Uso de Planejamento em Inteligência Artificial para o Desenvolvimento Automático de Software

Luciano A. Digiampietri, José J. Pérez-Alcázar, Ricardo Sudário e Freitas,
Jonatas C. Araújo, Éric H. Ostroski, and Caio R. N. Santiago
Escola de Artes, Ciências e Humanidades da Universidade de São Paulo (EACH-USP),
São Paulo, SP, Brasil
email: digiampietri@usp.br

***Abstract*—Com a grande disponibilidade de ferramentas de software na forma de serviços Web, aplicações locais e bibliotecas de funções, surge a necessidade do desenvolvimento de meios automatizados para a composição dessas ferramentas de forma a prover funcionalidades mais complexas. Workflows são uma das formas de se organizar conjuntos de atividades com o objetivo de atingirem uma dada meta. Este artigo apresenta uma infraestrutura que usa planejamento em inteligência artificial para a composição automática e execução de workflows considerando as características sintáticas e semânticas das interfaces das atividades envolvidas na composição. Além disso, os workflows resultantes podem ser exportados como códigos convencionais podendo ser facilmente personalizados ou estendidos pelo usuário.**

I. Introdução

A grande disponibilidade de bibliotecas de funções, componentes de *software*, aplicativos locais e serviços Web tem facilitado e tornado muito mais eficiente o desenvolvimento de software [1], [2]. Com estes recursos, diversos sistemas podem ser criados simplesmente combinando elementos básicos existentes e, eventualmente, personalizando o resultado dessa composição.

Um dos grandes desafios para o desenvolvimento de software do século XXI é permitir que esses elementos básicos sejam compostos automaticamente, fornecendo ao usuário um sistema que além de correto possa também ser atualizado ou personalizado facilmente.

Neste artigo é apresentada uma infraestrutura que utiliza planejamento em inteligência artificial para a composição automática de componentes de software. Estes componentes podem ser serviços Web, métodos desenvolvidos na linguagem Java e aplicativos locais. Em particular, será dado destaque aos métodos em linguagem Java. Há quatro razões principais para este destaque: (a) existem muitos métodos disponíveis, incluindo todos os métodos já existentes nas bibliotecas básicas da linguagem; (b) o resultado da composição será um novo sistema em Java, que poderá ser executado dentro da infraestrutura desenvolvida ou como um código Java para execução local; (c) por ser um código Java, este poderá ser personalizado ou estendido pelo usuário como se não tivesse sido gerado automaticamente; e (d) o sistema resultante poderá, por si, só ser considerado um novo componente de software disponível para futuras composições.

Planejamento em Inteligência Artificial foi escolhido como técnica para a composição automática devido à maturidade deste tipo de solução e capacidades providas por ela [3]. Neste artigo serão discutidas as principais características, limitações e desafios relacionados a este tipo de composição automática.

O restante deste artigo está organizado da seguinte forma. A Seção II contém alguns conceitos básicos fundamentais para o entendimento do artigo e trabalhos correlatos. A Seção III apresenta a solução desenvolvida, desde a especificação da arquitetura até os aspectos de implementação. A Seção IV apresenta um estudo de caso incluindo cópias de tela do sistema desenvolvido, código exportado automaticamente e o resultado da execução do estudo de caso. Por fim, a Seção V contém as conclusões e trabalhos futuros.

II. Conceitos Básicos e Trabalhos Correlatos

Diversos sistemas têm sido criados como a execução de um conjunto de atividades em uma dada ordem. Atualmente, é cada vez mais comum encontrar ferramentas disponíveis para a execução de cada uma das atividades envolvidas, tanto na forma de serviços Web, como aplicativos locais ou bibliotecas de funções. Este tipo de desenvolvimento de soluções é muito comum em processos de negócio e experimentos científicos, mas pode ser aplicado ao desenvolvimento de qualquer tipo de software.

Diversas soluções foram propostas para o armazenamento, compartilhamento e execução de experimentos científicos e processos de negócio na forma de workflows. Neste artigo, um **workflow** representa a execução de um conjunto de atividades básicas, podendo ser sequencial ou paralela, podendo conter decisões e laços, envolvendo ou não interação com o usuário. Partindo desta definição, qualquer software poderá ser visto como um workflow, dependendo apenas da definição do que são as atividades básicas.

As **atividades básicas** utilizadas neste trabalho são de três naturezas diferentes: serviços Web, métodos na linguagem Java e aplicativos locais. Estes três tipos de recursos poderão ser usados intercaladamente num mesmo workflow, porém será dado destaque aos métodos em Java, pois o workflow resultante, como será visto nas próximas seções, será um novo sistema em Java com todas as vantagens de interoperabilidade desta linguagem.

Para a composição automática de atividades existem trabalhos baseados no casamento das interfaces das atividades, no uso de semântica para validação da integração de dados e o uso de planejamento em inteligência artificial [4], [5]. A maioria das propostas ainda está em fase inicial de desenvolvimento deixando em aberto diversas questões tecnológicas e científicas. Além disso, muitas das propostas se restringem a composição automática de serviços Web. Esta restrição faz com que os workflows produzidos não sejam, a priori, ideais para tratar grandes volumes de dados ou processamentos de longa duração, devido os custos de transmissão de dados na Web e necessidade da persistência dos resultados.

Este trabalho utiliza Planejamento em Inteligência Artificial (IA) para a composição das atividades básicas. **Planejamento em IA** é a tarefa de apresentar uma sequência de ações a fim de atingir uma determinada meta [6], [7], esta sequência de tarefas é chamada de **plano**. A estratégia de planejamento adotada foi o **planejamento hierárquico**, que é uma técnica de planejamento baseada na decomposição sucessiva de tarefas. Um dos planejadores hierárquicos mais conhecidos é o SHOP2 (*Simple Hierarchical Ordered Planner 2*) [8] que utiliza Rede Hierárquica de Tarefas (*Hierarchical Task Network (HTN)*) [7]. Outras estratégias usadas para composição de tarefas básicas foram descritas em [9].

Ao se compor peças básicas de software existem alguns aspectos que precisam ser levados em consideração. Um dos aspectos primordiais está relacionado à integração de dados. Neste trabalho são tratados aspectos de **integração de dados** ligados à compatibilidade dos dados das entradas e saídas de cada uma das interfaces dos componentes de software. A integração de dados é verificada de duas formas: sintaticamente e semanticamente [10].

