UNIVERSIDADE FEDERAL DO CEARÁ
CENTRO DE CIÊNCIAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ALLBERSON BRUNO DE OLIVEIRA DANTAS

CERTIFICAÇÃO DE COMPONENTES EM UMA PLATAFORMA DE NUVENS COMPUTACIONAIS PARA SERVIÇOS DE COMPUTAÇÃO DE ALTO DESEMPENHO

FORTALEZA
2017

ALLBERSON BRUNO DE OLIVEIRA DANTAS

CERTIFICAÇÃO DE COMPONENTES EM UMA PLATAFORMA DE NUVENS COMPUTACIONAIS PARA SERVIÇOS DE COMPUTAÇÃO DE ALTO DESEMPENHO

Tese apresentada ao Programa de Pós-graduação em Ciência da Computação do Departamento de Ciência da Computação da Universidade Federal do Ceará, como parte dos requisitos necessários para a obtenção do título de Doutor em Ciência da Computação. Área de concentração: Matemática Computacional.

Orientador: Prof. Dr. Francisco Heron de Carvalho Junior.

FORTALEZA

2017

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

D21c Dantas, Allberson Bruno de Oliveira.
Certificação de Componentes em uma Plataforma de Nuvens Computacionais para Serviços de Computação de Alto Desempenho / Allberson Bruno de Oliveira Dantas. – 2017.
214 f. : il. color.

Tese (doutorado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação , Fortaleza, 2017.
Orientação: Prof. Me. Francisco Heron de Carvalho Junior.

1. Computação de Alto Desempenho. 2. Métodos Formais. 3. Componentes de Software. I. Título.
CDD 005

ALLBERSON BRUNO DE OLIVEIRA DANTAS

CERTIFICAÇÃO DE COMPONENTES EM UMA PLATAFORMA DE NUVENS COMPUTACIONAIS PARA SERVIÇOS DE COMPUTAÇÃO DE ALTO DESEMPENHO.

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal do Ceará, como requisito à obtenção do título de Doutor em Ciência da Computação.

Aprovada em: 18/10/2017.

BANCA EXAMINADORA

_____
Prof. Dr. Francisco Heron de Carvalho Junior (Orientador)
Universidade Federal do Ceará (UFC)

_____
Prof. Dr. Luís Manoel Dias Coelho Soares Barbosa
Universidade do Minho (UMinho)

_____
Prof. Dr. Ricardo Massa Ferreira Lima
Universidade Federal de Pernambuco (UFPE)

_____
Prof. Dr. Lincoln Souza Rocha
Universidade Federal do Ceará (UFC)

_____
Prof. Dr. Pablo Mayckon Silva Farias
Universidade Federal do Ceará (UFC)

À minha mãe, Maria Liduina Lima de Oliveira e à minha esposa, Camylla Rachelle Aguiar Araújo Dantas.

AGRADECIMENTOS

Vou contar uma história. Imagine que, por mágica, um homem foi enviado ao núcleo externo da terra vestindo uma armadura especial que o permite lá viver por algum tempo e também abrir espaço rumo à superficie da terra. Apesar de todo o calor emanado por aquele metal derretido em altíssima temperatura, ele consegue enxergar beleza naquilo e sente vontade de ficar. Toda aquela convolução dos metais liquidos o fascinava e o fazia pensar nas infinitas maneiras de moldar aquele material, para construir o que quisesse, armas, estruturas de prédios, aviões, etc.

Apesar de querer ficar, ele lembra que tem que subir, tanto para salvar sua vida, quanto para descobrir coisas novas durante o trajeto. Então, ele utiliza sua armadura para chegar à próxima camada da terra, o manto, gastando muito esforço com isso. A princípio, o manto, ao ser composto por rochas duras e silenciosas, não pareceu tão interessante o quanto a dança dos metais no núcleo externo. Todavia, ao observar bem aquelas rochas formadas há milhares de anos, percebeu texturas e padrões que jamais imaginou que pudessem ser formados a partir de simples rochas. Aquela beleza diferente saltou-lhe os olhos e fez-lhe perceber quanto valera a pena ter parado ali, mesmo com todo o cansaço.

