbsent: []).
	]]]]].

	"BIND NEXT"
	reference := self next.
	(reference isKindOf: MWfMWorkflowComponent) ifTrue: [reference :=
reference id.].

	self next: (parent activitys at: (reference) ifAbsent: []).
	(self next = nil) ifTrue: [
	self next: (parent joins at: (reference) ifAbsent: []).
	(self next = nil) ifTrue: [
	self next: (parent products at: (reference) ifAbsent: []).
```

```
	(self next = nil) ifTrue: [
	self next: (parent splits at: (reference) ifAbsent: []).
	(self next = nil) ifTrue: [
	self next: (parent macros at: (reference) ifAbsent: []).
	(self next = nil) ifTrue: [
	self next: (parent ends at: (reference) ifAbsent: []).
	]]]]].

	"BIND ACTORS"
	1 to: self actors size do:[ :count|
		reference := self actors at: count.
		(reference isKindOf: MWfMWorkflowComponent) ifTrue: [reference
:= reference id.].
		self actors at: count put: (parent actors at: reference).
	].

	"BIND RESOURCES"
	1 to: self resources size do:[ :count|
		reference := self resources at: count.
		(reference isKindOf: MWfMWorkflowComponent) ifTrue: [reference
:= reference id.].
		self resources at: count put: (parent resources at: reference).
	].

	"BIND MATERIALS"
	1 to: self materials size do:[ :count|
		reference := self materials at: count.
		(reference isKindOf: MWfMWorkflowComponent) ifTrue: [reference
:= reference id.].
		self materials at: count put: (parent materials at: reference).
	].

	"BIND PRODUCTS"
	1 to: self products size do:[ :count|
		reference := self products at: count.
		(reference isKindOf: MWfMWorkflowComponent) ifTrue: [reference
:= reference id.].
		self products at: count put: (parent products at: reference).
	].

	(parent isKindOf: MWfMMacro) ifTrue:[
		workflow := parent workflow
	] ifFalse:[
		workflow := parent
	].
	parentComponent :=  parent.
```

```
METHOD getDuration

        | duration timeAsInteger|

        [       (umt = 'hours') ifTrue:[
                        timeAsInteger := (((time asNumber) * 60) *60).
                ].
                (umt = 'minutes') ifTrue:[
                        timeAsInteger := (time asNumber)*60.
                ].
                duration:=Time fromSeconds: timeAsInteger.
                ^duration.
        ] on: Exception
        do: [:ex| ^'err - ',ex. "return a error processing."].

“-----------------------------------------------------------------------------------------------------------“
Smalltalk defineClass: #MWfMActor
        superclass: #{Smalltalk.MWfMWorkflowComponent}
        indexedType: #none
        private: false
        instanceVariableNames: 'kind skills activities '
        classInstanceVariableNames: ''
        imports: ''
        category: 'CyclopsMWfM_Workflow_Data_Structure'

METHOD bindComponents: parent
        | reference |
        "BIND ACTIVITIES"
        1 to: self activitys size do:[ :count|
                reference := self activitys at: count.
                (reference isKindOf: MWfMWorkflowComponent) ifTrue: [
                        reference := reference id.
                ].
                self activitys at: count put: (parent activitys at: reference).
        ].

“-----------------------------------------------------------------------------------------------------------“
Smalltalk defineClass: #MWfMEnd
        superclass: #{Smalltalk.MWfMWorkflowComponent}
        indexedType: #none
        private: false
        instanceVariableNames: 'type previous '
        classInstanceVariableNames: ''
```

```
	imports: ''
	category: 'CyclopsMWfM_Workflow_Data_Structure'
```

METHOD bindComponents: parent

```
	| reference |
	"BIND PREVIOUS"
	reference := self previous.
	(reference isKindOf: MWfMWorkflowComponent) ifTrue: [reference :=
reference id.].
	self previous: (parent activitys at: (reference) ifAbsent: []).
	(self previous = nil) ifTrue: [
	self previous: (parent joins at: (reference) ifAbsent: []).
	(self previous = nil) ifTrue: [
	self previous: (parent products at: (reference) ifAbsent: []).
	(self previous = nil) ifTrue: [
	self previous: (parent macros at: (reference) ifAbsent: []).
	]]].

	parentComponent := parent.
```

“---“

```
Smalltalk defineClass: #MWfMJoin
	superclass: #{Smalltalk.MWfMWorkflowComponent}
	indexedType: #none
	private: false
	instanceVariableNames: 'inputs next token '
	classInstanceVariableNames: ''
	imports: ''
	category: 'CyclopsMWfM_Workflow_Data_Structure'
```

METHOD bindComponents: parent

```
	| reference |
	"BIND NEXT"
	reference := self next.
	(reference isKindOf: MWfMWorkflowComponent) ifTrue: [reference :=
reference id.].
	self next: (parent activitys at: (reference) ifAbsent: []).
	(self next = nil) ifTrue: [
	self next: (parent joins at: (reference) ifAbsent: []).
	(self next = nil) ifTrue: [
	self next: (parent products at: (reference) ifAbsent: []).
	(self next = nil) ifTrue: [
	self next: (parent splits at: (reference) ifAbsent: []).
	(self next = nil) ifTrue: [
```

```
        self next: (parent macros at: (reference) ifAbsent: []).
        (self next = nil) ifTrue: [
        self next: (parent ends at: (reference) ifAbsent: []).
        ]]]]].

        "BIND INPUTS"
        1 to: self inputs size do:[ :count|
                        reference := self inputs at: count.
                        (reference isKindOf: MWfMWorkflowComponent) ifTrue:
[reference := reference id.].
                        self inputs at: count put: (parent activitys at: (reference)
ifAbsent: []).
                        ((self inputs at: count) = nil) ifTrue:[
                        self inputs at: count put: (parent joins at: (reference)
ifAbsent: []).
                        ]
        ].

        parentComponent := parent.

"-------------------------------------------------------------------------------------------------"

Smalltalk defineClass: #MWfMMacro
        superclass: #{Smalltalk.MWfMWorkflow}
        indexedType: #none
        private: false
        instanceVariableNames: 'previous next workflow '
        classInstanceVariableNames: ''
        imports: ''
        category: 'CyclopsMWfM_Workflow_Data_Structure'
```

METHOD bindComponents: parent

```
        | reference workflowReference |

        "SET WORKFLOW"
        workflowReference := parent.
        (workflowReference isKindOf: MWfMMacro) ifTrue:[
                workflowReference := parent workflow.
        ].
        workflow := workflowReference.
        parentComponent := parent.

        "BIND PREVIOUS"
        reference := self previous.
        (reference isKindOf: MWfMWorkflowComponent) ifTrue: [reference :=
reference id.].
```

```
	self previous: (parent activitys at: (reference) ifAbsent: []).
	(self previous = nil) ifTrue: [
	self previous: (parent joins at: (reference) ifAbsent: []).
	(self previous = nil) ifTrue: [
	self previous: (parent products at: (reference) ifAbsent: []).
	(self previous = nil) ifTrue: [
	self previous: (parent splits at: (reference) ifAbsent: []).
	(self previous = nil) ifTrue: [
	self previous: (parent macros at: (reference) ifAbsent: []).
	(self previous = nil) ifTrue: [
	self previous: (parent starts at: (reference) ifAbsent: []).
	]]]]].

	"BIND NEXT"
	reference := self next.
	(reference isKindOf: MWfMWorkflowComponent) ifTrue: [reference :=
reference id.].
	self next: (parent activitys at: (reference) ifAbsent: []).
	(self next = nil) ifTrue: [
	self next: (parent joins at: (reference) ifAbsent: []).
	(self next = nil) ifTrue: [
	self next: (parent products at: (reference) ifAbsent: []).
	(self next = nil) ifTrue: [
	self next: (parent splits at: (reference) ifAbsent: []).
	(self next = nil) ifTrue: [
	self next: (parent macros at: (reference) ifAbsent: []).
	(self next = nil) ifTrue: [
	self next: (parent ends at: (reference) ifAbsent: []).
	]]]]].


	self starts do:[ :each|
		each bindComponents: self.
	].
	self activitys do:[ :each|
		each bindComponents: self.
	].
	self actors do:[ :each |
		each bindComponents: self.
	].
	self resources do:[ :each |
		each bindComponents: self.
	].
	self materials do:[ :each |
		each bindComponents: self.
	].
```

```
	self products do:[ :each |
		each bindComponents: self.
	].
	self macros do:[ :each |
		each bindComponents: self.
	].
	self splits do:[ :each |
		each bindComponents: self.
	].
	self joins do:[ :each |
		each bindComponents: self.
	].
	self ends do:[ :each |
		each bindComponents: self.
	].