A **integração sintática** é responsável por garantir que a saída de um componente apresente um tipo de dados compatível com a entrada do componente seguinte. Por exemplo, um componente produz um valor inteiro e o componente seguinte espera receber um valor inteiro (ou mesmo um valor decimal). A **integração semântica** é responsável por garantir que os dados são semanticamente compatíveis, isto é, independentemente do tipo do dado, se o conteúdo (significado) dele é ou não compatível. Um exemplo de compatibilidade sintática e incompatibilidade semântica é: um componente tem como saída o valor decimal da temperatura em graus *Celsius*, porém o componente seguinte espera receber o valor decimal da temperatura em *Fahrenheit*. Apesar dos dois serviços tratarem de valores decimais de temperatura não é correto, sem nenhum tratamento prévio, utilizar a saída do primeiro componente como entrada do segundo.

Para possibilitar a integração semântica de dados é necessário o uso de mecanismos de anotação semântica. Uma das maneiras de representar conhecimentos semânticos é utilizar ontologias. Uma **ontologia** contém a descrição compartilhada de conceitos e das relações entre os conceitos de um dado domínio. Ontologias podem ser usadas tanto para resolver questões de ambiguidade como também para permitir o processamento automatizado de recursos [11].

Existem três **modelos ontológicos** principais desenvolvidos para facilitar a anotação semântica e recuperação de recursos, especialmente para anotação de Serviços Web Semânticos (WSMO[1] , WSDL-S[2] e OWL-S[3]). Uma breve comparação desses três modelos pode ser encontrada em [12].

Existem diversos tipos de propostas para tratar a **composição automática** de tarefas ou serviços [3], [9], [13]–[16]. Cada uma dessas aproximações tem vantagens e desvantagens sobre as demais e são mais adequadas para tratar alguns conjuntos específicos de problemas. Dentre estes tipos de propostas, há três mais relevantes no contexto deste artigo.

O primeiro utiliza apenas o casamento de interfaces entre as tarefas (casamento da saída de uma tarefa com a entrada da próxima) e é adequado para sistemas que utilizam fluxos de transformação de dados (*pipelines*), porém considera apenas a sintaxe dos dados de entrada e saída (o tipo) e não a semântica associada a esse dado. Além disso, mudanças de estado não são modeladas nesse tipo de solução (não possibilitando a representação de pré e pós-condições). Este tipo de solução é tipicamente usado em domínios restritos.

O segundo tipo de solução é uma extensão do primeiro considerando a semântica relacionada aos dados. Nesta abordagem o casamento das interfaces considera, além dos tipos, os conceitos relacionados a cada entrada e saída. Desta forma, o sistema consegue distinguir (e perceber a incompatibilidade) entre uma *string* que contenha uma sequência de DNA e uma *string* que contenha a descrição de uma imagem. Este tipo de solução permite ao sistema gerenciar componentes de diferentes domínios e evita a geração de composições de workflows semanticamente incorretos.

O terceiro tipo de solução considera, além do casamento sintático e semântico das interfaces dos serviços, o uso de pré e pós-condições possibilitando que uma tarefa explicite as mudanças no estado do mundo requeridas para sua execução ou produzidas após sua execução. Dois exemplos de fatos que não podem ser modelados na primeira e segunda propostas são: (a) uma tarefa que modifica o banco de dados (mas não produz nenhum dado de saída); (b) uma tarefa que só pode ser executada após o usuário estar *logado* no sistema.

Enquanto o terceiro tipo de solução é o mais completo, permitindo um conjunto muito maior de representações e verificações, há um problema sério envolvendo seu desenvolvimento: a gama de mudanças de estados possíveis deve ser conhecida pelo sistema e todas as atividades devem ser especificadas de acordo com esses estados. Isto torna a especificação das atividades muito mais trabalhosa e dificulta o compartilhamento das atividades entre diferentes usuários ou sistemas [3].

III. Solução Desenvolvida

Esta seção apresenta a solução desenvolvida que considera os três tipos de composição descritos na seção anterior. A utilização desses tipos de composição permite aos usuários

[1] www.wsmo.org
[2] http://www.w3.org/Submission/WSDL-S/
[3] www.daml.org/services

o uso de recursos mais ou menos complexos dependendo exclusivamente de como as atividades (softwares ou serviços) foram anotados dentro do sistema.

Esta seção está dividida em três partes. A Subseção III-A apresenta a arquitetura especificada e desenvolvida neste trabalho. A Subseção III-B contém a descrição de como os domínios são criados para o planejador. Por fim, a Subseção III-C contém uma detalhada discussão sobre os principais desafios técnicos enfrentados.

A. Arquitetura

Para possibilitar a composição automática de componentes básicos de software, uma arquitetura para gerenciamento de experimentos científicos [17] foi estendida com a adição de um módulo contendo um planejador. A Figura 1 apresenta a arquitetura desenvolvida, hachurados estão os módulos que foram adicionados ou significativamente estendidos em relação a arquitetura original.

A seguir todos os módulos da arquitetura serão apresentados, destacando-se apenas os referentes às extensões desenvolvidas neste artigo.

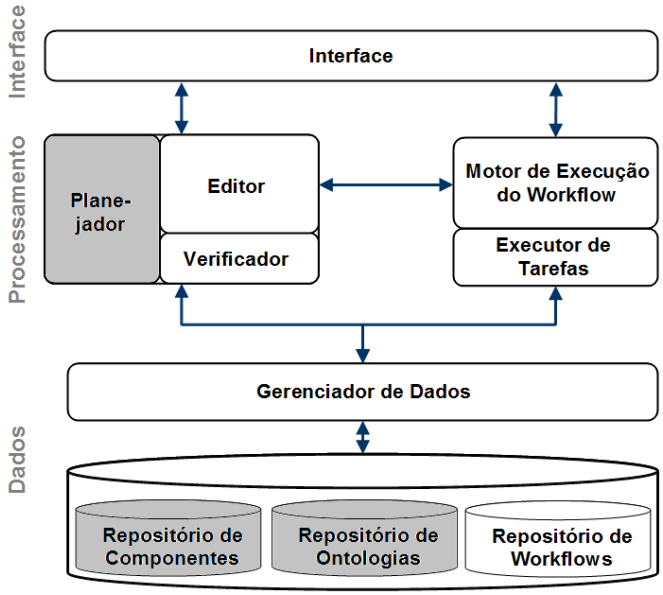

Fig. 1. Arquitetura Desenvolvida

Interface: interface gráfica que provê ao usuário final acesso às funcionalidades do sistema. Através da interface o usuário pode converter um plano gerado pelo planejador para um workflow; modificar e executar um workflow (utilizando as funcionalidades do módulo *Editor*; definir parâmetros de entrada e exportar o código do workflow, para ser executado fora da infraestrutura desenvolvida).

