Modelo de Sistema de Processamento de Fluxos de Trabalho Científico Utilizando Planejamento

Resumo. *Experimentos científicos produzem grande quantidade de informações que necessitam de processamento para uma posterior análise. Um cientista, que não da área da computação, nem sempre possui as habilidades para desenvolver seu próprio ambiente de testes, por isso a utilização de executores de fluxos de trabalhos científicos vêm sido largamente estudada. A principal vantagem ao utilizar um processador de fluxo de trabalho científico é a transparência oferecida para o cientista em relação a maneira com que os experimentos serão organizados, distribuídos e processados. O presente trabalho propõe um modelo para criação de um ambiente que seja capaz de processar esses fluxos de trabalho, com foco em um escalonamento inteligente utilizando para isso, técnicas para resolução de problemas de planejamento da área de inteligência artificial.*

1. Introdução

O ambiente científico produz *terabytes* de informações que precisam ser processadas e analisadas. A interação multidisciplinar entre a computação e outras áreas da ciência carece de ferramentas que ofereçam uma interface amigável e transparente. Desta forma, os cientistas não precisam se aprofundar em conhecimentos computacionais avançados, necessários para processar e extrair informações de suas pesquisas. Para que tais projetos científicos sejam executados, são criados fluxos de trabalho científicos, que dividem uma consulta maior em partes menores, permitindo uma distribuição estratégica do processamento desses fluxos, visando minimizar o tempo de execução [1].

A utilização de fluxos de trabalho científico é uma forma de encapsular a automatização de processos de negócios, compostos por dados que são transformados e transmitidos entre participantes, com uma certa regra procedimental, afim de obter um resultado de um experimento científico [2].

Uma das grandes dificuldades ao se trabalhar com fluxos de trabalhos científicos está no escalonamento das tarefas que o compõem. Os experimentos científicos necessitam de muito processamento computacional, por isso, normalmente utilizam um ambiente distribuído, que oferece recursos computacionais e técnicas de paralelismo que possibilitam tal processamento. Dessa forma, é necessário alguma estratégia para escalonar tarefas de múltiplos fluxos de trabalho científico.

O problema de escalonamento é um subproblema de planejamento, no qual já se sabe quais serão os elementos a serem trabalhados, necessita-se apenas de uma ordenação coordenada desses elementos. A alternativa proposta neste trabalho é a utilização de técnicas para resolução de problemas de planejamento da área da inteligência artificial com o objetivo de escalonar as tarefas dos fluxos de trabalho científico [3].

Um problema de planejamento pode ser definido como sequência de ações que quando executadas em um mundo que satisfaça um estado inicial descrito, vai atingir o objetivo. Ao mapear computacionalmente situações do mundo real, nota-se a complexidade

obtida por uma grande quantidade de escolhas, o que torna o problema intratável com algoritmos convencionais. Ao tratar problemas complexos torna-se necessário a utilização de meta informações específicas do domínio, que auxiliam na busca de uma solução para o problema de planejamento.

Com o objetivo de refinar o processo de escalonamento, propõe-se a utilização de informações armazenadas em um banco de metadados que servirão para alimentar a construção de um plano de execução, contendo os passos a serem executados para a obtenção do resultado.

Com o plano de execução traçado pelo planejador, torna-se necessária uma estrutura que possibilite interpretar o plano e disparar a execução das tarefas em um ambiente distribuído. Para isso utiliza-se um *gerenciador de execução* baseado no sistema *PeerUnit* [4], que originalmente foi concebido para executar testes em ambientes *peer to peer*. O *gerenciador de execução* faz com que os *nós* sejam capazes de executar as tarefas que lhe são designadas, respeitando o plano de execução anteriormente traçado.

No modelo proposto, um problema científico é mapeado em fluxos de trabalho científicos. Cada fluxo de trabalho é dividido em tarefas. Na definição dos fluxos, são estabelecidas possíveis dependências entre as tarefas, que serão analisadas posteriormente pelo planejador. Essas tarefas podem ser implementadas de diferentes maneiras, explorando ou não o paralelismo disponível, para que posteriormente possam ser ponderadas pelo planejador. Existe ainda um banco de metadados que armazena informações sobre execuções anteriores, bem como as características dos nós disponíveis para execução. O planejador utiliza as informações desse banco de metadados para criar um plano de execução que define qual tarefa será executada em qual nó. Com o plano de execução pronto, uma camada de execução é acionada e dispara as tarefas para seus respectivos nós. Após a execução de uma tarefa, cada nó envia informações sobre a utilização de seus recursos, que são armazenados e utilizados para refinar uma próxima execução.