“----------------------------------------------------------------------------------------------------“

Smalltalk defineClass: #MWfMResource
	superclass: #{Smalltalk.MWfMWorkflowComponent}
	indexedType: #none
	private: false
	instanceVariableNames: 'type characteristics activities '
	classInstanceVariableNames: ''
	imports: ''
	category: 'CyclopsMWfM_Workflow_Data_Structure'
```

METHOD bindComponents: parent

```
	| reference |
	"BIND ACTIVITIES"
	1 to: self activitys size do:[ :count|
		reference := self activitys at: count.
		(reference isKindOf: MWfMWorkflowComponent) ifTrue: [reference
:= reference id.].
		self activitys at: count put: (parent activitys at: reference).
	].

	parentComponent := parent.

“----------------------------------------------------------------------------------------------------“
Smalltalk defineClass: #MWfMSplit
	superclass: #{Smalltalk.MWfMWorkflowComponent}
	indexedType: #none
	private: false
	instanceVariableNames: 'outputs previous token '
	classInstanceVariableNames: ''
```

```
        imports: ''
        category: 'CyclopsMWfM_Workflow_Data_Structure'
```

METHOD bindComponents: parent

```
        | reference |
        "BIND PREVIOUS"
        reference := self previous.
        (reference isKindOf: MWfMWorkflowComponent) ifTrue: [reference :=
reference id.].
        self previous: (parent activitys at: (reference) ifAbsent: []).
        (self previous = nil) ifTrue: [
        self previous: (parent joins at: (reference) ifAbsent: []).
        (self previous = nil) ifTrue: [
        self previous: (parent products at: (reference) ifAbsent: []).
        (self previous = nil) ifTrue: [
        self previous: (parent splits at: (reference) ifAbsent: []).
        (self previous = nil) ifTrue: [
        self previous: (parent macros at: (reference) ifAbsent: []).
        (self previous = nil) ifTrue: [
        self previous: (parent ends at: (reference) ifAbsent: []).
        (self previous = nil) ifTrue: [
        self previous: (parent starts at: (reference) ifAbsent: []).
        ]]]]]].

        "BIND OUTPUTS"
        1 to: self outputs size do:[ :count|
                reference := self outputs at: count.
                (reference isKindOf: MWfMWorkflowComponent) ifTrue: [reference
:= reference id.].
                self outputs at: count put: (parent activitys at: (reference) ifAbsent:
[]).
                ((self outputs at: count) = nil) ifTrue:[
                self outputs at: count put: (parent joins at: (reference) ifAbsent: []).
                ((self outputs at: count) = nil) ifTrue:[
                self outputs at: count put: (parent splits at: (reference) ifAbsent: []).
                ((self outputs at: count) = nil) ifTrue:[
                self outputs at: count put: (parent starts at: (reference) ifAbsent: []).
                ((self outputs at: count) = nil) ifTrue:[
                self outputs at: count put: (parent macros at: (reference) ifAbsent:
[]).
                ]]]].
        ].

        parentComponent := parent.
"----------------------------------------------------------------------------------------------------"
Smalltalk defineClass: #MWfMStart
```

```
        superclass: #{Smalltalk.MWfMWorkflowComponent}
        indexedType: #none
        private: false
        instanceVariableNames: 'type next '
        classInstanceVariableNames: ''
        imports: ''
        category: 'CyclopsMWfM_Workflow_Data_Structure'
```

METHOD bindComponents: parent

```
        | reference |
        "BIND NEXT"
        reference := self next.
        (reference isKindOf: MWfMWorkflowComponent) ifTrue: [reference :=
reference id.].
        self next: (parent activitys at: (reference) ifAbsent: []).
        (self next = nil) ifTrue: [
        self next: (parent joins at: (reference) ifAbsent: []).
        (self next = nil) ifTrue: [
        self next: (parent products at: (reference) ifAbsent: []).
        (self next = nil) ifTrue: [
        self next: (parent splits at: (reference) ifAbsent: []).
        (self next = nil) ifTrue: [
        self next: (parent macros at: (reference) ifAbsent: []).
        (self next = nil) ifTrue: [
        self next: (parent ends at: (reference) ifAbsent: []).
        ]]]]].

        parentComponent := parent.

“-----------------------------------------------------------------------------------------------------“
Smalltalk defineClass: #MWfMWorkflowLoader
        superclass: #{Core.Object}
        indexedType: #none
        private: false
        instanceVariableNames: 'workflow xmlFile '
        classInstanceVariableNames: ''
        imports: '
                        XML.XMLParser
                        XML.Element
                        '
        category: 'CyclopsMWfM_Workflow_Data_Structure'
```

METHOD loadComponents

```
        | parser workflowRoot |

        parser := XML.XMLParser new.
```

```
	parser validate: false.
	"Step 1: get the root to represent the workflow"
	workflowRoot := (parser parse: xmlFile readStream) root. "workflowRoot :=
(parser parse: xmlFile asFilename) root."
	"Step 2: Scan each node recursively and instanciate the respective
WorkflowComponent object (regardless references to not yet created objects)"
	self loadComponent: workflowRoot parentObject: nil.
	"Step 3: Set the references to the objects that were newly created"
	workflow bindComponents: nil.

	^workflow.
```

METHOD loadComponent: aXMLElement parentObject: parent

```
	"loads a WorkflowComponent into the respective Dictionary, without
referencing domain-specific objects"

	| aComponent componentElements element componentTagName |

	componentTagName := (aXMLElement tag type asLowercase).
	componentTagName at: 1 put: (componentTagName at: 1) asUppercase.

	(aXMLElement parent isKindOf: XML.Document) ifTrue:[ "Must create the
root (workflow) Instance"
		aComponent := MWfMWorkflow new.
		workflow  := aComponent.
		workflow xmlFile: self xmlFile.
	].

	componentElements :=  aXMLElement elements.
	(componentElements size <= 1) ifTrue:[
		(componentElements size = 1) ifTrue:[ "The Element is a leaf
element"
			| leafTagValue leafTagName|
			leafTagValue := (aXMLElement elements at:1) text.
			leafTagName := componentTagName asLowercase.
			(aXMLElement parent parent isKindOf: XML.Document)
ifTrue:[ "It´s a Root-direct leaf"
				parent perform: ((leafTagName, ':') asSymbol) with:
leafTagValue.
			]
			ifFalse:[ "It´s any deeper leaf"
				| parentTagName |
				parentTagName := aXMLElement parent tag type
asLowercase.
				(parentTagName findString: 's' startingAt:
(parentTagName size)) = 0 ifTrue:[ "It´s a component attribute"
```

```
                                        parent perform: ((leafTagName, ':') asSymbol)
with: leafTagValue.
                                ]
                                ifFalse:[ "The parent represents a set and the
component is a reference to another component: Use the method
add<Component>"
                                        (parent perform: ((leafTagName, 's')
asSymbol)) perform: ('add:' asSymbol) with: leafTagValue.
                                ]
                        ]
                ]
        ]
        ifFalse:[ "The component has children"
                | addMethodName |

                (aComponent = nil) ifFalse:[ ""
                        1 to: componentElements size do: [:count |
                                element := componentElements at: count.
                                (element isKindOf: XML.Element) ifTrue:[
                                        self loadComponent: element parentObject:
aComponent.
                                ]
                        ]
                ]
                ifTrue:[ "component object not yet created"
                        ((aXMLElement parent tag type asLowercase = 'workflow')
or: [ aXMLElement parent tag type asLowercase = 'macro' ]) ifTrue:[ "Component
is an Object: Let´s create it"
                                aComponent := (Object readFrom:
('MWfM',componentTagName)) new.
                                1 to: componentElements size do: [:count |
                                        element := componentElements at: count.
                                        (element isKindOf: XML.Element) ifTrue:[
                                        self loadComponent: element parentObject:
aComponent.
                                        ]
                                ].
                                "Add it into the respective dictionary"
                                addMethodName :=        componentTagName.