Editor: contém as funcionalidades ligadas à criação e manipulação de workflows.

Verificador: módulo responsável por verificar se um workflow possui algum tipo de problema em suas dependências de dados (sintática ou semântica) ou de fluxo de controle. Este módulo também verifica se todos os dados de entrada e parâmetros estão preenchidos e, assim, o workflow pode ser executado.

Planejador: módulo que utiliza planejamento hierárquico para criar planos que atendam aos requisitos do usuário. O plano resultante (ou planos resultantes) é lido pelo sistema e transformado em workflows que podem ser editados pelo usuário, avaliados pelo módulo *verificador*, executados ou exportados. Para este módulo foram utilizados dois planejadores hierárquicos diferentes, descritos na Subseção III-C.

Motor de Execução do Workflow: módulo responsável pela execução de workflows. Ele coordena a execução concorrente das tarefas, gerenciando as dependências de dados e controle.

Executor de Tarefas: módulo responsável pela execução individual de cada tarefa. Na implementação atual, cada tarefa é executada em uma *thread* permitindo a execução paralela das mesmas (respeitando-se as dependências). Este é o único módulo do sistema que reconhece as diferentes naturezas dos tipos de tarefas (método na linguagem Java, aplicativo local ou serviço Web). Para o restante do sistema, todas as tarefas são tratadas da mesma maneira.

Gerenciador de Dados: módulo responsável por fazer a interface entre os repositórios de dados e os módulos de processamento.

Repositório de Componentes: repositório que contém as descrições e anotações de todas as tarefas/componentes de software. Para possibilitar a composição automática de qualidade é recomendado que todos os componentes tenham sua funcionalidade principal anotada semanticamente (segundo uma ontologia de componentes/serviços) e que cada um dos parâmetros de entrada e saída sejam também anotados (segundo uma ontologia de domínio). Além disso, para a utilização de todas as funcionalidades do planejador, cada componente deve ter suas pré-condições representadas (condições necessárias para que o componente possa executar, além dos dados de entrada), bem como as pós-condições (mudanças no estado do mundo causadas pela execução do componente). O funcionamento detalhado desse sistema de anotação é descrito na Subseção III-B.

Repositório de Ontologias: repositório que contém dois tipos de ontologias: *ontologia de domínio*, descrevendo os conceitos de um dado domínio de aplicação, por exemplo, bioinformática ou processamento de imagens; e *ontologia de componentes* descrevendo os conceitos relacionados às funcionalidades dos componentes, num dado domínio.

Repositório de Workflows: repositório que contém os workflows já salvos pelo sistema, incluindo aqueles gerados pelo planejador. Pode conter informações relativas à rastreabilidade e proveniência de dados [18], [19]. Todos os workflows sem pendências, já avaliados pelo módulo verificador, podem, por si só, serem considerados componentes de software.

B. Definição dos Domínios e Anotações

Neste trabalho dois tipos principais de ontologias são utilizados: ontologias de domínio e ontologias de componentes. As *ontologias de domínio* contém a descrição e os relacionamentos entre os conceitos de um dado domínio. As ontologias

devem ser construídas por especialistas do domínio e servem de base para a anotação dos componentes de software e suas interfaces. É recomendado que seja desenvolvida uma ontologia (ou subontologia) específica para cada domínio ao invés do uso de ontologias gerais pois ontologias específicas facilitam o uso do planejador por aumentar a qualidade dos planos gerados (impedindo que tarefas de outros domínios sejam inseridas indevidamente nos planos) e reduzindo o número de planos gerados (facilitando a tarefa do usuário de escolher o plano que melhor satisfaz sua requisição). Além das ontologias de domínio há uma ontologia geral de tipos de dados. Esta ontologia serve para descrever os tipos básicos de dados, permitindo ao planejador e ao verificador de workflows avaliarem a compatibilidade sintática entre tipos de dados. Sem esta ontologia não seria possível, automaticamente, saber que uma saída com valor inteiro é compatível com uma entrada que espera receber um valor decimal.

As *ontologias de componentes* servem para descrever as funcionalidades dos componentes. É possível definir uma única ontologia que contenha os conceitos da ontologia de domínio e da ontologia de componentes correspondente, por mais que, a priori, estas duas ontologias não possuam interseção (uma serve para anotar os dados de entrada e saída, já a outra serve para anotar apenas as funcionalidades dos componentes). A Figura 2 contém um exemplo simplificado dessas ontologias. No exemplo, uma única ontologia foi utilizada para descrever o domínio, as funcionalidades e os tipos de dados. Neste projeto está sendo utilizada a ferramenta Protégé[4] para a criação das ontologias.

Na Figura 2 é possível observar um pequeno subconjunto dos conceitos ligados aos tipos de dado (*DataType*). Há também conceitos de mais de um domínio. Dentro do domínio de processamento de imagens é possível observar conceitos de descrição do domínio (*Domain* como subconceito de *Image-Processing*) e conceitos para descrever a funcionalidade dos componentes de software (*Functionality*). A raiz da ontologia é o conceito *Thing* a partir do qual todos os demais conceitos são criados.

O uso de planejamento em inteligência artificial permite que os três tipos de composição citados anteriormente sejam efetuados. Ao se usar planejamento hierárquico, a única diferença entre os três tipos está relacionado a como o domínio do planejador será produzido. Está subseção descreve como esses domínios são definidos.

É comum dividir o planejamento [7], [8] em duas etapas principais: definição do domínio do planejador e definição do problema. Eventualmente uma terceira etapa é usada, para descrever como as soluções (planos) deverão ser geradas. A seguir, estas etapas são descritas considerando o planejamento hierárquico.