Na seção 2 são apresentados os conceitos de fluxos de trabalho científico. Na seção 3 faz-se um breve levantamento sobre as técnicas para resolução de problemas de planejamento, bem como a motivação para a escolha da técnica e seu funcionamento. A execução do modelo está descrita na seção 4. O modelo para um sistema de processamento de fluxos de trabalho científico utilizando planejamento está descrito na seção 5, bem como um exemplo que ilustra o fluxo de execução do modelo. A seção 6 apresenta as considerações finais do artigo e os trabalhos futuros previstos.

2. Fluxo de Trabalho Científico

Um fluxo de trabalho científico é o encapsulamento de informações que pode ser processado automaticamente em um fluxo de execução, gerenciado por um sistema dito *sistema gerenciador de fluxos de trabalho científico*. Ele é composto por dados que são transformados e transmitidos entre os componentes do sistema, para atingir um objetivo. O processo de execução de um fluxo de trabalho científico pode ser dividido em etapas, são elas: *composição*, *orquestração*, *mapeamento* e *execução* [3].

O processo de *composição* de um fluxo de trabalho científico está relacionado a descrição das tarefas que cada fluxo deverá executar, bem como a relação de dependência entre elas. Normalmente sua representação é feita através de grafos dirigidos, no qual os

vértices representam as tarefas e as arestas suas dependências [3].

Existem linguagens que especificam um fluxo de trabalho científico, entre elas: *MoML* [5], *Scufl* [6], *BPEL* [7] e *DAX* [8]. Essas, variam de acordo com a semântica, tipo de aplicação, ambiente de execução ou representação *abstrato* ou *concreto*. Uma representação *abstrata* não descreve os detalhes de execução, diferente de uma representação *concreta*, que descreve. *MoML*, *Scufl* e *DAX*, são linguagens especificamente desenvolvidas para o processamento de fluxos de trabalho. *BPEL* é uma linguagem para especificar ações em processos de negócios dentro de serviços *web*, adaptada para execução de fluxos de trabalho. No modelo proposto indica-se a adaptação de uma linguagem descrita em XML, afim de aplicar inferências e parâmetros específicos das camadas de planejamento e execução.

O processo de automatização de tarefas requer uma coordenação, para que haja um controle sobre as dependências entre as tarefas de um fluxo de trabalho. O componente que efetua essa automatização é chamado de *orquestrador*. O componente de *mapeamento* tem como função a ligação entre tarefas e recursos. Dessa forma, uma tarefa que necessite de mais recursos é alocada em um nó com maior capacidade de processamento. A *execução* tem como função a alocação das tarefas nos nós disponíveis [3].

Os executores de fluxos de trabalho científico, em sua maioria, precisam da interferência direta do usuário, que ordena manualmente os fluxos de trabalho. No modelo proposto, um problema científico é mapeado em 1 ou mais fluxos de trabalhos. O planejador é responsável por analisar e organizar os fluxos de trabalho automaticamente, obtidos pela análise de execuções anteriores. Com isso é possível inferir critérios de otimização como por exemplo: tempo de execução, gasto de energia e minimização ou maximização de recursos computacionais.

As etapas de *composição*, *orquestração* e *mapeamento* são tratadas na fase de planejamento e a etapa de *execução* é feita pelo *gerenciador de execução*.

3. Planejamento

A computação oferece recursos para minimizar esforços ao resolver problemas do mundo real, tais problemas podem apresentar um alto nível de complexidade. Embora seja difícil representar algumas características tais como: representação temporal, eventos inesperados e principalmente como estará o mundo depois de determinada modificação. Problemas que envolvam tais características estão em uma classe de complexidade que são intratáveis com algoritmos convencionais [9]. Seguem algumas definições para o entendimento do mapeamento do problema de escalonar fluxos de trabalho científico em um problema de planejamento em inteligência artificial.