Ele então parte em direção à superfície, tendo que passar pela última camada, chamada de crosta. Na sua impressão, as rochas da crosta eram bem mais leves e fáceis de serem atravessadas do que as do manto. Contudo, ele, a todo momento, se perguntava se elas eram de fato mais leves, ou se era ele que havia aprendido a operar melhor sua armadura mágica. Ainda hoje ele não sabe a resposta para isso, mas a verdade é que aquele caminho sem tantos obstáculos aparentes foi de fato o mais gratificante, pois lhe permitiu refletir sobre tudo que sua jornada de volta ao mundo real havia lhe proporcionado. Ao passar da crosta e chegar na superfície, apesar de tudo continuar no mesmo lugar, o seu olhar sobre tudo havia mudado. Ele costuma dizer que apesar de ter feito o percurso por um único caminho, isso foi suficiente para mudar sua vida inteiramente. Nessa história, o homem apresentado sou eu, a superfície da terra representa o conhecimento científico e as camadas representam, respectivamente, graduação, mestrado e doutorado.

Gostaria de agradecer primeiramente a Deus, pelo dom da vida e por me proporcionar saúde e perseverança para alcançar todos os objetivos da minha vida.

Em segundo lugar, gostaria de agradecer ao meu orientador, Heron Carvalho, por ser um exemplo de professor, pesquisador e pessoa para mim. Também por estar presente em toda minha carreira acadêmica e em especial na orientação deste trabalho.

À UFC, mais especificamente ao Departamento de Computação, pela oportunidade da realização deste trabalho.

Ao professor Luís Barbosa, pela gentileza em contribuir de forma voluntária e fundamental a este trabalho.

À banca examinadora, por aceitar o convite desta defesa e por todas as valiosas

contribuições feitas durante o doutorado.

Aos amigos do grupo de pesquisa ParGO/CAD, pelas discussões e apoio que certamente foram imprescindíveis para a realização deste trabalho.

À minha mãe e avó Liduina, que me ensinou tudo na vida e me fez ser tudo o que sou hoje.

À minha tão amada esposa, Camylla, pelo amor, apoio constante e confança irrestrita em todos os momentos.

À minha mãe Alba, que me deu o dom da vida e é para mim exemplo de perseverança.

Aos meus pais Altamiro e Willdson (*in memoriam*), que foram em vida exemplos de simplicidade e dedicação.

Aos tios, irmãos, sobrinhos, padastros, sogros, avós, primos, cunhados e amigos, que tanto torceram por mim nesta empreitada.

Ao Serpro, empresa que me acolheu tão bem e permitiu liberação de horas de trabalho para a realização deste doutorado.

Aos meus chefes Adriana e José Mario, que tanto me apoiaram e incentivaram para a concretização deste sonho.

RESUMO

O desenvolvimento de aplicações de Computação de Alto Desempenho (CAD) corretas e seguras é um desafio para desenvolvedores, uma vez que tais aplicações geralmente utilizam paralelismo e executam em plataformas heterogêneas de computação paralela. A Tese de Doutorado proposta neste documento dispõe-se a apresentar a arquitetura de um mecanismo de certificação de componentes para plataformas de nuvens computacionais de serviços de computação de alto desempenho. Em particular, esse mecanismo é proposto no contexto da plataforma HPC Shelf, permitindo a construção de componentes certificados quanto a propriedades funcionais e não funcionais, os quais podem ser utilizados para compor aplicações para usuários especialistas.