A *Definição do Domínio do Planejador* define os operadores e métodos do domínio. Operadores são as tarefas básicas, incluindo suas pré e pós-condições. Métodos são tarefas potencialmente compostas. No planejamento hierárquico problemas são resolvidos iterativamente pela decomposição dos métodos, assim, os operadores só serão considerados caso sejam utilizados por métodos. Os métodos podem descrever alternativas para resolver o mesmo problema, por exemplo, um método de filtro de imagem pode ser decomposto em dois outros métodos filtro passa-alta e filtro passa-baixa. Cada um desses métodos pode ser sucessivamente decomposto em outros métodos ou tarefas básicas (operadores).

A *Definição do Problema* define o estado inicial do mundo e o estado desejado (meta).

A primeira diferença entre o planejamento tradicional e o desenvolvido para compor componentes de software é que o planejamento tradicional não trata entradas e saídas e sim mudanças no estado do mundo. Para adaptar esta característica, ao criar o domínio do planejador cada operador recebe, além de suas pré-condições, uma pré-condição especial para cada parâmetro de entrada indicando que o operador (componente de software) só poderá ser executado caso os dados de entrada estejam disponíveis. Analogamente, além das pós-condições, cada componente que produza alguma saída terá uma pós-condição adicional referente a produção deste dado de saída.

A definição do problema poderá ter estados iniciais e estado meta contendo tanto pré e pós-condições normais quanto as referentes a entradas e saídas. Por exemplo, um usuário pode definir um estado inicial como: possuo uma imagem de entrada e estou *logado* no sistema, e meu objetivo é possuir a imagem segmentada.

Anotar os componentes de acordo com as ontologias é uma tarefa muito importante no sistema e, até o momento, não existem soluções gerais para o problema de anotação automática de componentes de software [20]. Desta forma, a versão atual do sistema depende da anotação manual dos componentes. Como será apresentado na próxima subseção, esta anotação não é obrigatória, mas a qualidade e quantidade dos planos gerados dependem muito dela.

Cada entrada e saída pode ser anotada como pertencendo a um conceito da ontologia de domínio e cada componente pode ser anotado como executando um certo tipo de funcionalidade, da ontologia de domínio. Os tipos de dados são anotados automaticamente segundo a ontologia de tipos, por exemplo, se a saída de um componente retorna a temperatura em graus *Celsius* em valores decimais, o sistema automaticamente anotara que, sintaticamente esta saída está associada ao tipo *Double* que é um subtipo do conceito *Number*. Porém apenas o usuário poderá anotar que, semanticamente, esse número corresponde à temperatura em graus *Celsius*.

Além disso, cada entrada dos componentes pode receber uma marca para indicar que esta entrada deverá ser ignorada durante o planejamento. Este tipo de marcação é útil para evitar que parâmetros de configuração, como por exemplo, *nome do arquivo de saída* ou *valor de um dado limiar* sejam inseridos no domínio do planejador, pois o planejador não teria como decidir os valores desses atributos. Este tipo de entrada é específico demais para ser utilizado pelo planejador e deverá ser preenchido pelo usuário apenas na hora de executar

[4]http://protege.stanford.edu/

Fig. 2. Recorte da Ontologia Utilizada

o workflow resultante.

A Figura 3 apresenta um pequeno exemplo de definição de domínio, representada no formato utilizado pelo planejador SHOP2 [8]. Neste exemplo três componentes de software são definidos como operadores básicos: *lowPassFilter*, *backgroundRemoval* e *highPassFilter*. Nesta definição é possível ver, após o nome do operador, quais são os parâmetros utilizados por eles, as pré-condições e pós-condições. Por exemplo, o operador *lowPassFilter* recebe um parâmetro a, que precisa ser do tipo *image* e produz como resultado um elemento do tipo *filteredImage*. O último parâmetro de cada operador corresponde a seu custo. Este custo será utilizado pelo planejador a fim de obter os planos menos custosos.

Neste domínio também é possível observar a presença de três métodos *m_filter*, *m_lowPassFilter*, e *m_highPassFilter*. Um método contém em sua descrição os parâmetros que serão tratados por ele, as pré-condições e decomposição do método em operadores ou outros métodos. É possível observar que o método *m_filter* pode ser decomposto no método *m_lowPassFilter* ou *m_highPassFilter*. O método *m_lowPassFilter* tem como pré-condição a existência de uma imagem *x* que ainda não tenha sido filtrada e é decomposto pela execução das operações *backgroundRemoval* e *lowPassFilter*.

Um problema definido em SHOP2 consiste de um estado inicial e da solicitação da execução de algum método, a Figura 4 a apresenta um exemplo de problema para o domínio da Figura 3. Neste problema, e informado que se possui um objeto i1 do tipo *image* e se deseja que seja executado um filtro (*m_filter*) sobre esse objeto.

O resultado da resolução do problema, solicitando-se ao planejador apenas a melhor resposta pode ser observado na Figura 5. O plano resultante contém a execução de dois operadores: *backgroundremoval* e *lowpassfilter*. São planos como este que serão transformados em workflows para serem manipulados pelo usuário, executados ou exportados como códigos Java.

```
(defproblem problem imageProcessing
 ((image i1))
 ((m_filter i1))
)
```

Fig. 4. Exemplo de Definição de Problema

C. Aspectos de Implementação

Esta subseção descreve os mecanismos de conversão automática dos domínios e anotações de componentes em domínios do planejador. Além disso, é apresentado o mecanismos de conversão de planos em workflows, um novo planejador que foi desenvolvido e a exportação dos workflows em código Java.

```
(defdomain imageProcessing (
  (:operator (!lowPassFilter ?a) ((image ?a)) () ((filteredImage ?a)) 10.0)
  (:operator (!backgroundRemoval ?a) ((image ?a)) () () 13.0)
  (:operator (!highPassFilter ?a) ((image ?a)) () ((filteredImage ?a)) 25.0)

  (:method (m_filter ?x)
    ()
    ((m_lowPassFilter ?x))
    ()
    ((m_highPassFilter ?x))
   )

   (:method (m_lowPassFilter ?x)
     ((image ?x) (not (filteredImage ?x)))
     ((!backgroundRemoval ?x) (!lowPassFilter ?x))
   )

   (:method (m_highPassFilter ?x)
     ((image ?x) (not (filteredImage ?x)) )
     ((!highPassFilter ?x))
   )
  )
 )
```

Fig. 3. Exemplo de Definição de Domínio

```
Plan #1:
Plan cost: 23.0

(!backgroundremoval i1)
(!lowpassfilter i1)
--------------------

Time Used = 0.0040
```

Fig. 5. Exemplo de Resultado

A fim de possibilitar a composição automática é necessário criar domínios na linguagem entendida pelo planejador. Esta tarefa pode ser totalmente automatizada. O tipo de composição utilizado dependerá das informações disponibilizadas pelo usuário.