                                addMethodName := 'add', addMethodName, ':'.
                                parent perform: addMethodName asSymbol with:
aComponent.
                        ]
                        ifFalse:[ "The component represents a List: Just drill-down"
                                1 to: componentElements size do: [:count |
```

```
                                        element := componentElements at: count.
                                        (element isKindOf: XML.Element) ifTrue:[
                                                self loadComponent: element
parentObject: parent.
                                        ]
                                ]
                        ]
                ]
        ].
```

"---"

```
Smalltalk defineClass: #MWfMInstancedWorkflow
        superclass: #{Core.Object}
        indexedType: #none
        private: false
        instanceVariableNames: 'id workflow patient initialTime '
        classInstanceVariableNames: ''
        imports: ''
        category: 'CyclopsMWfM_Workflow_Data_Manipulation'
```

METHOD createNewWorkflow: aWorkflow

```
"Answers a workflow clone (a new MWfMWorkflow object with the same data
values)"
        | newWorkflow |

        newWorkflow := aWorkflow cloneExecutionFlow.
        ^newWorkflow.
```

"---"

```
Smalltalk defineClass: #MWfMSchedule
        superclass: #{Core.Object}
        indexedType: #none
        private: false
        instanceVariableNames: 'appointments appointmentRanges '
        classInstanceVariableNames: ''
        imports: ''
        category: 'CyclopsMWfM_Workflow_Scheduling'
```

METHOD allAppointmentRanges

```
        "Answers a List containing all the appointment ranges"
        appointments isEmpty ifTrue:[
                ^List new.
        ].
        ^self allAppointmentRangesStartingFrom: appointments first.
```

METHOD allAppointmentRangesForActivity: anActivity
"Answers a List containing all the appointment ranges that match a specific Activity"
| ranges |
ranges := self allAppointmentRanges select:[:range| (range activity = anActivity)].

^ranges asSortedCollection:[:each1 :each2 | each1 initialTime < each2 initialTime].

METHOD allAppointmentRangesStartingFrom: anInitialTime
"Answers a List containing all the appointment ranges after a given initial time"
| ranges |
ranges := appointmentRanges select:[:range |
((range initialTime >= anInitialTime) |
((range initialTime < anInitialTime) & (range finishTime >= anInitialTime))
)
].

^ranges asSortedCollection:[:each1 :each2 | each1 initialTime < each2 initialTime].

METHOD appointmentsBetween: anInitialTime and: aFinishTime

^appointments select: [:nextElement |
((nextElement asTimestamp >= anInitialTime) & (nextElement asTimestamp <= aFinishTime)).

METHOD appointmentsInDate: aDate

^appointments select: [:nextElement |
(nextElement asDate = aDate).
].

METHOD findFirstFit: aTime startingAt: aTimestamp
"Answers the first available time interval that is equal or grater than the duration specified in aTime, starting from aTimestamp"

| current next durationAsSeconds ranges |

ranges := self allAppointmentRangesStartingFrom: aTimestamp.
(ranges isEmpty) ifTrue:[^aTimestamp].

```
	current := ranges first.
	durationAsSeconds := aTime asSeconds.
	(((current initialTime asSeconds) - (aTimestamp asSeconds)) >=
durationAsSeconds) ifTrue:[^aTimestamp].
" tratar aqui o caso do firstFit estar antes do primeiro range   (aTimestamp <
(current initialTime)) ifTrue:[^aTimestamp].        "

	next := ranges detect:[:range | (range initialTime) > (current finishTime)]
ifNone:[nil].
	(next isNil) ifTrue: [^(current finishTime addSeconds: 1)].

	[((next initialTime asSeconds) - (current finishTime asSeconds)) <
durationAsSeconds] whileTrue:[
		current := next.
		next := ranges detect:[:range | (range initialTime) > (next
finishTime)] ifNone:[nil].
		(next isNil) ifTrue: [^(current finishTime addSeconds: 1)].
	].
	^(current finishTime addSeconds: 1).
```

METHOD firstAppointmentStartingFrom: aTimestamp
"Answers the next Appointment starting from aTimestamp"

```
	^appointments detect: [ :next | ((next asTimestamp) >= aTimestamp)]
ifNone: [nil].
```

METHOD isAvailable: initialTime until: finishTime
"Tests if the given time interval is available in schedule"

```
	|durationInSeconds firstFit|
	durationInSeconds := ((finishTime asSeconds) - (initialTime asSeconds) +
1).
	durationInSeconds := durationInSeconds - (durationInSeconds \\ 60).
	(durationInSeconds < 0) ifTrue:[
		Error raiseSignal: 'Initial time cannot be greater than finish time'
	].
	firstFit := self findFirstFit: (Time fromSeconds: durationInSeconds)
startingAt: initialTime.
	^(firstFit = initialTime) | (firstFit = nil).
```

METHOD isAvailable: initialTime withDuration: aTime
"Tests if the given time interval is available in schedule"

```
	^self isAvailable: initialTime until: ( initialTime addSeconds: ((aTime
asSeconds) - (aTime seconds) - 1) ).
```

```
“-----------------------------------------------------------------------------------------------------“
Smalltalk defineClass: #MWfMScheduler
	superclass: #{Core.Object}
	indexedType: #none
	private: false
	instanceVariableNames: 'schedule '
	classInstanceVariableNames: ''
	imports: ''
	category: 'CyclopsMWfM_Workflow_Scheduling'
```

METHOD schedule: initialTime until: finishTime forActivity: anActivity
"Schedules something between initialTime and finishTime"

```
	|difference newAppointmentAsTimestamp newAppointment |

	(self isAvailable: initialTime until: finishTime) ifFalse:[
		Error raiseSignal: 'There is no time available for scheduling'.
	].
	difference := ((finishTime asSeconds) - (initialTime asSeconds)) // 60.
	(difference < 0) ifTrue:[
		Error raiseSignal: 'Initial time cannot be greater than finish time'
	].
	0 to: (difference) do: [ :count |
		newAppointmentAsTimestamp := initialTime addSeconds: (count *
60).
		newAppointment := MWfMAppointment fromDate:
(newAppointmentAsTimestamp asDate) andTime:
(newAppointmentAsTimestamp asTime).
		newAppointment activity: anActivity.
		schedule appointments add: newAppointment.
	].
	schedule appointmentRanges add: (MWfMAppointmentRange new:
initialTime until: finishTime forActivity: anActivity).
	schedule appointments sort.
```

METHOD schedule: initialTime withDuration: aTime forActivity: anActivity
"Schedules from initialTime until initialTime + duration"

```
	self schedule: initialTime until: ( initialTime addSeconds: ((aTime
asSeconds) - (aTime seconds) - 1) ) forActivity: anActivity.
```

METHOD unschedule: anInitialTime until: aFinishTime
"Unschedules from initialTime until initialTime + duration"

```
	self deleteAppointments: anInitialTime until: aFinishTime.
	self fixAppointmentRanges: anInitialTime until: aFinishTime.
```

METHOD unschedule: initialTime withDuration: aTime
"Unschedules from initialTime until initialTime + duration"

self unschedule: initialTime until: (initialTime addSeconds: ((aTime asSeconds) - (aTime seconds) - 1)).

METHOD allAppointmentRanges
"Answers a List containing all the appointment ranges"

^schedule allAppointmentRanges.

METHOD allAppointmentRangesForActivity: anActivity
"Answers a List containing all the appointment ranges that match a specific Activity"

^schedule allAppointmentRangesForActivity: anActivity.

METHOD allAppointmentRangesStartingFrom: anInitialTime
"Answers a List containing all the appointment ranges after a given initial time"

^schedule allAppointmentRangesStartingFrom: anInitialTime.

METHOD appointmentsInDate: aDate

^ schedule appointmentsInDate: aDate.

METHOD findFirstFit: aTime
"Answers a Timestamp that represents the next free time in schedule, starting from
the current moment, considering the duration given in aTime"
^schedule findFirstFit: aTime.

METHOD findFirstFit: aTime startingAt: aTimestamp
"Answers a Timestamp that represents the next free time in schedule, starting from
aTimestamp, considering the duration given in aTime"
^schedule findFirstFit: aTime startingAt: aTimestamp.

METHOD firstAppointmentStartingFrom: aTimestamp
"Answers the next Appointment starting from aTimestamp"
^schedule firstAppointmentStartingFrom: aTimestamp.

METHOD isAvailable: initialTime until: finishTime
"Tests if the given time interval is available in schedule"

```
	^schedule isAvailable: initialTime until: finishTime.

METHOD isAvailable: initialTime withDuration: aTime
"Tests if the given time interval is available in schedule"

	^schedule isAvailable: initialTime withDuration: aTime.

METHOD deleteAppointments: anInitialTime until: aFinishTime
"deleção de appointements"
	| toDeleteList |
	toDeleteList := self schedule appointmentsBetween: anInitialTime and:
aFinishTime.
	toDeleteList do:[:appointment |
		self schedule appointments remove: appointment.
	].