Dois componentes certificadores particulares são propostos utilizando o mecanismo de certificação introduzido na Tese: SWC2 (*Scientific Workflow Certifier Component*) e C4 (*Computation Component Certifier Component*). Componentes SWC2 são utilizados para verificar propriedades formais em *workflows* na HPC Shelf. Já os componentes C4 são empregados para verificar propriedades formais em componentes de computação. Existem ainda componentes táticos, que expõem serviços de infraestruturas de verificação formal de *software* e podem ser orquestrados, por certificadores, através da linguagem TCOL (*Tactical Component Orchestration Language*), também proposta nesse trabalho.

Espera-se contribuir com o estado da arte nos seguintes pontos: em nuvens computacionais, fornecendo a primeira infraestrutura em nuvem voltada à verificação formal de *software* utilizando exclusivamente técnicas de CAD; em plataformas orientadas a componentes, provendo componentes não disruptivos que podem certificar outros de forma reflexiva; possibilitando a criação dos chamados sistemas de certificação paralela, os quais são formados por orquestrações de provadores para verificar propriedades formais; em *workflows* científicos, extraindo os principais padrões verificáveis desses workflows; e em aplicações de CAD, fornecendo um estudo sobre quais ferramentas de verificação formal de *software* se aplicam na verificação de suas propriedades.

Palavras-chave: Computação de Alto Desempenho. Métodos Formais. Componentes de *Software*.

ABSTRACT

The development of correct and safe High Performance Computing (HPC) applications is a challenge for developers, since such applications generally use parallelism and run on heterogeneous parallel computing platforms. The Doctoral Thesis proposed in this document is aimed at presenting an architecture of a component certification mechanism for cloud computing platforms of high performance computing services. In particular, this mechanism is proposed within the context of the HPC Shelf platform, allowing the construction of certified components for functional and non-functional properties, which can be used to compose applications for expert users.

Two particular certifier components are proposed using the certification mechanism introduced in this Thesis: SWC2 (*Scientific Workflow Certifier Component*) e C4 (*Computation Component Certifier Component*). SWC2 components are used to verify formal properties of workflows in HPC Shelf. In turn, C4 components are employed to verify formal properties on computation components. There are still tactical components, which expose the services of software formal verification infrastructures and can be orchestrated, by certifiers, by means of the TCOL (*Tactical Component Orchestration Language*) language, also proposed in this work.

It is expected to contribute to the state-of-the-art in the following points: in cloud computing, by providing the first cloud infrastructure focused on software formal verification using exclusively high performance computing techniques; in component-oriented platforms, by providing nondisruptive components that can certify others in a reflexive way; enabling the creation of the so-called parallel certification systems, which are formed by the orchestration of provers to verify formal properties; in scientific workflows, by extracting the main verifiable patterns in these workflows; and in high performance computing applications, by providing a study on which software formal verification tools are able to verify their properties.

Keywords: High Performance Computing. Formal Methods. Software Components.

SUMÁRIO

1 Introdução

Esta Tese de Doutorado introduz um arcabouço orientado a componentes para certificação de componentes paralelos para a HPC Shelf, uma plataforma orientada a componentes de serviços de Computação de Alto Desempenho baseada na abstração e em tecnologias de nuvens computacionais ("nuvem de componentes"). Sobre esse arcabouço, é apresentada a arquitetura de componentes certificadores cujo propósito é garantir que componentes de determinadas espécies suportadas pela HPC Shelf sejam certificados a fim de serem utilizados por aplicações sobre essa plataforma, ou seja, atendam a certas propriedades formais impostas por essas aplicações.