Para a composição utilizando apenas casamento sintático das interfaces das tarefas não é necessário ao usuário ter anotado nenhuma tarefa. Já que este tipo de composição envolve apenas os tipos de entradas e saídas, o domínio para o planejador pode ser criado com dados extraídos automaticamente das interfaces dos serviços Web ou dos parâmetros dos métodos da linguagem Java. Apenas os aplicativos locais precisam ter esse tipo de informação fornecida pelo usuário. Com base nas interfaces desses componentes o sistema criará um operador para cada componente e um método para cada tipo de transformação de dados. Para definir um problema o usuário terá apenas que escolher que tipo de dado de saída ele deseja em função dos dados de entrada que possui. Este tipo de solução é interessante por não exigir do usuário nenhum tipo de anotação semântica dos dados, porém produzirá muitos planos (workflows) e estes provavelmente não serão muito úteis devido a generalidade dos dados envolvidos. Todos os componentes serão entendidos como transformadores de dados de entrada em dados de saída, sem nenhuma semântica associada a estes dados, ou seja, dois componentes terão suas funcionalidades consideradas idênticas caso suas entradas e saídas sejam do mesmo tipo sintático. Por exemplo, dois componentes recebem um número com entrada e produzem uma *string* (sequências de caracteres) como saída, por mais que um dos componentes receba um CEP e retorne o nome do bairro e o outro receba um identificador e retorne uma sequência de DNA, para o planejador os dois serão equivalentes.

A fim de melhorar a qualidade dos planos gerados é necessário que os dados das entradas e saídas dos componentes sejam anotados semanticamente. Para isto basta que, sempre que um novo componente seja inserido no sistema o usuário relacione cada entrada e saída a um conceito da ontologia. Além disso, é recomendado que o usuário associe uma funcionalidade ao componente. Com isto é possível ao gerador do domínio do planejador que ele decomponha os métodos de acordo com suas funcionalidades além de permitir apenas o casamento semântico das interfaces. A decomposição dos componentes em métodos funciona da da seguinte maneira: dado um superconceito na ontologia de componentes, por exemplo, *Filtro* e dois subconceitos *FiltroPassaBaixa* e *FiltroPassaAlta* será criado no domínio do planejador um método chamado Filtro que poderá ser executado por qualquer componente que execute um FiltroPassaBaixa ou um FiltroPassaAlta.

Por fim, para o terceiro tipo de composição, que utiliza pré e pós-condições, além das entradas e saída, é necessário que o usuário defina para cada componente de software quais são (caso existam) suas pré e pós-condições. No sistema apresentado neste artigo as condições são modeladas como predicados com quatro parâmetros: *nome do predicado*, *primeiro termo*, *segundo termo* e *valor*. Por exemplo, nome do predicado:estarLogado, primeiro termo:usuário, segundo termo:sistema, valor:true, significando que a condição é que o usuário esteja *logado* no sistema. Fazer este tipo de anotação pode ser considerado muito custoso ao usuário [3], mas é importante lembrar que nem todas as atividades possuem pré e pós condições e só serão verificadas pré-condições dos componentes que tiverem esta característica definida. A exportação

desse tipo de anotação para o domínio do planejador é feita de maneira direta pois o planejador já espera receber as pré e pós-condições de cada operador.

Para a definição do problema a ser resolvido pelo planejador, basta ao usuário informar quais tipos de dados ele possui e qual tipo de dado ele deseja que seja produzido. Estes tipos de dados são conceitos na ontologia de tipos de dados ou, preferencialmente, da ontologia de domínio. Pelo fato de para cada tipo de transformação de dados o sistema criar automaticamente um método no domínio do planejador, sempre que houver ao menos um componente que produza o tipo de dado desejado haverá um método que corresponda ao objetivo do usuário.

Os planos gerados são importados pelo sistema e convertidos para workflows. Como mostrado no exemplo da Figura 5 o plano produzido consiste de uma sequência de operações que devem ser executadas. O pseudocódigo para a reconstrução de um workflow a partir de um plano é apresentado pela Figura 6.

Para cada operador no plano:
  criar a atividade equivalente no workflow
  criar uma dependência de fluxo de controle entre a atividade atual e a atividade anterior
  Para cada entrada da atividade atual:
    Se houver saída compatível nas atividades anteriores:
      criar um fluxo de dados da saída para esta entrada

Fig. 6. Pseudocódigo da Conversão de Planos para Workflows

Existem duas limitações principais na conversão apresentada pelo pseudocódigo. A primeira é que os workflows serão sequenciais, por mais que o ambiente desenvolvido permita a execução concorrente das atividades. A segunda desvantagem é que, eventualmente, é possível que um fluxo de dados criado durante a importação do plano não fosse exatamente o fluxo computado pelo planejador.

Os workflows gerados são passados ao usuário para verificação e preenchimento de parâmetros e dados de entrada e, a priori, estas limitações poderiam ser facilmente sanadas pelo usuário. Porém, para resolver essas limitações de maneira automática, foi desenvolvido um novo planejador, interno a arquitetura com três principais objetivos: evitar as chamadas externas que precisavam ser feitas ao planejador SHOP2; permitir que o processo de conversão de planos em workflows seja mais eficiente, corrigindo as limitações apresentadas; e possibilitar o desenvolvimento de mecanismos de correção de planos para corrigir, em tempo de execução, qualquer problema de execução de um workflow.