METHOD fixAppointmentRanges: anInitialTime until: aFinishTime
	| toDeleteList appointmentToFixFinish appointmentToFixInitial
appointmentToFixInitialAndFinish newRange1 newRange2 |

"deleção de AppointmentRanges (que estão dentro do intervalo do intervalo
dado)"
	toDeleteList := schedule appointmentRanges select:[:range |
		( ((range initialTime) >= anInitialTime) & ((range finishTime) <=
aFinishTime) )
	].
	toDeleteList do:[:range |
		self schedule appointmentRanges remove: range.
	].
"correção de AppointmentRanges (que colidem com os limites do intervalo
dado)"
	appointmentToFixInitialAndFinish := schedule appointmentRanges
		detect:[:range | ( ((range initialTime) < anInitialTime) & ((range
finishTime) > aFinishTime) )]
		ifNone: [nil].
	(appointmentToFixInitialAndFinish isNil)
		ifFalse:[
			newRange1 := (MWfMAppointmentRange new:
(appointmentToFixInitialAndFinish initialTime)

	 until: (anInitialTime subtractSeconds: 60)

	 forActivity: (appointmentToFixInitialAndFinish activity)).
			newRange2 := (MWfMAppointmentRange new: (aFinishTime
addSeconds: 60)
```

```
          until: (appointmentToFixInitialAndFinish finishTime)

          forActivity: (appointmentToFixInitialAndFinish activity)).
                         schedule appointmentRanges add: newRange1.
                         schedule appointmentRanges add: newRange2.
                         schedule appointmentRanges remove:
appointmentToFixInitialAndFinish.
                    ].

          appointmentToFixFinish := schedule appointmentRanges
                    detect:[:range | ( ((range initialTime) < anInitialTime) & ((range
finishTime) >= anInitialTime) & ((range finishTime) <= aFinishTime) )]
                    ifNone: [nil].
          (appointmentToFixFinish isNil)
                    ifFalse:[ appointmentToFixFinish finishTime: (anInitialTime
subtractSeconds: 60)].

          appointmentToFixInitial := schedule appointmentRanges
                    detect:[:range | ( ((range initialTime) <= aFinishTime)  & ((range
initialTime) >= anInitialTime) & ((range finishTime) > aFinishTime) )]
                    ifNone: [nil].
          (appointmentToFixInitial isNil)
                    ifFalse:[ appointmentToFixInitial initialTime: (aFinishTime
addSeconds: 60)].
"----------------------------------------------------------------------------------------------------"

Smalltalk defineClass: #MWfMAppointment
          superclass: #{Core.Timestamp}
          indexedType: #none
          private: false
          instanceVariableNames: 'activity '
          classInstanceVariableNames: ''
          imports: ''
          category: 'CyclopsMWfM_Workflow_Scheduling'
```

METHOD asTimestamp

```
          ^Timestamp fromDate: (self asDate) andTime: (self asTime).

"----------------------------------------------------------------------------------------------------"
Smalltalk defineClass: #MWfMAppointmentRange
          superclass: #{Core.Object}
          indexedType: #none
          private: false
          instanceVariableNames: 'initialTime finishTime activity durationInMinutes '
          classInstanceVariableNames: ''
```

```
        imports: ''
        category: 'CyclopsMWfM_Workflow_Scheduling'

METHOD updateDuration
        ( (initialTime notNil) and: [finishTime notNil] )
        ifTrue:[
                durationInMinutes := ((finishTime asSeconds) - (initialTime
asSeconds) // 60) + 1.
        ].

"-----------------------------------------------------------------------------------------------------"
Smalltalk defineClass: #MWfMGoal
        superclass: #{Core.Object}
        indexedType: #none
        private: false
        instanceVariableNames: 'association parameters operator
dependenceGoals dependentGoals parentGoal derivedGoals executionContext '
        classInstanceVariableNames: ''
        imports: ''
        category: 'CyclopsMWfM_Inference_Engine_Goal'

METHOD setExecutionContextVariables

"
        sets the abstract execution context variables in subclasses.
        Example:
                self executionContext at: 'variableName1' put anObject1.
                self executionContext at: 'variableName2' put anObject2.

METHOD setOperator

"
        create the corresponding operator instance in subclasses.
        Example:
                self operator: (MWfMSampleOperator new: (self association)).
"
"

METHOD deriveGoal: aGoalClass goalCreationParams: aParameterList
dependences: aDependenceList
"cria uma nova meta, seta seus dependentes e a insere na máquina de
inferência"

        | newGoal |

        "instancia nova meta"
```

```
        newGoal := aGoalClass new: aParameterList association: (self
association).
        "define as dependencias da meta"
        newGoal setGoalDependences: aDependenceList.
        "auto-define-se a meta-pai da meta gerada"
        newGoal parentGoal: self.

        "coloca a meta para execuçao"
        self association inferenceMachine putNewProcess: 80 goal: newGoal.
        "adiciona a nova meta à lista de metas derivadas"
        self derivedGoals add: newGoal.
```

METHOD reduce

```
        | validationMsg |

        validationMsg := self operator validateGoalParams: (self parameters).
        (validationMsg = 'ok') ifFalse: [^validationMsg].
        ^self operator eval: self parameters.
```

METHOD setGoalDependences: aDependenceGoalList

```
        (aDependenceGoalList isNil) ifFalse:[
                aDependenceGoalList do: [: eachGoal |
                        self addDependenceGoal: eachGoal.
                        eachGoal addDependentGoal: self.
                ].
        ].
```

METHOD addDependenceGoal: aDependenceGoal

```
        dependenceGoals add: aDependenceGoal.
        aDependenceGoal addDependentGoal: self.
```

METHOD areThereDependenceGoals

```
        ^ ((self dependenceGoals isEmpty) not).
```

METHOD addDependentGoal: aDependentGoal

```
        self dependentGoals add: aDependentGoal.
        aDependentGoal addDependenceGoal: self.
```

METHOD areThereDependentGoals

```
        ^ ((self dependentGoals isEmpty) not).
```

METHOD removeDependentGoal: aGoal

```
	self dependentGoals remove: (aGoal) ifAbsent: [nil].
	aGoal dependenceGoals remove: (self) ifAbsent: [nil].
```

“---“

```
Smalltalk defineClass: #MWfMOperator
	superclass: #{Core.Object}
	indexedType: #none
	private: false
	instanceVariableNames: 'association goal '
	classInstanceVariableNames: ''
	imports: ''
	category: 'CyclopsMWfM_Inference_Engine_Operator'
```

METHOD eval: aParameters

"To be implemented in Subclasses."

METHOD validateGoalParams: goalParameters
"to be overridden in subclasses"
^'ok'

“---“

```
Smalltalk defineClass: #MWfMActivityConfigurationGoal
	superclass: #{Smalltalk.MWfMGoal}
	indexedType: #none
	private: false
	instanceVariableNames: ''
	classInstanceVariableNames: ''
	imports: ''
	category: 'CyclopsMWfM_Inference_Engine_Goal'
```

METHOD setOperator

```
	self operator: (MWfMActivityAllocationOperator new: (self association)).
	self operator goal: self.
```

“---“

```
Smalltalk defineClass: #MWfMActivityAllocationOperator
	superclass: #{Smalltalk.MWfMOperator}
	indexedType: #none
	private: false
	instanceVariableNames: ''
	classInstanceVariableNames: ''
	imports: ''
```

```
        category: 'CyclopsMWfM_Inference_Engine_Operator'

METHOD eval: aParameters

        |activity|

        activity:=(aParameters at:1).
        (activity isKindOf: MWfMActivity) ifFalse:[
                ^'err4 - Type of parameters not expected'.
        ].

        self allocateObjectsActivity: activity.

METHOD allocateObjectsActivity: anActivity
        | parentContext param duration originalInitialTime initialTime splitsStack
nextActivityTime |
        parentContext := self goal parentGoal executionContext.
        duration := anActivity getDuration.
        splitsStack := (parentContext at: 'currentSplitsStack').

        (anActivity previous isKindOf: MWfMSplit) ifTrue:[
                | currentSplit |

                ((splitsStack isNotEmpty) and: [((splitsStack last at: 'split') =
(anActivity previous))]) ifTrue:[
                        currentSplit := splitsStack last.
                        initialTime := currentSplit at: 'realInitialTime'.
                        originalInitialTime := currentSplit at: 'originalInitialTime'.
                ] ifFalse:[
                        initialTime := parentContext at: 'nextActivityRealInitialTime'.
                        originalInitialTime := parentContext at:
'nextActivityOriginalInitialTime'.
                        currentSplit := Dictionary new.
                        currentSplit at: 'split' put: (anActivity previous).
                        currentSplit at: 'originalInitialTime' put: originalInitialTime.
                        currentSplit at: 'realInitialTime' put: initialTime.
                        currentSplit at: 'joinInitialTime' put: originalInitialTime.
                        splitsStack add: currentSplit.
                ].
        ] ifFalse: [
                        initialTime := parentContext at: 'nextActivityRealInitialTime'.
                        originalInitialTime := parentContext at:
'nextActivityOriginalInitialTime'.
        ].