As propriedades em questão visam permitir a especificação de aplicações funcionalmente corretas (propriedades funcionais), seguras (propriedades de segurança) e que sempre alcancem estados desejáveis (propriedades de continuidade). A corretude se dá no sentido em que as aplicações cumprem corretamente as tarefas que se propõem a fazer. Já a segurança recai sobre o fato de que tais aplicações não alcançam estados indesejáveis. É importante salientar que a noção de segurança, no português, pode se referir a ambos os termos do inglês *security* e *safety*. *Security* é geralmente pensado em termos de proteção de dados contra acesso não autorizado. *Safety*, por outro lado, se refere a comportamentos bem definidos de programas. Como o exemplo de propriedades *safety*, pode-se citar ausência de *deadlock*, ausência de acesso a posições de memória não alocadas, dentre outras. Neste trabalho, o interesse recai sobre propriedades de segurança do tipo *safety*, doravante chamadas apenas de propriedades de segurança. Por fim, o alcance de estados desejáveis pode dizer respeito tanto a traços de execução específicos que as aplicações devem percorrer até alcançar os estados desejáveis ou a condições envolvendo outros estados, que devem ser satisfeitas para que os estados desejáveis de fato ocorram. Um exemplo comum de propriedade de continuidade é a que garante que um estado que representa a saída de um laço de repetição específico é sempre alcançado.

O projeto HPC Shelf é fruto do esforço conjunto do grupo de pesquisa em Computação de Alto Desempenho (CAD) do Programa de Pós-Graduação em Ciência da Computação (MDCC) da Universidade Federal do Ceará (UFC), em torno do qual outras teses de doutorado têm sido concluídas (SILVA, 2016; REZENDE, 2017; ALENCAR, 2017) ou se encontram em andamento. Nas próximas seções, é apresentado o contexto da área no qual se insere o projeto que tem esta Tese como resultado, quais as suas motivações, seus objetivos gerais e específicos, suas principais contribuições e a estrutura deste documento.

1.1 O Contexto da Computação de Alto Desempenho

No mais recente ranque dos 500 supercomputadores mais rápidos (Top500[1]), anterior ao fechamento desta Tese (junho de 2017), o "supercomputador" de maior desempenho, denominado Sunway TaihuLight[2], é constituído por 10.649.600 núcleos de processamento. Raras são as aplicações capazes de utilizar, de forma eficiente, todos esses núcleos durante todo o tempo de execução de uma computação. No entanto, é razoável pensar, em plataformas incluídas nesse ranque, no uso efetivo de dezenas de milhares de núcleos simultaneamente no processamento de uma tarefa. Nesse contexto, em que as aplicações para as quais foram construídas tais máquinas são de interesse estratégico, tanto estatal quanto industrial ou corporativo, a pesquisa e o desenvolvimento relacionados a técnicas de programação paralela, notadamente sofisticadas, para controle de paralelismo, sincronização e comunicação entre os núcleos de processamento de uma plataforma moderna de computação paralela de uma aplicação, são necessários. Daí surge o interesse pela área de Computação de Alto Desempenho (CAD).

CAD é uma área da Ciência da Computação voltada ao estudo de técnicas para extrair o máximo do poder computacional disponível em plataformas modernas de computação, demandadas por grandes volumes de dados a serem processados ou por algoritmos de computação intensiva, os quais podem ter seus tempos de execução encurtados quando há um bom escalonamento dos recursos computacionais para sua execução. Apesar de diversas áreas utilizarem as técnicas de CAD, seu uso intensivo é atualmente mais comum em aplicações das ciências e engenharias, dentre as quais a previsão de catástrofes naturais, mudanças climáticas e reações nucleares (BASILI et al., 2008).

As aplicações de CAD devem ser descritas de acordo com certos padrões predefinidos de paralelismo, os quais determinam a qualidade da solução e o tempo gasto para escrever o *software*. Dentre os muitos modelos de programação paralela, podem ser citados o de passagem de mensagens, cuja principal ferramenta de suporte é a biblioteca MPI *(Message Passing Interface)* (DONGARRA; OTTO; SNIR; WALKER, 1995), e a de memória compartilhada, cuja ferramenta mais disseminada é o OpenMP (OPENMP, 1997). As aplicações de CAD fazem, em geral, uso de diversas tecnologias, as quais lhe inserem em um ambiente altamente heterogêneo. Essa heterogeneidade pode se concentrar em diferentes linguagens de programação, tais como C, C++ e Fortran, em diferentes bibliotecas de execução, tais como MPI e OpenMP, diferentes arquiteturas, tais como X86 e PowerPC, e diferentes sistemas operacionais, tais como Linux ou Windows.