O planejador desenvolvido também é um planejador hierárquico baseado na decomposição de tarefas. Um resumo da implementação deste planejador pode ser encontrado em [21]. A integração deste planejador com a infraestrutura ainda está em fase de desenvolvimento, porém as funcionalidades básicas ligadas a produção de planos e importação na forma de workflows já estão disponíveis. Na próxima seção é apresentado um estudo de caso real da utilização da composição automática.

IV. Estudo de Caso

Nesta seção é apresentado um estudo de caso real ligado a processamento de fotos de embriões de uma mosca do gênero *Drosophila*. A motivação do problema é que existem alguns genes (co-)responsáveis por modificações nas estruturas das faixas dos embriões dessas moscas. O objetivo deste estudo de caso é analisar a morfologia dessas faixas em um conjunto de embriões que tiveram ou não modificações na expressão desses genes, identificando correlações entre a expressão gênica e a morfologia [22], [23].

Para o desenvolvimento do estudo de caso foi desenvolvida uma ontologia de domínio e componentes relacionados ao processamento de imagens. Os componentes de software, no caso métodos desenvolvidos na linguagem Java, já haviam sido desenvolvidos e foram anotados segundo essa ontologia. O conjunto composto pela ontologia mais os componentes anotados foi exportado para a linguagem do planejador e o seguinte problema foi passado para o planejador: dado o caminho do arquivo de uma imagem (*path*), deseja-se salvar as inversões do histograma (função que salva as coordenadas das faixas) e exibir o resultado final.

A Figura 7 apresenta a saída (plano) resultante da execução do planejador. Esta saída é ligeiramente diferente da saída do planejador SHOP2, mas segue o mesmo princípio: imprimir o conjunto de operadores que deverão ser executados. As constantes apresentadas nesse plano correspondem aos valores padrão de alguns dos parâmetros de entrada dos componentes.

```
retornarFigura2[path1, figura2_0]
limparFundo[figura2_0, (int)35]
destacar[(short)0, figura2_0]
suavizarMatriz[figura2_0, (int)3]
encontrarExtremos[figura2_0]
encontrarHistograma2[figura2_0, (int)5]
suavizarHistogramaIterativo[figura2_0, (int)15]
imprimirInversoes[figura2_0, c:\temp\saida.txt]
mostrarResultado[figura2_0, Resultado Final]
```

Fig. 7. Saída do Planejador Desenvolvido

O plano gerado é importado pelo sistema dando origem ao workflow apresentado na Figura 8. Neste workflow cada retângulo corresponde a um componente de software, cada seta preta corresponde a um fluxo de dados e cada seta cinza corresponde a um fluxo de controle. Na versão atual da conversão do plano para workflow, o workflow resultante sempre será sequencial.

O usuário poderá editar o workflow preenchendo ou modificando valores de entrada dos componentes, alterando fluxos de dados ou de controle ou mesmo adicionando novas atividades. A Figura 9 apresenta algumas modificações feitas pelo usuário no workflow: a inclusão de dados de entrada e a remoção de um fluxo de controle que impedia que os dois componentes mais a direita fossem executados de maneira paralela. A alteração dos parâmetros de entrada pode ser feita através de um duplo clique no componente desejado, conforme