        (anActivity previous isKindOf: MWfMJoin) ifTrue:[
```

```
            originalInitialTime := (splitsStack last) at: 'joinInitialTime'.
            splitsStack remove: (splitsStack last).
        ].

        param:=OrderedCollection new.
        param add: initialTime.
        param add: duration.
        param add: anActivity.

        anActivity originalInitialTime: originalInitialTime.
        "defines the next Activity's OriginalInitialTime"
        nextActivityTime := (originalInitialTime addSeconds: (duration
asSeconds)).
        parentContext at: 'nextActivityOriginalInitialTime' put: nextActivityTime.
        (anActivity next isKindOf: MWfMJoin) ifTrue:[
            | joinInitialTime |
            joinInitialTime := (splitsStack last) at: 'joinInitialTime'.
            (nextActivityTime > joinInitialTime) ifTrue:[
                (splitsStack last) at: 'joinInitialTime' put: nextActivityTime.
            ].
        ].

        self goal deriveGoal: MWfMParticipantAllocationGoal
goalCreationParams: param dependences: nil.

“-----------------------------------------------------------------------------------------------“
Smalltalk defineClass: #MWfMWorkflowConfigurationGoal
        superclass: #{Smalltalk.MWfMGoal}
        indexedType: #none
        private: false
        instanceVariableNames: ''
        classInstanceVariableNames: ''
        imports: ''
        category: 'CyclopsMWfM_Inference_Engine_Goal'
```

METHOD setOperator

```
        self operator: (MWfMWorkflowAllocationOperator new: (self association)).
        self operator goal: self.
```

METHOD setExecutionContextVariables

```
"Adds 2 variables into the goal's execution context"
            self executionContext at: 'workflowInitialTime' put: nil.
            self executionContext at: 'nextActivityRealInitialTime' put: nil.
            self executionContext at: 'nextActivityOriginalInitialTime' put: nil.
```

```
                self executionContext at: 'currentSplitsStack' put: OrderedCollection
new.

"-----------------------------------------------------------------------------------------------------"
Smalltalk defineClass: #MWfMWorkflowConfigurationOperator
        superclass: #{Smalltalk.MWfMOperator}
        indexedType: #none
        private: false
        instanceVariableNames: ''
        classInstanceVariableNames: ''
        imports: ''
        category: 'CyclopsMWfM_Inference_Engine_Operator'
```

METHOD eval: aParameters

```
        (((aParameters at: 1) isKindOf: MWfMWorkflow) and: [(aParameters at: 2)
isKindOf: Timestamp]) ifTrue:[
                self allocateActivityParticipants: aParameters.
        ] ifFalse:[Dialog warn: 'Wrong parameters'].
```

METHOD do: aWorkflowComponent params: parametersStack

```
        (aWorkflowComponent isKindOf: MWfMStart) ifTrue:[
                self do: (aWorkflowComponent next) params: parametersStack.
        ]

        ifFalse:[
        (aWorkflowComponent isKindOf: MWfMActivity) ifTrue:[
                self allocateActivity: aWorkflowComponent.
                self do: (aWorkflowComponent next) params: parametersStack.
        ]

        ifFalse:[
        (aWorkflowComponent isKindOf: MWfMMacro) ifTrue:[
                self allocateMacro: aWorkflowComponent.
                self do: (aWorkflowComponent next) params: parametersStack.
        ]

        ifFalse:[
        (aWorkflowComponent isKindOf: MWfMSplit) ifTrue:[
                | joinInputList |
                joinInputList := OrderedCollection new.
                "adds the input list to the stack"
                parametersStack add: joinInputList.
                aWorkflowComponent outputs do:[ :output|
                        self do: output params: parametersStack.
```

```
		].
	]

	ifFalse:[
	(aWorkflowComponent isKindOf: MWfMJoin) ifTrue:[
		| joinInputList |

		"gets the join's input list from the top of the lists stack"
		joinInputList  := parametersStack last.
		joinInputList add: 0.
		((joinInputList size) = (aWorkflowComponent inputs size)) ifTrue:[
			"unstack the last-used list"
			parametersStack remove: (parametersStack last).
			self do: (aWorkflowComponent next) params:
parametersStack.
		].
	]]]]].

	(aWorkflowComponent isKindOf: MWfMProduct) ifTrue:[
		self do: (aWorkflowComponent next) params: parametersStack.
	].
```

METHOD allocateMacro: aWorkflowComponent

```
	self do: (aWorkflowComponent starts values at:1) params:
(OrderedCollection new).
```

METHOD allocateActivityParticipants: aParameter

```
	|workflow start initialTime|

	workflow := aParameter values at: 1.
	initialTime := aParameter values at: 2.
	start := workflow starts values at: 1.
	self goal executionContext at: 'workflowInitialTime' put: initialTime.
	self goal executionContext at: 'nextActivityRealInitialTime' put: initialTime.
	self goal executionContext at: 'nextActivityOriginalInitialTime' put:
initialTime.

	self do: start params: (OrderedCollection new).
```

METHOD allocateActivity: aWorkflowActivity

```
	| param |
	param:=OrderedCollection new.

	param add: aWorkflowActivity.
```

```
	"deriva a meta de alocação de atividade"
	self goal deriveGoal: MWfMActivityAllocationGoal goalCreationParams:
param dependences: nil.

"-----------------------------------------------------------------------------------------------------"
Smalltalk defineClass: #MWfMWorkflowDeallocationGoal
	superclass: #{Smalltalk.MWfMGoal}
	indexedType: #none
	private: false
	instanceVariableNames: ''
	classInstanceVariableNames: ''
	imports: ''
	category: 'CyclopsMWfM_Inference_Engine_Goal'
```

METHOD setOperator

```
	self operator: (MWfMWorkflowDeallocationOperator new: (self
association)).
	self operator goal: self.
```

METHOD eval: aParameters

```
"Este operador efetua desalocação no escopo de uma Atividade, Macro ou
Workflow, dependendo da classe à qual pertence o parametro 'scope'.
 A atividade de início da desalocação se dá pelo parâmetro 'startingActivity'.
 Caso o parâmetro 'deallocateForwards' seja true, então a desalocação se dará
da atividade inicial até o final do escopo.
 Se 'deallocateForwards' for false, a desalocação se limita à atividade inicial.
"
	| startingActivity deallocateForwards scope |

	startingActivity := aParameters at: 'startingActivity'.
	scope := aParameters at: 'scope'.
	deallocateForwards := aParameters at: 'deallocateForwards'.
"	self deallocateActivity: startingActivity."
	self performDeallocation: startingActivity scope: scope doForwards:
deallocateForwards.
```

METHOD validateGoalParams: goalParameters

```
	| startingActivity deallocateForwards scope |

	[ scope := goalParameters at: 'scope'.]
	on: Exception do: [^'missing parameter for scope'].

	[ startingActivity := goalParameters at: 'startingActivity'.]
	on: Exception do: [^'missing parameter for startingActivity'].
```

```
        [ deallocateForwards := goalParameters at: 'deallocateForwards'.]
        on: Exception do:[^'missing parameter for deallocateForwards'].

        ((startingActivity isKindOf: MWfMActivity) | (startingActivity isKindOf:
MWfMWorkflow))
                ifFalse: [^'parameter not recognized as MWfMActivity or
MWfMWorkflow'].
        (deallocateForwards isKindOf: Boolean)
                ifFalse: [^'parameter not recognized as MWfMInstancedWorkflow'].
        ^'ok'
```

METHOD deallocateComponentsStartingFrom: aStartingActivity inScope: aScope

```
        self deallocateComponent: aStartingActivity scope: aScope.
```

METHOD deallocateComponent: aWorkflowComponent scope: aScope

```
        (aWorkflowComponent isKindOf: MWfMActivity) ifTrue:[
                self deallocateActivity: aWorkflowComponent.
                self deallocateComponent: (aWorkflowComponent next) scope:
aScope.
        ].

        (aWorkflowComponent isKindOf: MWfMJoin) ifTrue:[
                self deallocateComponent: (aWorkflowComponent next) scope:
aScope.
        ].

        (aWorkflowComponent isKindOf: MWfMSplit) ifTrue:[
                aWorkflowComponent outputs do:[: each |
                        self deallocateComponent: each scope: aScope.
                ].
        ].

        (aWorkflowComponent isKindOf: MWfMMacro) ifTrue:[
                self deallocateComponent: ((aWorkflowComponent starts values
at:1) next) scope: aScope.
                self deallocateComponent: (aWorkflowComponent next) scope:
aScope.
        ] ifFalse: [
                (aWorkflowComponent isKindOf: MWfMWorkflow) ifTrue:[
                        self deallocateComponent: ((aWorkflowComponent starts
values at:1) next) scope: aScope.
                ].
```

```
	].