Um *estado* é definido por um conjunto de proposições que identificam as características de algum elemento no mundo descrito. Um *plano* é a sequencia de ações que levam do estado inicial dado a um estado objetivo desejado, e também é a solução de um problema de planejamento. Uma *ação* ou *operador* é um modificador de estado. Quando uma *ação* é executada algum efeito é aplicado no mundo mapeado. Uma ação é formada por:

- *lista de pré-condições*: são os requisitos necessários para que uma determinada ação seja executada;

- *lista de adições*: são os fatos a serem adicionados ao mundo mapeado, depois que a ação é executada;
- *lista de exclusões*: são os efeitos negativos, ou seja, o que será retirado da representação do estado atual, após a execução de uma ação.

A busca no espaço de estados de um problema de planejamento foi definido inicialmente por STRIPS (*STranford Research Institute Problem Solver*) [10], esse foi o precursor dos resolvedores dos problemas de planejamentos modernos, com essa abordagem definiu-se a forma com que as ações são representadas, e é mantida na maioria dos planejadores atuais.

As abordagens clássicas para resolução de problemas de planejamento executam uma busca no espaço de estados com objetivo de encontrar um caminho (que representará um conjunto de ações) do estado inicial até o estado objetivo. Esses planejadores são efetivos para resolução de problemas com domínios relativamente pequenos e estáticos [11, 12]. Entretanto, ao tentar tratar domínios dinâmicos e mais complexos não obtém bons resultados [9].

Os planejadores *hierárquicos* são uma evolução dos clássicos, e conseguem tratar uma gama de problemas com maior grau de complexidade pela inferência direta de informações que refinam o processo de busca. Basicamente, a principal característica da abordagem hierárquica está na utilização de ações capazes de resolver subobjetivos que formam um objetivo maior [13, 14]. Essa inferência de informações é feita através de *métodos*. Cada *método* é responsável por descrever quais ações ou subtarefas irão compor uma tarefa mais complexa. Se uma tarefa é primitiva, ela não pode ser subdividida em uma tarefa menor, pois é possível resolvê-la com as ações já descritas. O principal objetivo dessa abordagem é reduzir as tarefas complexas até que existam somente tarefas primitivas, obtendo assim o plano [14].

A maioria dos planejadores hierárquicos são independentes do domínio, mas os métodos, são específicos do domínio. Isso faz com que seja necessária a ação de um especialista para construir o domínio do problema, o que torna o planejador muito mais rápido.

JSHOP2 [15] é uma implementação de um planejador hierárquico desenvolvida na linguagem Java. Seu desenvolvimento foi inspirado no planejador SHOP2 [14]. O SHOP2 planeja as tarefas T na mesma ordem que serão executadas. Para isso, uma escolha não-determinística de uma tarefa $t \in T$ que não possui predecessores, resulta em dois casos:

1. se t é primitivo, então encontra-se uma ação a que resolve t de forma que as precondições são satisfeitas em um estado s, e aplica a em s. Se não existe essa ação, então esse ramo do espaço de busca falha;
2. se a tarefa t é composta, então o planejador encontra de forma não-determinística um método m que irá decompor a tarefa t em subtarefas. Se não existe um método para tal decomposição, então esse ramo do espaço de busca falha.

Se existe uma solução que envolve m, então suas subtarefas farão parte de uma nova lista atualizada T' que é percorrida recursivamente até que todas as tarefas sejam primarias e resolvíveis com operadores.

As duas escolhas não-determinística presentes no algoritmo apresentado recaem inevitavelmente em um processo de busca exponencial [13].

O planejador será responsável por gerar um plano de execução, que necessita de alguma camada que possa despachar as tarefas planejadas para os nós disponíveis. Para isso, propõe-se a utilização de um *gerenciador de execução*, descrito na seção 4.