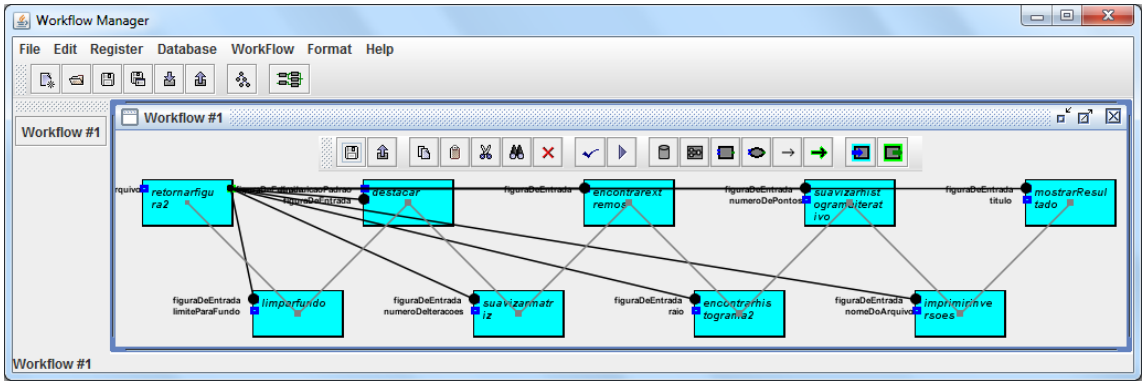

Fig. 8. Workflow Importado a Partir de um Plano

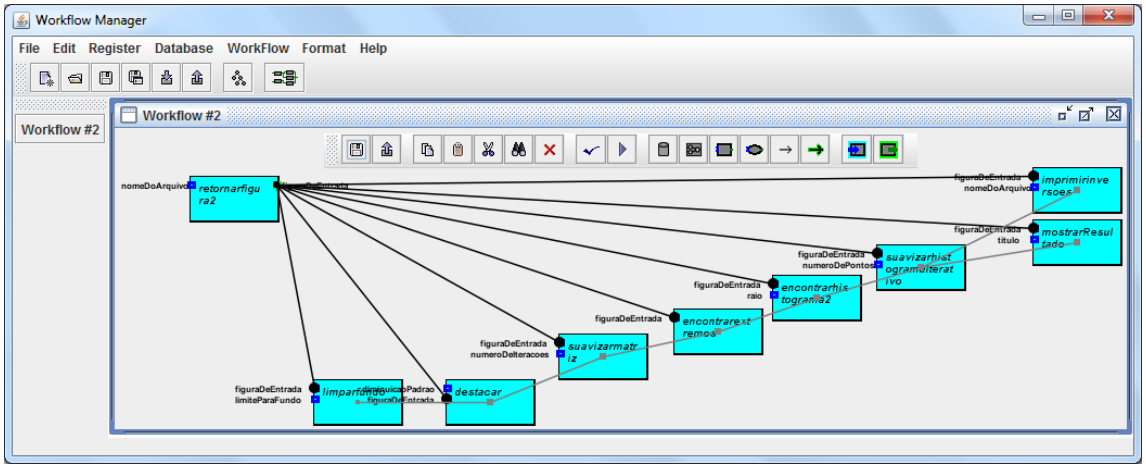

Fig. 9. Workflow Modificado pelo Usuário

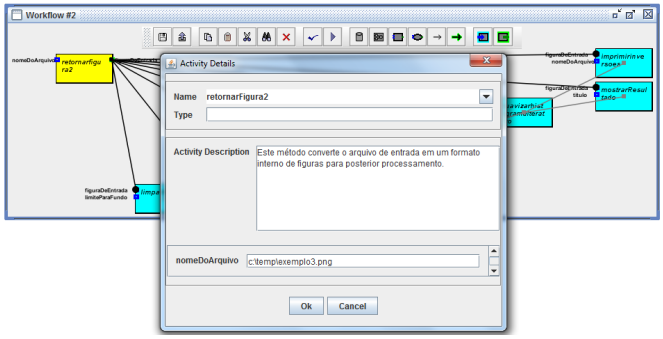

Fig. 10. Janela para Edição de Parâmetros ou Troca de Atividades

apresentado na Figura 10. Neste exemplo, o usuário selecionou como figura de entrada o arquivo "$c:\backslash temp\backslash exemplo3.png$".

Para a execução do workflow gerado há três possibilidades: (a) a execução dentro da infraestrutura desenvolvida; (b) a exportação e posterior execução como um código em Java convencional; e (c) a exportação e posterior execução do workflow como um código em Java utilizando o Motor de Execução de Workflows. Para qualquer uma das possibilidades este workflow produzirá um arquivo de saída criado pelo componente *imprimirInversoes* e gerará uma janela gráfica produzida pelo componente *mostrarResultado*. Esta janela gráfica é apresentada na Figura 11, resultante do processamento do arquivo indicado pelo usuário.

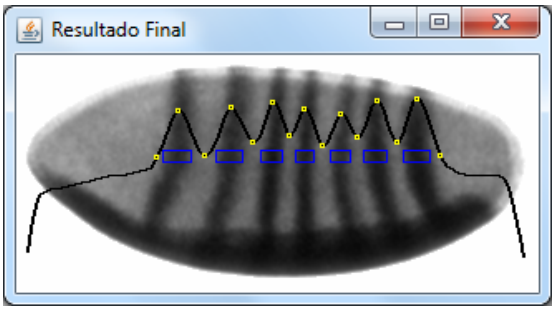

Fig. 11. Saída Produzida pela Execução do Workflow

O código resultante da exportação do workflow deste estudo de caso é apresentado na Figura 12. Conforme pode ser observado, o código gerado é sequencial e o fluxo de controle é determinado pela ordem das linhas do código.

Parte do código resultante da exportação do workflow utilizando o Motor de Execução de Workflows pode ser visto na Figura 12. Este código se beneficia do Motor de Execução para possibilitar o processamento paralelo das atividades,

```
import ProcessamentoDeImagens.Figura2;
public class CodigoEquivalente {
  public static void main(String[] args){
    Figura2 figura2_0 = Figura2.retornarFigura2("c:\\temp\\exemplo3.png");
    Figura2.limparFundo(figura2_0, (int)35);
    Figura2.destacar((short)0,figura2_0);
    Figura2.suavizarMatriz(figura2_0,(int)3);
    Figura2.encontrarExtremos(figura2_0);
    Figura2.encontrarHistograma2(figura2_0,(int)5);
    Figura2.suavizarHistogramaIterativo(figura2_0,(int)15);
    Figura2.imprimirInversoes(figura2_0,"c:\\temp\\exemplo3.txt");
    Figura2.mostrarResultado(figura2_0,"Resultado Final");
  }
}
```

Fig. 12. Código Exportado Equivalente ao Workflow

```
import Reflection.OperacaoReflection;
import Workflow.Operacao;
import Workflow.Tarefa;
import Workflow.Workflow;
public class WorkflowProcessamentoDeImagem {
  public static void main(String[] args) throws ClassNotFoundException {

    Workflow w = new Workflow();
    Operacao retornarFigura2 = new OperacaoReflection("ProcessamentoDeImagens.Figura2", "retornarFigura2");
    Tarefa t1 = new Tarefa(retornarFigura2);
    w.addExecutavel(t1);
    w.addDadosIniciais("nomeDoArquivo", t1, "c:\\temp\\exemplo3.png");

    Operacao limparFundo = new OperacaoReflection("ProcessamentoDeImagens.Figura2", "limparFundo");
    Tarefa t2 = new Tarefa(limparFundo);
    w.addExecutavel(t2);
    w.addConexao(t1, t2, "figuraDeEntrada");
    w.addDadosIniciais("limite", t2, 35);

    // .........

    Operacao imprimirInversoes = new OperacaoReflection("ProcessamentoDeImagens.Figura2", "imprimirInversoes");
    Tarefa t8 = new Tarefa(imprimirInversoes);
    w.addExecutavel(t8);
    w.addConexao(t1, t8, "figuraDeEntrada");
    w.addDadosIniciais("nomeDoArquivo", t8, "c:\\Temp\\resultado_teste");
    w.addDependencia(t8,t7);

    Operacao mostrarResultado = new OperacaoReflection("ProcessamentoDeImagens.Figura2", "mostrarResultado");
    Tarefa t9 = new Tarefa(mostrarResultado);
    w.addExecutavel(t9);
    w.addConexao(t1, t9, "figuraDeEntrada");
    w.addDadosIniciais("titulo", t9, "Resultado Final");
    w.addDependencia(t9,t8);

    w.iniciaExecucao();
  }
}
```

Fig. 13. Parte do Código Exportado que Utiliza o Motor de Execução de Workflows

respeitando-se os fluxos de dados e de controle.

V. Conclusões e Trabalhos Futuros

Este artigo apresentou uma infraestrutura para a composição automática de software baseada em três pilares. O primeiro é um repositório de componentes que podem ser métodos escritos na linguagem de programação Java; serviços Web; e aplicativos locais. O segundo pilar é a anotação semântica dos componentes utilizando-se ontologias. O terceiro é o uso de planejamento em inteligência artificial para a produção automática de planos, os quais são convertidos para workflows.