	(aWorkflowComponent isKindOf: MWfMEnd) ifTrue:[
		| parent |
		parent := aWorkflowComponent parentComponent.
		(parent ~= aScope) ifTrue:[
			self deallocateComponent: (parent next) scope: aScope.
		].
	].
```

METHOD deallocateActivity: anActivity

```
	| actor resource |

	anActivity allocatedActors keysDo:[: each |
		actor := ((anActivity allocatedActors) at: each).
		(actor notNil) ifTrue: [
			actor scheduler unschedule: (anActivity realInitialTime)
withDuration: (anActivity getDuration).
			(anActivity allocatedActors) at: each put: nil.
		].
	].

	anActivity allocatedResources keysDo:[: each |
		resource := ((anActivity allocatedResources) at: each).
		(resource notNil) ifTrue: [
			resource scheduler unschedule: (anActivity realInitialTime)
withDuration: (anActivity getDuration).
			(anActivity allocatedResources) at: each put: nil.
		].
	].

	anActivity realInitialTime: nil.
```

METHOD performDeallocation: aStartingActivity scope: aScope doForwards: aBoolean

```
	(aScope isKindOf: MWfMActivity) ifTrue: [
		^self deallocateActivity: aStartingActivity.
	] ifFalse:[
		self deallocateComponentsStartingFrom: aStartingActivity inScope:
aScope.
	].

"-----------------------------------------------------------------------------------------------"
Smalltalk defineClass: #MWfMWorkflowLoadingGoal
	superclass: #{Smalltalk.MWfMGoal}
```

```
        indexedType: #none
        private: false
        instanceVariableNames: ''
        classInstanceVariableNames: ''
        imports: ''
        category: 'CyclopsMWfM_Inference_Engine_Goal'
```

METHOD setOperator

```
        self operator: (MWfMWorkflowLoadingOperator new: association).
        self operator goal: self.
```

“---“

```
Smalltalk defineClass: #MWfMWorkflowLoadingOperator
        superclass: #{Smalltalk.MWfMOperator}
        indexedType: #none
        private: false
        instanceVariableNames: ''
        classInstanceVariableNames: ''
        imports: ''
        category: 'CyclopsMWfM_Inference_Engine_Operator'
```

METHOD eval: aParameters

```
        | workflowXML |

        workflowXML := aParameters first.
        (workflowXML isKindOf: XML.Element) ifTrue:[
                self loadWorkflow: workflowXML.
        ] ifFalse:[Dialog warn: 'Wrong parameters'].
```

METHOD loadWorkflow: aXMLFile

```
"This method loads a XMLFile containing the workflow definition, translates it to
an MWfMWorkflow and inserts it into the list of abstract workflows"
        | parameters workflow |

        "parse a XMLFile, instanciating an abstractWorkflow"
        workflow := ((MWfMWorkflowLoader new: (aXMLFile printString))
loadComponents).
        workflow id: (workflow id asNumber).

        "fill the parameters list to insert the newly created abstractWorkflow object
into the abstractWorkflows list"
        parameters:=OrderedCollection new.
        parameters add:      (workflow id).
        parameters add: (workflow).
```

```
	MWfMDBAPI insertNewElementOn: 'abstractWorkflow' parameters:
parameters.

"----------------------------------------------------------------------------------------------------"
Smalltalk defineClass: #MWfMParticipantAllocationGoal
	superclass: #{Smalltalk.MWfMGoal}
	indexedType: #none
	private: false
	instanceVariableNames: ''
	classInstanceVariableNames: ''
	imports: ''
	category: 'CyclopsMWfM_Inference_Engine_Goal'
```

METHOD setOperator

```
	self operator: (MWfMParticipantAllocationOperator new: (self
association)).
	self operator goal: self.


"----------------------------------------------------------------------------------------------------"
Smalltalk defineClass: #MWfMParticipantAllocationOperator
	superclass: #{Smalltalk.MWfMOperator}
	indexedType: #none
	private: false
	instanceVariableNames: ''
	classInstanceVariableNames: ''
	imports: ''
	category: 'CyclopsMWfM_Inference_Engine_Operator'
```

METHOD eval: aParameters

```
	| initialTime duration activity parentContext splitsStack
workflowGoalContext nextActivityTime mustAllocate |

"	initialTime:=(aParameters at:1)."
	duration:=(aParameters at:2).
	activity:=(aParameters at:3).

	parentContext := (self goal parentGoal) parentGoal executionContext.
	splitsStack := (parentContext at: 'currentSplitsStack').

	(activity previous isKindOf: MWfMSplit) ifTrue:[
		| currentSplit |

		((splitsStack isNotEmpty) and: [((splitsStack last at: 'split') = (activity
previous))]) ifTrue:[
```

```
				currentSplit := splitsStack last.
				initialTime := currentSplit at: 'realInitialTime'.
			] ifFalse:[
				initialTime := parentContext at: 'nextActivityRealInitialTime'.
				currentSplit := Dictionary new.
				currentSplit at: 'split' put: (activity previous).
				currentSplit at: 'realInitialTime' put: initialTime.
				currentSplit at: 'joinInitialTime' put: initialTime.
				splitsStack add: currentSplit.
			].
		] ifFalse: [
				initialTime := parentContext at: 'nextActivityRealInitialTime'.
		].

		(activity previous isKindOf: MWfMJoin) ifTrue:[
			initialTime := (splitsStack last) at: 'joinInitialTime'.
			splitsStack remove: (splitsStack last).
		].
		mustAllocate := (activity realInitialTime isNil).
"		mustAllocate := ((activity allocatedActors keys detect: [: each | (activity
allocatedActors at: each) isNil ]

ifNone: [nil]) notNil) or: [activity allocatedActors isEmpty]."
		mustAllocate ifTrue:[ "Existe algum recurso ou ator desalocado"
			self initAllocationSearch: initialTime duration: duration activity:
activity.
		] ifFalse: [
			"gets the MWfMPreAllocationWorkflowGoal´s execution context
and defines the next Activity's initialTime"
			workflowGoalContext := (self goal parentGoal) parentGoal
executionContext.
			nextActivityTime := (initialTime addSeconds: (duration
asSeconds)).
			workflowGoalContext at: 'nextActivityRealInitialTime' put:
nextActivityTime.
			(activity next isKindOf: MWfMJoin) ifTrue:[
				| joinInitialTime |
				splitsStack := (workflowGoalContext at: 'currentSplitsStack').
				joinInitialTime := (splitsStack last) at: 'joinInitialTime'.
				(nextActivityTime > joinInitialTime) ifTrue:[
					(splitsStack last) at: 'joinInitialTime' put:
nextActivityTime.
				].
			].
		].
```

METHOD allocateActor: aDBActor initialTime: anInitialTime duration: aDuration forActivity: anActivity replaceActor: anAbstractActor

```
"	anInitialTime = Timestamp object
	aDuration = Time object
"
	(anActivity allocatedActors isNil) ifTrue:[
		anActivity allocatedActors: (Dictionary new).
	].
	anActivity allocatedActors at: anAbstractActor put: aDBActor.
	aDBActor scheduler schedule: anInitialTime withDuration: aDuration
forActivity: anActivity.
```

METHOD allocateParticipants: aParticipantsDictionary initialTime: anInitialTime duration: aDuration forActivity: anActivity

```
"	anInitialTime = Timestamp object
	aDuration = Time object
"
| participant |
aParticipantsDictionary keys do:[:eachAbstractParticipant|
	participant := (aParticipantsDictionary at: eachAbstractParticipant).
	(participant isKindOf: MWfMDBActor) ifTrue:[
		self allocateActor: participant initialTime: anInitialTime duration:
aDuration forActivity: anActivity replaceActor: eachAbstractParticipant.
	].
	(participant isKindOf: MWfMDBResource) ifTrue:[
		self allocateResource: participant initialTime: anInitialTime duration:
aDuration forActivity: anActivity replaceResource: eachAbstractParticipant.
	].
].
```

METHOD allocateResource: aDBResource initialTime: anInitialTime duration: aDuration forActivity: anActivity replaceResource: anAbstractResource

```
"	anInitialTime = Timestamp object
	aDuration = Time object
"
	(anActivity allocatedResources isNil) ifTrue:[
		anActivity allocatedResources: (Dictionary new).
	].
	anActivity allocatedResources at: anAbstractResource put: aDBResource.
	aDBResource scheduler schedule: anInitialTime withDuration: aDuration
forActivity: anActivity.
```

METHOD continueAllocationSearch: anInitialTime duration: aDuration activity: anActivity relatedParticipants: activityRelatedParticipants

```
"	anInitialTime = Timestamp object; aDuration = Time object"
```

```
	| participantsToAllocate participant canContinue newInitialTime
workflowGoalContext nextActivityTime |

	participantsToAllocate := Dictionary new.
	canContinue := true.

	activityRelatedParticipants keys do:[:abstractParticipant| "Para cada Ator
ou recurso abstrato da Atividade:"
		"Tenta achar o Ator ou Recurso no Banco de dados mais capaz de
participar da atividade e que esteja disponível no momento"
		participant := self performParticipantAllocationSearch:
activityRelatedParticipants abstractParticipant: abstractParticipant initialTime:
anInitialTime duration: aDuration.
		canContinue := canContinue and: [(participant = nil) not].