Como desafios para o cenário de CAD, além dessa heterogeneidade, alia-se o fato de o usuário especialista do domínio da aplicação (cientista ou engenheiro) frequentemente não possuir familiaridade com as técnicas de programação paralela, por estar interessado na descrição dos processos da aplicação em alto nível (lógica do negócio).

[1] <http://www.top500.org/>
[2] <https://www.top500.org/system/178764>

Certamente, esse cenário pode levar a atrasos no prazo e no custo do projeto. Como uma alternativa a isso, observa-se que a forma de paralelização do código pode ser abstraída da perspectiva do usuário especialista. Porém, aumentar o nível de abstração mantendo modularidade do código, eficiência, interoperabilidade e generalidade de computação paralela é uma tarefa desafiadora (STEEN, 2006; CARVALHO JUNIOR; LINS; CORREA; ARAÚJO, 2007; BERNHOLDT; NIEPLOCHA; SADAYAPPAN, 2004; POST; VOTTA, 2005), embora muito se tenha alcançado em pesquisas na academia e na indústria recentemente.

É notória a busca, por parte dos desenvolvedores de aplicações com requisitos de CAD, por ferramentas que forneçam aumento no "poder de abstração". Isto é decorrente, dentre outras coisas, do alto grau de evolução dos sistemas de *hardware*, fato que incorre em constantes procedimentos de compatibilização do código com as versões mais novas do *hardware*. Some-se a isso a disseminação de técnicas de *engenharia de software*, as quais demandam por ferramental de alto nível.

Diante do exposto, faz-se necessário o desenvolvimento de ferramentas que potencializem a produtividade no desenvolvimento de aplicações que demandam pela computação paralela, diminuindo o esforço no tratamento dos requisitos de processamento paralelo eficiente. Como uma alternativa viável a essa questão, ressalta-se a técnica de Programação Orientada a Componentes, a qual é um ramo da engenharia de *software* que visa a criação de sistemas complexos, compostos por módulos independentes. Como opção para incrementar o nível de abstração das aplicações de CAD, é possível criar componentes que encapsulem a lógica de comunicação paralela, abstraindo detalhes de baixo nível do usuário especialista. Tais componentes são chamados de *componentes de alto desempenho*, uma vez que estão neles concentrados a parte computacionalmente crítica da aplicação, a qual influencia preponderantemente o seu tempo de execução.

1.2 Componentes de Alto Desempenho

A Programação Orientada a Componentes permite a construção de programas a partir de componentes de software pré-construídos, os quais constituem blocos de código reusáveis, auto-contidos e auto-executáveis (WANG; QIAN, 2005). Esses componentes devem seguir certos padrões predefinidos de *interface*, conexões, versionamento e implantação. Além disso, a criação de componentes pode se dar pela composição de outros componentes menores e a criação de aplicações baseadas em componentes é feita por uma colagem adequada de componentes uns aos outros.

Componentes podem ter várias formas e tamanhos, partindo de componentes que servem a pequenas aplicações e podem ser comprados de corretores de componentes na *internet*, até componentes ditos de alta granularidade, os quais absorvem uma lógica de negócio complexa de uma aplicação comercial. A noção de componente de *software* surgiu

inicialmente na indústria, para refletir na produção de *software* a maturidade alcançada por componentes mecânicos, sobretudo os advindos da indústria automotiva.

Com base em suas diversas vantagens, os componentes de software logo foram adotados pela comunidade científica, a qual propôs diversos modelos, inclusive alguns que versam sobre aplicações com requisitos de CAD e serão abordados a seguir.