A combinação de componentes de software para a criação de novos sistemas com funcionalidades mais complexas é um desafio relevante para a ciência da computação. O reuso dos componentes possui diversas vantagens potenciais como redução do tempo de desenvolvimento do novo sistema, redução do custo, e aumento da qualidade do produto final. Porém, ainda há diversos desafios relacionados a utilização de componentes, especialmente a combinação automática destes [1], [2].

Este artigo enfrentou alguns desses desafios produzindo um sistema capaz de compor automaticamente componentes de software e exportar o resultado da combinação em diferentes formatos permitindo ao usuário final que configure, adapte ou personalize o workflow resultante.

Os principais trabalhos futuros estão ligados a extensão do planejador de forma a tratar não-determinismo, observação parcial do mundo e processamento de objetos complexos criados dinamicamente. Estas três características não são en-

contradas no planejamento clássico [6]. Além disso, pretende-se finalizar a ligação entre o planejador e o restante do sistema a fim de automatizar o tratamento de erros durante a execução de workflows com funcionalidade ligadas ao reparo de planos e replanejamento automático.

AGRADECIMENTOS

O trabalho apresentado neste artigo foi parcialmente financiado pela FAPESP (projeto Jovem Pesquisador 2009/10413-5 e Bolsa de Iniciação Científica), CNPq (Bolsa de Iniciação Científica e Bolsa Produtividade em Pesquisa 304937/2010-0), pela Pró-Reitoria de Graduação da Universidade de São Paulo (Bolsa Ensinar com Pesquisa) e pela CAPES (Bolsa de Mestrado).

REFERENCES

[1] P. Vitharana, "Risks and challenges of component-based software development," *Communications of the ACM*, vol. 46, pp. 67–72, August 2003.

[2] I. Oshri, S. Newell, and S. L. Pan, "Implementing component reuse strategy in complex products environments," *Communications of the ACM*, vol. 50, pp. 63–67, December 2007.

[3] J. C. Zuniga, J. J. Perez-Alcazar, and L. Digiampietri, "Implementation issues for automatic composition of web services," in *Proceedings of the International Workshop on Database and Expert Systems Applications,*, vol. 1. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 201–205.

[4] D. Long and M. Fox, "The 3rd International Planning Competition: Results and Analysis," *Journal of Artificial Intelligence Research*, vol. 20, pp. 1–59, 2003.

[5] J. Rao and X. Su, "A survey of automated web service composition methods," in *Proc. Of the 1st International Workshop on Semantic Web Services and Web Process Composition (SWSWPC)*, 2004.

[6] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning, Theory and Practice*. Morgan Kaufmann Publishers, Elsevier, 2004.

[7] S. Russel and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice-Hall, 2003.

[8] D. Nau, T. Au, O. Ilghami, U. Kuter, W. Murdock, D. Wu, and F. Yaman, "SHOP2: An HTN planning system," *Journal of Artificial Intelligence Research (JAIR)*, vol. 20, pp. 379–404, 2003.

[9] L. A. Digiampietri, J. Pérez-Alcázar, and C. B. Medeiros, "AI planning in web services composition: A review of current approaches and a new solution," in *VI Encontro Nacional de Inteligência Artificial, XXVII Congresso da Sociedade Brasileira de Computação (CSBC2007)*, 2007, pp. 983–992.

[10] A. Santanchè and C. Medeiros, "Self describing components: Searching for digital artifacts on the web," in *Proceedings of Brazilian Symposium on Databases (SBBD)*, 2005.

[11] T. R. Gruber, "A translation approach to portable ontologies," *Knowledge Acquisition*, vol. 5, no. 2, pp. 199–220, 1993.

[12] K. S. de Souza, J. J. Pérez-Alcázar, L. A. Digiampietri, and J. C. Z. niga, "Um estudo comparativo de frameworks para desenvolvimento de aplicações baseadas em serviços web semânticos," *Revista Eletrônica de Iniciação Científica (REIC)*, vol. 9, no. 2, p. 5, junho 2009.

[13] M. Kuzu and N. Cicekli, "Dynamic planning approach to automated web service composition," *Applied Intelligence*, pp. 1–28, 2010.

[14] O. Sapena and E. Onaindía, "Planning in highly dynamic environments: an anytime approach for planning under time constraints," *Applied Intelligence*, vol. 29, pp. 90–109, August 2008.

[15] J. Fu, "Advanced automated ai planning-based program synthesis," Ph.D. dissertation, The University of Texas at Dallas, Richardson, TX, USA, 2009, aAI3375944.

[16] D. Mitra and W. Bond, "Component-oriented programming as an ai-planning problem," in *Developments in Applied Artificial Intelligence*, ser. Lecture Notes in Computer Science, T. Hendtlass and M. Ali, Eds. Springer Berlin / Heidelberg, 2002, vol. 2358, pp. 1–6.

[17] L. A. Digiampietri, J. C. Araújo, and C. R. N. S. Éric H. Ostroski, "Combinando workflows e semântica para facilitar o reuso de software," in *Anais do Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS 2011)*, 2011, submetido.

[18] R. S. Barga and L. A. Digiampietri, "Automatic capture and efficient storage of escience experiment provenance," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 5, pp. 419–429, 2008.

[19] L. Digiampietri, J. Setubal, C. Medeiros, and R. Barga, "Traceability mechanisms for bioinformatics scientific workflows," in *AAAI Workshop*, vol. WS-07-11, 2007, pp. 26–33.

[20] L. Reeve and H. Han, "Survey of semantic annotation platforms," in *Proceedings of the 2005 ACM Symposium on Applied computing*. New York, NY, USA: ACM, 2005, pp. 1634–1638.

[21] R. S. e Freitas and J. de Jesús Pérez-Alcazar, "Planejamento hierárquico de tarefas: Implementação de um protótipo de planejador," in *Anais do 18 Simpósio de Iniciação Científica da USP*, 2010.

[22] L. P. Andrioli, L. A. Digiampietri, L. P. de Barros, and A. Machado-Lima, "Huckebein is part of a combinatorial repression code in the anterior blastoderm," *Developmental Biology*, p. 18, 2011, submetido.

[23] U. Grossniklaus, R. K. Pearson, and W. Gehring, "The drosophila sloppy paired locus encodes two proteins involved in segmentation that show homology to mammalian transcription factors," *Genes & Development*, vol. 6, pp. 1030–1051, 1992.