		(canContinue) ifTrue:[ "Conseguiu achar um participante: Adiciona
à lista para alocação"
			participantsToAllocate at: abstractParticipant put: participant.
		] ifFalse:[ "Não há participantes aptos disponíveis: Encontra uma
nova hora inicial e faz nova tentativa de Alocação (METHOD recursivo)"
			newInitialTime := self
tryNewInitialTime:activityRelatedParticipants initialTime: (anInitialTime
addSeconds: 60) duration: aDuration.
			^self continueAllocationSearch: newInitialTime duration:
aDuration activity: anActivity relatedParticipants: activityRelatedParticipants.
		].
	].
	"Encontrou todos os Atores/Recursos necessários para a Atividade."
	self allocateParticipants: participantsToAllocate initialTime: anInitialTime
duration: aDuration forActivity: anActivity.
	anActivity realInitialTime: anInitialTime.

	"gets the MWfMPreAllocationWorkflowGoal´s execution context and
defines the next Activity's initialTime"
	workflowGoalContext := (self goal parentGoal) parentGoal
executionContext.
	nextActivityTime := (anInitialTime addSeconds: (aDuration asSeconds)).
	workflowGoalContext at: 'nextActivityRealInitialTime' put: nextActivityTime.
	(anActivity next isKindOf: MWfMJoin) ifTrue:[
		| splitsStack joinInitialTime |
		splitsStack := (workflowGoalContext at: 'currentSplitsStack').
		joinInitialTime := (splitsStack last) at: 'joinInitialTime'.
		(nextActivityTime > joinInitialTime) ifTrue:[
			(splitsStack last) at: 'joinInitialTime' put: nextActivityTime.
		].
	].
```

```
METHOD initAllocationSearch: anInitialTime duration: aDuration activity:
anActivity
"       anInitialTime = Timestamp object; aDuration = Time object"
        | activityRelatedParticipants |
        Dictionary new.
        "guarda todos os Atores/Recursos capazes de executar a atividade"
        activityRelatedParticipants := self findActivityRelatedParticipants:
anActivity.
        ^self continueAllocationSearch: anInitialTime duration: aDuration activity:
anActivity relatedParticipants: activityRelatedParticipants.

METHOD performActorAllocationSearch: relatedActors initialTime: anInitialTime
duration: aDuration
"       anInitialTime = Timestamp object; aDuration = Time object"

        | optimalActors bestActor availableRelatedActors |
                "Dos DBActors relacionados, são retornados apenas os que estão
disponíveis no momento da execução da atividade"
                availableRelatedActors := self getAvailableDBActors: relatedActors
initialTime: anInitialTime duration: aDuration.
                (availableRelatedActors isEmpty) ifTrue:[
                        ^nil.
                ].
                "Retorna apenas os Actors mais apropriados para a atividade
(Menor número de Skills)"
                optimalActors := self getOptimalDBActors: availableRelatedActors.
                bestActor := optimalActors first.
                ^bestActor.

METHOD performParticipantAllocationSearch: aRelatedParticipantsDictionary
abstractParticipant: anAbstractParticipant initialTime: anInitialTime duration:
aDuration
"       anInitialTime = Timestamp object; aDuration = Time object"

        | relatedParticipants |
        "retorna os Participants (Actors ou Resources) capazes de realizar a
atividade"
        relatedParticipants := aRelatedParticipantsDictionary at:
anAbstractParticipant.
        (relatedParticipants isEmpty) ifTrue:[
                ^nil.
        ].

        (anAbstractParticipant isKindOf: MWfMActor) ifTrue:[
```

```
		^self performActorAllocationSearch: relatedParticipants initialTime:
anInitialTime duration: aDuration.
	].
	(anAbstractParticipant isKindOf: MWfMResource) ifTrue:[
		^self performResourceAllocationSearch: relatedParticipants
initialTime: anInitialTime duration: aDuration.
	].
```

METHOD performResourceAllocationSearch: relatedResources initialTime: anInitialTime duration: aDuration

```
"	anInitialTime = Timestamp object; aDuration = Time object"

	| optimalResources bestResource availableRelatedResources |
		"Dos DBResources relacionados, são retornados apenas os que
estão disponíveis no momento da execução da atividade"
		availableRelatedResources := self getAvailableDBResources:
relatedResources initialTime: anInitialTime duration: aDuration.
		(availableRelatedResources isEmpty) ifTrue:[
			^nil.
		].
		"Retorna apenas os Resources mais apropriados para a atividade
(Menor número de Characteristics)"
		optimalResources := self getOptimalDBResources:
availableRelatedResources.
		bestResource := optimalResources first.
		^bestResource.
```

METHOD tryNewInitialTime: activityRelatedParticipants initialTime: anInitialTime duration: aTime

```
"Retorna o próximo horário provável para alocação.
- activityRelatedParticipants é um Dictionary"
	| timeList dbParticipants tempTimeList |

	timeList := List new.

	"Preenche uma lista com o próximo horário livre, a partir de 'anInitialTime',
de cada um dos Atores/Recursos relacionados à Atividade"
	activityRelatedParticipants keys do:[:abstractParticipant |
		tempTimeList := List new.
		dbParticipants := activityRelatedParticipants at: abstractParticipant .
		(dbParticipants) do: [:participant |
			tempTimeList add: (participant scheduler findFirstFit: aTime
startingAt: anInitialTime).
		].
		tempTimeList sort.
		timeList add: tempTimeList first.
```

```
        ].
        timeList sort.
        ^timeList last. "Obtém o próximo horário"
```

METHOD findActivityRelatedParticipants: anActivity
"retorna um Dictionary contendo todos os DBActors e DBResources com o type e as characteristics necessárias para alocação na atividade"

```
        | participantsDict |

        participantsDict := Dictionary new.
        anActivity actors do:[:actor |
                participantsDict at: actor put: (self findRelatedDBActors: actor).
        ].
        anActivity resources do:[:resource |
                participantsDict at: resource put: (self findRelatedDBResources:
resource).
        ].
        ^participantsDict.
```

METHOD findActorSkills: anActorIDCollection
"retorna uma lista de Skills, referentes a determinado Ator.
Caso anActorIDCollection seja vazia ou nil, a função retorna uma OrderedCollection vazia"

```
        | result |

        result := OrderedCollection new.

        anActorIDCollection do:[ :each|
                result add: (self findInClassDataByID: 'actorSkills' id: each
asNumber).
        ].
        ^result.
```

METHOD findActorSpecialty: anActorSpecialtyId

```
        ^self findInClassDataByID: 'actorSpecialties' id: anActorSpecialtyId.
```

METHOD findInClassDataByID: aClassData id: anId

```
        | param obj |

        [       ((anId notNil) and:[aClassData notNil]) ifTrue:[
                        param:=OrderedCollection new.
                        param add: anId.
                        obj:=MWfMDBAPI queryByIDs: aClassData parameters:
param.
                        ^obj.
```

```
			] ifFalse:[^'err2' "Parameters expected"].
		] on: GenericException do: [:ex| ^'err1 - ',ex.  "Exception occurence during
execution"].
```

METHOD findMaterials: aMaterials

```
		| obj type|

		aMaterials do:[:material|
			type:=self findMaterialsType: material type asNumber.
			obj:=self findObjectsInClassData: 'materials' object: type attribute:
'type'.
			(obj isNil) ifFalse:[
				^obj.
			].
		].
		^nil.
```

METHOD findObjectsInClassData: aClassData object: anObject attribute: anAttribute

```
		| param |

		[	(((aClassData notNil) and:[anObject notNil]) and:[anAttribute
notNil]) ifTrue:[
				param:=OrderedCollection new.
				param add: anAttribute.
				param add: anObject.
				^MWfMDBAPI findObjectsByAnyData: aClassData
parameters: param.
			] ifFalse:[^'err2' "Parameters expected"].
		] on: Exception do: [:ex| self halt. 'err1 - ',ex.  "Exception occurence during
execution"].
```

METHOD findRelatedDBActors: anActor

```
"retorna uma OrderedCollection contendo todos os DBActors com a specialty e
as skills necessárias para alocação na atividade"
		| actorSpecialty actorsSkills  specializedObjects hasAllNeededSkills
resultDBActors  |
		resultDBActors := OrderedCollection new.
		"encontra a specialty a qual os atores devem ter"
		actorSpecialty:=self findActorSpecialty: (anActor kind) asNumber.
		"encontra as skills as quais os atores devem ter"
		actorsSkills := self findActorSkills: (anActor skills asOrderedCollection).
		"filtra os atores com determinada specialty"
```

```
	specializedObjects:=self findObjectsInClassData: 'actors' object:
actorSpecialty attribute: 'specialties'.
	(specializedObjects isNil) ifTrue: [^resultDBActors.].