O primeiro deles e o de maior difusão foi o CCA *(Common Component Architecture)* (ARMSTRONG et al., 2006), inspirado no padrão de interoperabilidade CORBA. Ele foi proposto por um conjunto de pesquisadores de diversas partes do mundo, contudo coordenados pelo Departamento de Defesa do governo dos EUA (DOE), visando superar limitações das abordagens orientadas a componentes comumente aplicadas à computação científica. Fundamentalmente, o CCA é uma especificação de um padrão de programação orientado a componentes e de uma *interface* através da qual os componentes enxergam um arcabouço de suporte subjacente. As *interfaces* dos componentes são especificadas em uma linguagem específica de domínio, SIDL (*Scientific Interface Definition Language*) (CLEARY; KOHN; SMITH; SMOLINSKI, 1998), sendo definidas de uma maneira independente das linguagens de programação em que foram escritos os componentes. Os componentes podem se comunicar através de portas do tipo *uses/provides*, mas o CCA ainda permite a ligação direta entre eles, sem a necessidade de nenhuma mediação, a fim de evitar sobrecarga nas trocas de informações entre eles. Observa-se que o CCA não é um modelo inerentemente paralelo, ou seja, não especifica diretamente que os requisitos de alto desempenho do sistema devem ser alcançados através de técnicas de paralelismo, deixando isso, entretanto, a cargo dos arcabouços compatíveis com ele. Dentre os vários arcabouços compatíveis com o CCA, o que mais tomou notoriedade foi o CCAffeine (ALLAN et al., 2002), que utiliza a ferramenta BABEL/SIDL (EPPERLY et al., 2011) para a conexão entre as portas dos componentes e que introduziu a noção de regimento de componentes (*cohort*), para suporte a paralelismo com memória compartilhada.

Já o modelo Fractal (BRUNETON et al., 2006) define um modelo de componentes hierárquico (componentes podem conter componentes aninhados), de forma a permitir a visualização da aplicação sob diferentes níveis de abstração. Permite também a noção de componentes compartilhados, de forma que um componente aninhado possa pertencer ao mesmo tempo a mais de um componente composto. Por fim, oferece funcionalidades de introspecção de componentes, reconfiguração dinâmica e a separação da aplicação em vários "interesses". Tais interesses surgem como um meio para separar pedaços de código ou entidades de execução de acordo com o serviço que estão a cumprir, como tornar a aplicação configurável, segura e disponível. Essa noção de interesse também se aplica a cada um dos componentes presentes na hierarquia do componente composto.

O GCM *(Grid Component Model)* (BAUDE et al., 2009), por sua vez, consiste em uma extensão do modelo Fractal. A grande novidade trazida por esse modelo recai sobre a noção de *interfaces* coletivas, as quais permitem diversos padrões de interações

entre portas *uses* e *provides*. Por exemplo, em uma operação de *multicast*, pode-se ter uma porta *provides* que se conecta a diversas portas *uses*. Essa possibilidade de interação permite chegar a qualquer modelo de interação que se deseje, bastando apenas compor componentes hierarquicamente da forma apropriada. Esse modelo destina-se a aplicações que fazem o uso de infraestruturas de grades computacionais, as quais são compostas por recursos computacionais heterogêneos, distribuídos geograficamente e que compartilham recursos a fim de atender aos interesses da aplicação.

Ressalta-se também o modelo de componentes Hash (CARVALHO JUNIOR; LINS, 2005), o qual tem como proposta o suporte a componentes hierárquicos inerentemente paralelos, voltados à implantação, composição e execução de plataformas de computação paralela de memória distribuída, separando o interesse de paralelismo do interesse do negócio dos componentes de interesse da aplicação. A plataforma de referência desse modelo é o HPE *(Hash Programming Environment)* (CARVALHO JUNIOR; LINS; CORREA; ARAÚJO, 2007; CARVALHO JUNIOR; REZENDE, 2013).