4. Gerenciador de Execução

O modelo proposto carece de uma camada que possa distribuir as tarefas de forma coordenada. O *gerenciador de execução* recebe como entrada um *plano de execução* gerado pelo planejador. Esse *plano de execução* pode ser definido como P que é um conjunto de tarefas: $P = \{s_1, s_2, \ldots, s_n\}$. Cada tarefa é uma tripla formada por: $s_i = \{O, L, H\}$, onde:

- O é um número inteiro que define a ordem que a tarefa será executada. Assim, é possível definir que a tarefa s_i seja executada em uma ordem específica na linha de tempo da execução. Nessa definição é possível alocar mais de uma tarefa para uma determinada ordem. Dessa forma, a execução deverá ocorrer paralelamente;
- L define o tempo *limite* de execução estimado da tarefa em segundos;
- H é a identificação do nó que executará a tarefa.

Por utilizar um sistema de coleta de metadados que alimentam o planejador, é necessário que cada nó tenha uma identificação única H. Dessa forma, é possível estabelecer uma distribuição de carga baseada em históricos de execução.

O *gerenciador de execução* deve estar preparado para executar em um ambiente paralelo e escalável, pois a execução de fluxos de trabalho científicos exige grande quantidade de processamento. Portanto, é proposta uma adaptação do sistema *PeerUnit* [4].

4.1. PeerUnit

O sistema *PeerUnit* foi concebido inicialmente para realizar testes em sistemas *peer to peer*, afim de garantir as seguintes propriedades:

- *funcionalidade*: garante que o sistema irá responder como o esperado;
- *escalabilidade*: garante que a técnica aplicada poderá ser expandida para um ambiente com vários *peers*;
- *volatilidade*: garante que mesmo com a entrada ou a saída de *peers* o sistema continuará funcionando como o esperado.

A estrutura *PeerUnit* basicamente embute a todos os *peers* um código escrito na linguagem Java. Nele são definidas instruções de como tais elementos devem se comportar. Ainda é possível (através de anotações especiais) indicar qual método será executado por qual *peer* e qual será a ordem dessa execução. Além disso, é definido um tempo limite que se superado, retorna que o *peer* não foi capaz de executar o método com sucesso.

O código 1 representa um trecho de um caso de uso, no qual exemplifica-se a utilização do sistema *PeerUnit*.

Código 1: Caso de uso de exemplo.

```
1  @Test(range = "*", timeout = 100, order = 1)
2  public void join(){
3      peer.join();
4  }
5  @Test(range = 0, timeout = 100, order = 2)
6  public void put(){
7      peer.put(expectedKey,expected);
8  }
9  @Test(range = "1-3", timeout = 100, order = 3)
10 public void retrieve(){
11     actual = peer.get(expectedKey);
12 }
13 @Test(range = "1-3", timeout = 100, order = 4)
14 public void assertRetrieve(){
15     assert expected.equals(actual);
16 }
```

Nota-se que, antes da declaração do método, é definida uma linha de anotação iniciada pelo caractere @. A anotação é composta dos seguintes elementos:

- *range*: especifica quais serão os *peers* atingidos. Pode ser uma samblagem que representa somente um *peer* (3), uma faixa de atuação (1-3), elementos específicos (1,3,4), ou todos os *peers* (*);
- *order*: controla qual será a ordem de execução do método após a anotação;
- *timeout*: estabelece um tempo limite para execução. Caso esse tempo seja atingido sem que o método tenha sido executado com sucesso, a ação falha.

No código, 1 o método *join()* inicia todos os *peers*. O método *put()* faz com que o *peer* de identificação 0 insira um dado em uma variável global, visível para todos os outros *peers*. O método *retrive()* captura o dado inserido pelo *peer* 0. Por fim, o método *assertRetrive()* faz uma verificação se o valor inserido foi recuperado de forma correta.

O *gerenciador de execução* utiliza as mesmas características do *PeerUnit*. A implementação oferece uma estrutura que suporta as necessidades do plano de execução P. O único elemento não definido nativamente nas anotações é H, pois o mesmo não é especificado na camada de planejamento.

Na seção seguinte o modelo proposto é apresentado. Para tal, as definições de fluxo de trabalho científico, planejamento, e uma camada executora são levados em consideração.

5. Modelo Proposto

A figura 1 representa uma visão geral do modelo proposto.

A camada de *aplicação* é definido em um documento XML. Esse documento contém a descrição dos metadados necessários para executar os fluxos de trabalhos codificados na camada de *implementação*. Nessa descrição deve-se detalhar quais são os fluxos de trabalho, a forma de processamento distribuído e as dependências de dados.