	"dos selecionados, filtra os atores que possuem todas as Skills
necessárias"
	specializedObjects do:[ :dbActor|
		hasAllNeededSkills := true.
		actorsSkills do:[ :skill |
			hasAllNeededSkills := hasAllNeededSkills & ( ((dbActor skills
detect: [: each | each = skill] ifNone: [nil]) isNil) not).
		].
		(hasAllNeededSkills) ifTrue: [	resultDBActors add: dbActor].
	].
	^resultDBActors.
```

METHOD findRelatedDBResources: aResource

```
"retorna uma OrderedCollection contendo todos os DBResources com o type e
as characteristics necessárias para alocação na atividade"
	| resourceType specializedObjects resultDBResources
resourceCharacteristics hasAllNeededCharacteristics |
	resultDBResources := OrderedCollection new.
	"encontra a type a qual os recursos devem ter"
	resourceType:=self findResourceType: (aResource type) asNumber.
	"encontra as characteristics as quais os recursos devem ter"
	resourceCharacteristics := self findResourceCharacteristics: (aResource
characteristics asOrderedCollection).
	"filtra os recursos com determinado type"
	specializedObjects:=self findObjectsInClassData: 'resources' object:
resourceType attribute: 'type'.
	(specializedObjects isNil) ifTrue: [^resultDBResources.].

	"dos selecionados, filtra os recursos que possuem todas as
Characteristics necessárias"
	specializedObjects do:[ :dbResource|
		hasAllNeededCharacteristics := true.
		resourceCharacteristics do:[ :characteristic |
			hasAllNeededCharacteristics :=
hasAllNeededCharacteristics & ( ((dbResource characteristics detect: [: each |
each = characteristic] ifNone: [nil]) isNil) not).
		].
		(hasAllNeededCharacteristics) ifTrue: [resultDBResources add:
dbResource].
	].
	^resultDBResources.
```

METHOD findResourceCharacteristics: aResourceIDCollection
"retorna uma lista de Characteristics, referentes a determinado Recurso.
Caso aResourceIDCollection seja vazia ou nil, a função retorna uma OrderedCollection vazia"

```
	| result |

	result := OrderedCollection new.

	aResourceIDCollection do:[ :each|
		result add: (self findInClassDataByID: 'resourceCharacteristics' id:
each asNumber).
	].
	^result.
```

METHOD findResourceType: aResourceTypeID

```
	^self findInClassDataByID: 'resourceTypes' id: aResourceTypeID.
```

METHOD getAvailableDBActors: aCollection initialTime: aInitialTime duration: aDuration
"retorna uma OrderedCollection contendo os DBActors com disponibilidade para serem alocados na atividade"
"aCollection - OrderedCollection of DBActors"

```
	| resultActors |

	resultActors := OrderedCollection new.
	aCollection do:[ :dbActor |
		(dbActor scheduler isAvailable: aInitialTime withDuration:
aDuration) ifTrue:[
			resultActors add: dbActor.
		].
	].

	^resultActors.
```

METHOD getAvailableDBResources: aCollection initialTime: aInitialTime duration: aDuration
"retorna uma OrderedCollection contendo os DBResources com disponibilidade para serem alocados na atividade"
"aCollection - OrderedCollection of DBResources"

```
	| resultResources |

	resultResources := OrderedCollection new.
	aCollection do:[ :dbResource |
```

```
		(dbResource scheduler isAvailable: aInitialTime withDuration: aDuration) ifTrue:[
			resultResources add: dbResource.
		].
	].

	^resultResources.
```

METHOD getOptimalDBActors: aCollection

```
"retorna uma OrderedCollection contendo todos os DBActors mais adequados (com o menor número de skills possível)
 para alocação na atividade, dada uma OrderedCollection de DBActors.
NOTA: aCollection deve ser um OrderedCollection"

| minSkillCount actualSkillCount |

"Encontra o menor número de skills entre os DBActors aptos"
	minSkillCount := 9999.
	aCollection do:[ :dbActor|
		actualSkillCount := dbActor skills size.
		(actualSkillCount < minSkillCount) ifTrue:[
			minSkillCount := actualSkillCount.
		].
	].

"retorna somente os DBActors com o menor número possível de skills"
	^aCollection select: [: dbActor|
		((dbActor skills size) = minSkillCount).
	].
```

METHOD getOptimalDBResources: aCollection

```
"retorna uma OrderedCollection contendo todos os DBResources mais adequados (com o menor número de skills possível)
 para alocação na atividade, dada uma OrderedCollection de DBResources.
NOTA: aCollection deve ser um OrderedCollection"

| minSkillCount actualSkillCount |

"Encontra o menor número de skills entre os DBResources aptos"
	minSkillCount := 9999.
	aCollection do:[ :dbResource|
		actualSkillCount := dbResource characteristics size.
		(actualSkillCount < minSkillCount) ifTrue:[
			minSkillCount := actualSkillCount.
		].
	].
```

```
"retorna somente os DBResources com o menor número possível de skills"
	^aCollection select: [: dbResource|
		((dbResource characteristics size) = minSkillCount).
	].

METHOD validateGoalParams: goalParamaters
	|initialTime duration activity |

	initialTime:=(goalParamaters at:1).
	duration:=(goalParamaters at:2).
	activity:=(goalParamaters at:3).

	(initialTime isKindOf: Timestamp) ifFalse:[
		^'err4 - Type of parameters expected is Timestamp' .
	].

	(duration isKindOf: Time) ifFalse:[
		^'err4 - Type of parameters expected is Time' .
	].

	(activity isKindOf: MWfMActivity) ifFalse:[
		^'err4 - Type of parameters expected is MWfMActivity' .
	].

	^'ok'
“-----------------------------------------------------------------------------------------------------“
Smalltalk defineClass: #MWfMScheduleDisplayFormatter
	superclass: #{Core.Object}
	indexedType: #none
	private: false
	instanceVariableNames: 'schedulableObject initialTime finishTime
formattedSketch '
	classInstanceVariableNames: ''
	imports: ''
	category: 'CyclopsMWfM_Client_Test'

METHOD initFormattedStrokesStartingFrom: anInitialTime until: aFinishTime
forSchedulableObject: aSchedulableObject

	| ranges initialTimeInMinutes y initialX finishX stroke appointmentsColor
rangeInitialTimeInMinutes |
	formattedSketch := MWfMSketch new.
	schedulableObject := aSchedulableObject.
	initialTime := anInitialTime.
	finishTime := aFinishTime.
```

```
	ranges := (schedulableObject scheduler
allAppointmentRangesStartingFrom: initialTime).
	ranges := ranges select: [:appointment | appointment finishTime <=
finishTime].
	initialTimeInMinutes := (anInitialTime asSeconds) // 60.

	appointmentsColor := nil.
	ranges do:[:range |
		(appointmentsColor = (ColorValue royalBlue))
			ifTrue: [appointmentsColor := (ColorValue paleGreen)]
			ifFalse: [appointmentsColor := (ColorValue royalBlue)].
		rangeInitialTimeInMinutes := (((range initialTime asSeconds) // 60)).
		"pega a linha (coordenada y) correspondente ao dia. cada linha
tem 30 pixels de altura"
		y := ((rangeInitialTimeInMinutes -  initialTimeInMinutes) quo: 1440)
* 30.
		"pega a coordenada x correspondente ao horário inicial do range"
		initialX := (rangeInitialTimeInMinutes -  initialTimeInMinutes) rem:
1440.
		finishX := initialX + (range durationInMinutes).
		stroke := MWfMSketchStroke new.
		stroke color: appointmentsColor.
		y to: (y + 30) do: [:currentY |
			(stroke lineSegments) add: initialX @ currentY.
			(stroke lineSegments) add: finishX @ currentY.
		].
		formattedSketch strokes add: stroke.
	].

	^formattedSketch
```

METHOD updateFormattedStrokes

```
	schedulableObject isNil ifTrue:[
		Error raiseSignal: 'schedulable Object cannot be nil'
	].
	initialTime isNil ifTrue:[
		Error raiseSignal: 'Initial Time must be defined'
	].
	finishTime isNil ifTrue:[
		Error raiseSignal: 'Finish Time must be defined'
	].
	^self initFormattedStrokesStartingFrom: initialTime until: finishTime
forSchedulableObject: schedulableObject.
```