O HPE é voltado para plataformas de *cluster computing* e sua implementação foi feita, visando uma maior interoperabilidade, utilizando a plataforma CLI *(Common Language Infrastructure)* (ECMA International, 2006), cujas principais implementações são Mono[3] e .NET[4]. Em 2010, o HPE foi compatibilizado com o CCA (CARVALHO JUNIOR; CORREA, 2010), de forma a tornar possível sua comparação com outras plataformas de componentes compatíveis com tal modelo, bem como permitir a comparação do HPE com outras plataformas que implementam Fractal e GCM, uma vez que existem trabalhos que comparam plataformas CCA, Fractal e GCM (MALAWSKI et al., 2007).

Esta compatibilidade permitiu aferir o grau de impacto da expressividade incorrida pelo modelo Hash, oferecendo um modelo de componentes inerentemente paralelos ao próprio CCA. Tais estudos evidenciaram que, para representar padrões gerais de paralelismo, o desempenho do HPE se apresenta similar ao código paralelo monolítico como o uso da biblioteca MPI (CARVALHO JUNIOR; REZENDE, 2013). Por fim, vale salientar que a arquitetura do HPE permitiu que os usuários especialistas não tenham que se preocupar com questões de paralelismo advindas da aplicação, se preocupando apenas em compor aplicações através de componentes. Os componentes, então, devem ser fornecidos por outro usuário do sistema, o desenvolvedor de componentes, o qual, por sua vez, deve ser conhecedor das técnicas de paralelismo.

1.3 Computação em Nuvem e Computação de Alto Desempenho

A arquitetura SOA (*Service Oriented Architecture*) é um estilo arquitetural de *software* cujo princípio fundamental é que as funcionalidades da aplicação sejam disponibilizadas através de serviços (BROWN; HAMILTON, 2006). Preza que os serviços sejam conectados

[3] <http://www.mono-project.com/>
[4] <http://www.microsoft.com/net/>

por um "barramento de serviços", o qual disponibiliza *interfaces*, ou contratos, que podem ser acessados através de um protocolo de comunicação. A SOA se baseia na computação distribuída, utilizando o padrão *request/reply* para estabelecer a comunicação entre os sistemas clientes e os sistemas que implementam os serviços.

A SOA é o marco inicial de iniciativas que possibilitaram o advento da "Computação de Serviços" (*Service Computing*)[5], um ramo da ciência da computação que cobre todo o ciclo de vida da prestação de serviços computacionais, como componentização de negócio, criação, implantação, descoberta, composição, monitoramento de serviços, dentre outros. O alvo principal da Computação de Serviços é fornecer serviços de computação que executem serviços de negócio mais eficientes e robustos.

A Computação de Serviços vem ganhando destaque nos últimos anos, impulsionada, sobretudo, com a difusão da tecnologia de serviços *web* (*web services*). Nesta tecnologia, um serviço remoto pode ser criado utilizando qualquer linguagem de programação e disponibilizado pela *internet*. Qualquer cliente que deseje usufruir dos serviços ofertados deve apenas se dirigir à sua *interface* (um arquivo XML), para descobrir suas operações, e, uma vez sabendo qual operação executar e a localização do serviço, pode "consumi-lo". Outros fatores que também ajudaram nessa difusão foram o barateamento crescente observado nos equipamentos de *hardware*, o incremento nas velocidades de comunicação na *internet* e a facilidade oferecida pelos arcabouços de programação em criar serviços *web* a partir de códigos disponíveis.

Componentes de software também podem ser acessados remotamente, fornecendo seus serviços computacionais através de serviços *web*. No caso de componentes de CAD, o processamento paralelo intensivo continuaria sendo feito com técnicas e ferramentas de paralelismo convencionais, apenas necessitando expor a *interface* do componente através de contratos XML e estabelecer uma conexão *reply/request* com o cliente através de trocas de arquivos XML.

Outras soluções de software no âmbito de Computação de Serviços foram propostas nas últimas décadas. Dentre elas, cita-se a Computação em Nuvem (*Cloud Computing*), a qual se destina a expor os serviços, sejam eles recursos de *hardware* ou *software*, de uma forma transparente, compondo uma "nuvem" de funcionalidades.