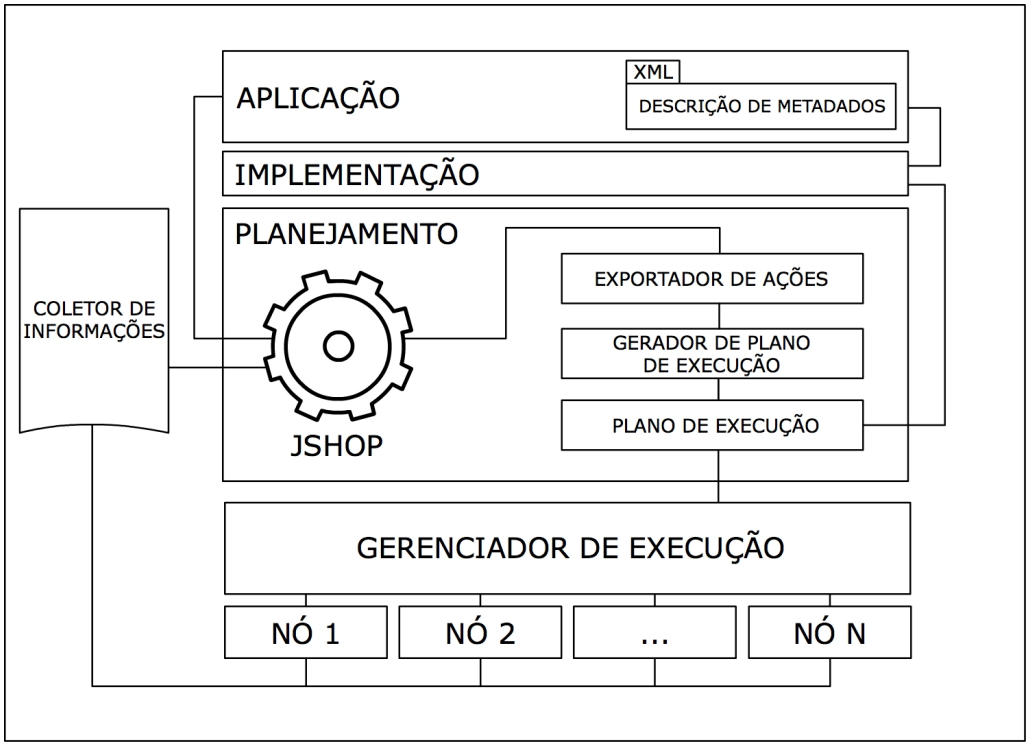

Figura 1: Visão geral do modelo proposto

A camada de *implementação* define as funcionalidades da aplicação. Estas são baseadas em regras definidas pelo *plano de execução*. Pode-se implementar diferentes formas de paralelização para uma mesma funcionalidade. Todas as operações descritas na camada de aplicação devem ser implementadas.

A camada de *planejamento* engloba os componentes: *JSHOP*, *exportador de ações*, *gerador do plano de execução* e *plano de execução*. O *JSHOP* é o planejador propriamente dito. Nele são processadas as informações da descrição de metadados e do *coletor de informações* de forma a se criar um plano de execução dos fluxos de trabalho disponíveis. O *coletor de informações* é um componente externo à camada de *planejamento* e está ligado diretamente a todos os planos de execução. Esse componente coleta os metadados sobre o processamento de tarefas em cada nó. O propósito desta ação é popular um banco de dados com informações relevantes para serem utilizadas nas execuções futuras do planejador. O *exportador de ações* é um componente acessado diretamente pelo *JSHOP*. Esse componente traduz o resultado fornecido pelo planejador para um documento padronizado XML. Este dará origem ao *plano de execução* através do componente *gerador de plano de execução*, que interpreta a descrição XML recebida e traduz em um formato interpretável para o *gerenciador de execução*. Finalmente, o componente *plano de execução* é, como o nome sugere, o plano de execução propriamente dito e pronto para ser enviado ao *gerenciador de execução*.

O *gerenciador de execução* é uma camada que implementa o *PeerUnit* [4]. Recebe-se um plano de execução semelhante às configurações demonstradas no código 1. Desta forma, aciona-se devidamente qual nó deve executar qual trecho do plano.

No modelo proposto, utiliza-se um planejador hierárquico como motor de distribuição de fluxos de trabalho para a camada de execução. Com o planejamento

hierárquico é possível descrever o domínio do problema, bem como seus métodos, de forma a otimizar o processo de escalonamento. Além do escalonamento, são aproveitadas informações provenientes de metadados, coletados dos nós da camada de execução. Essas informações são utilizadas para alimentar automaticamente o domínio do sistema, possibilitando a obtenção de informações relevantes para próximas execuções. Assim, ao permitir diferentes formas de paralelização de uma funcionalidade do sistema, o planejador pode avaliar qual a melhor, baseando-se no histórico de execuções.

Dentro do cenário apresentado, verifica-se a necessidade da utilização de um ambiente controlado. Isto porque, além de escalonar as tarefas de um fluxo de trabalho, o planejador também tem a função de analisar o histórico de execução e aproveitar informações do coletor de informações. Com isso, é possível atribuir não somente uma ordem com que as tarefas serão executadas, mas também em qual nó especificamente. Para tanto, deve-se obter a mesma identificação de um *host* toda vez que o sistema for iniciado. Assim, defini-se que a ordem com que os nós serão inseridos no sistema deve ser a mesma.

Para ilustrar o funcionamento do modelo proposto, propõe-se um exemplo que demonstra o fluxo de execução, descrito na seção 5.1.

5.1. Exemplo de Funcionamento

Seja um fluxo de trabalho científico, um conjunto W composto por três tarefas a serem executadas $W = \{s_1, s_2, s_3\}$. Em um ambiente composto por três nós $Y = \{n_1, n_2, n_3\}$. Após a definição dos metadados que descrevem W, implementa-se as funcionalidades que executam s_1, s_2 e s_3. Mais de uma implementação pode ser definida para a mesma tarefa s_1. Uma implementação pode ser feita de forma sequencial s_{1s} e outra explorando as possibilidades de paralelização s_{1p}.

Em seguida são extraídas as informações do *coletor de informações* a fim de popular o planejador. Caso haja execuções anteriores, são consideradas informações referentes ao tempo de execução e recursos (nós) disponíveis para execução. Desta forma, o planejador consegue ponderar qual a ordem de execução. Assume-se nesse exemplo que não há dependência entre as tarefas a serem executadas.

Dado que a tarefa s_1 tem múltiplas implementações, o planejador pondera quais foram executadas com sucesso em casos anteriores e seleciona a que minimiza o tempo de execução. Seja o *plano de execução* definido por $P = \{s_{1s}, s_2, s_3\}$, temos que s_{1s}, s_2 e s_3 são triplas, formadas por: $s_{1s} = \{1, 1000, n_2\}$, $s_2 = \{3, 500, n_1\}$ e $s_3 = \{2, 500, n_3\}$. Os parâmetros representam respectivamente: a ordem na linha de execução, o tempo estimado em segundos para execução e o nó onde a tarefa será executada.

Invoca-se, então, o *gerenciador de execução*, que despacha as tarefas para os seus respectivos nós de acordo com as definições do plano de execução. Cada nó ainda é observado pelo componente *coletor de informações* que recupera e re-alimenta a base de metadados a cada execução da aplicação.

6. Conclusão

O ambiente científico produz *terabytes* de informações que precisam ser processadas e analisadas. A utilização de ferramentas com interfaces amigáveis e transparentes

permitem aos cientistas processarem e extraírem informações para suas pesquisas. Para tanto, são criados fluxos de trabalho científicos, que dividem uma consulta maior em partes menores, permitindo, assim, o processamento e a distribuição destas partes estrategicamente.

O modelo proposto visa esta facilidade e melhoria para o tratamento dos fluxos de trabalho científicos. Este modelo está em fase de implementação. Para testes iniciais propõe-se a utilização de fluxos de trabalhos na área de processamento de sequências de DNA, por exemplo. As transformações nessas cadeias devem possibilitar o mapeamento de subtarefas em fluxos de trabalho. Estes são problemas que tratam grandes volumes de dados em fluxos de trabalho complexos e que demandam muito processamento.

A implementação deste modelo está em andamento e utiliza a linguagem Java. A motivação para o uso da linguagem é relacionada a linguagem de desenvolvimento da camada de execução (*PeerUnit*) e do planejador (*JSHOP*). A estrutura da codificação está seguindo os padrões do projeto *PeerUnit*, fazendo dele uma ramificação direta, porém com um objetivo diferente, ou seja, ao invés de testar *peer*, utiliza-se a mesma estrutura para a construção de um ambiente de execução inteligente.

Uma característica do modelo proposto é a modularização dos componentes, que se interligam através de arquivos XML. Desta forma, será criado um modelo que pode ser adaptado para executar com diferentes componentes. Por exemplo, se houver a necessidade de alteração do escalonador de fluxos de trabalho, basta retirar o planejador e inserir um outro escalonador que exporte as ações no mesmo formato aceito pelo modelo. Ou ainda, caso seja necessário alterar a camada de execução, o mesmo procedimento pode ser acionado.

Referências

[1] Simmhan, Y. L., Plale, B., and Gannon, D., "A survey of data provenance in e-science," *ACM SIGMOD Record*, Vol. 34, No. 3, Sept. 2005, pp. 31.

[2] Hollingsworth, D., "Workflow Management Coalition - The Workflow Reference Model," Tech. rep., Workflow Management Coalition, Jan. 1995.

[3] Deelman, E., Gannon, D., Shields, M., and Taylor, I., "Workflows and e-Science: An overview of workflow system features and capabilities," *Journal of Future Generation Computer Systems*, Vol. 25, No. 5, May 2009, pp. 528–540.

[4] Almeida, E. C. D., Sunyé, G., Traon, Y. L., and Valduriez, P., "A Framework for Testing Peer-to-Peer Systems," *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*, Nov. 2008, pp. 167–176.

[5] Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., and Mock, S., "Kepler: an Extensible System for Design and Execution of Scientific Workflows," *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*, IEEE Computer Society, 2004, pp. 423–424.

[6] Belhajjame, K., Wolstencroft, K., Corcho, O., Oinn, T., Tanoh, F., William, A., and Goble, C., "Metadata Management in the Taverna Workflow System," *Journal of 8th IEEE International Symposium on Cluster Computing and the Grid*, Vol. 0, May 2008, pp. 651–656.

[7] Akram, A., Meredith, D., and Allan, R., "Evaluation of BPEL to Scientific Workflows," *Proceedings of 6th IEEE International Symposium on Cluster Computing and the Grid*, IEEE Computer Society, 2006, pp. 269–274.

[8] Lee, K., Paton, N. W., Sakellariou, R., Deelman, E., Fernandes, A. A. A., and Mehta, G., "Adaptive Workflow Processing and Execution in Pegasus," *Proceedings of 3rd International Conference on Grid and Pervasive Computing Workshops*, May 2008, pp. 99–106.

[9] Weld, D., "Recent advances in AI planning," *AI magazine*, Vol. 20, No. 2, 1999, pp. 93.

[10] Fikes, R. E. and Nilsson, N. J., "Strips: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, Vol. 2, No. 3-4, Jan. 1971, pp. 189–208.

[11] Blum, A., "Fast Planning Through Planning Graph Analysis." Tech. Rep. 1-2, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, Feb. 1995.

[12] Kautz, H., "Planning as satisfiability," *Proceedings of the European Conference on*, , No. August 1992, 1992, pp. 1–12.

[13] Nau, D., Cao, Y., Lotem, A., and Muñoz Avila, H., "SHOP: Simple hierarchical ordered planner," *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, Morgan Kaufmann Publishers Inc., 1999, pp. 968–973.

[14] Nau, D., Au, T., Ilghami, O., Kuter, U., Murdock, J., Wu, D., and Yaman, F., "SHOP2: An HTN planning system," *Journal of Artificial Intelligence Research*, Vol. 20, No. 1, 2003, pp. 379–404.

[15] Ilghami, O., "Documentation for JSHOP2," *Tech. rep, Department of Computer Science, University of Maryland*, Vol. 51, 2006.