Como um cenário de exemplo recorrente para nuvens computacionais tem-se o processo de geração e distribuição de energia elétrica, onde os clientes se preocupam somente em atestar que seus equipamentos são compatíveis com as características nominais da rede de distribuição. O pagamento é mensal, referente somente o que foi gasto. Melhorias contínuas na produção e transmissão são transparentes ao cliente.

O NIST *(National Institute of Standards and Technology)*[6] é um instituto do departamento de comércio do governo dos EUA que busca trabalhar juntamente com a

[5] <http://tab.computer.org/tcsc/>
[6] <http://www.nist.gov/>

indústria para produzir tecnologias aplicadas, métricas e padrões. Segundo o NIST, a definição de Computação em Nuvem consiste em um modelo que permite acesso em rede conveniente e por demanda a um conjunto compartilhado de recursos computacionais, incluindo redes, servidores, armazenamento, aplicações e serviços, que podem ser rapidamente provisionados e liberados com mínimo esforço de gerenciamento ou interação com o provedor do serviço (MELL; GRANCE, 2011).

Esse modelo de nuvem deve garantir alta disponibilidade e é composto por cinco características essenciais: serviço por demanda, acesso de rede amplo, um *pool* de recursos, elasticidade rápida e medição de serviços; três modelos de serviços, os quais serão detalhados adiante: SaaS *(Software as a Service)*, PaaS *(Platform as a Service)*, IaaS *(Infrastructure as a Service)*; e quatro modelos de implantação: redes privadas (fechadas à organização), redes comunitárias (compartilhadas por várias organizações), redes públicas (abertas para o público em geral) e redes híbridas (composição das anteriores). Tais nuvens devem usar três tecnologias: redes de longo alcance de alta velocidade, computadores servidores econômicos e serviços de virtualização de *hardware*.

A seguir, são resumidos os três modelos de serviços previstos pelo NIST para Computação em Nuvem:

- **SaaS**, no qual o cliente pode fazer uso de diversas aplicações previamente disponíveis, ofertadas através de diversos tipos de *interfaces*, as quais podem ser acessadas por navegadores *web* ou dispositivos móveis;
- **PaaS**, no qual o cliente tem acesso a um conjunto de recursos de programação, como linguagens, bibliotecas e serviços, com os quais ele pode criar aplicações e monitorar sua implantação e configuração na nuvem. O cliente pode ainda adquirir acesso a aplicações já disponíveis na nuvem e incorporá-las ao seu espaço de trabalho (*workspace*);
- **IaaS**, no qual um conjunto de recursos computacionais é oferecido aos clientes, como serviços de armazenamento, redes de alta velocidade, memória e processamento (*batch* e gráfico). É responsabilidade do cliente implantar *softwares* próprios, ou sistemas operacionais, sobre a infraestrutura que foi alugada. A responsabilidade das manutenções de *hardware* é exclusiva do provedor da nuvem.

Diversas estruturas de nuvens computacionais vêm obtendo destaque no meio comercial, sob a modalidade de Nuvens Públicas. A maior e mais bem sucedida delas é a nuvem EC2 (*Amazon Elastic Computing Cloud*[7]), a qual disponibiliza um serviço de computação IaaS redimensionável na *web*. Fornece uma *interface* de serviço simples, com o controle completo dos recursos computacionais contratados. Garante um fluxo otimizado para obtenção e inicialização de novas instâncias de servidor, sendo possível escalonar a capacidade contratada para mais ou para menos, a partir do momento em que os requisitos de computação do cliente tomam outra configuração. Implementa o

[7] <http://aws.amazon.com/pt/ec2/>

serviço "pay-as-you-go", de forma que seja necessário somente o pagamento do recurso que for utilizado. Fornece ainda, aos desenvolvedores, ferramentas para construir aplicações tolerantes a falhas e isolá-las quando isso ocorrer.

Destaca-se ademais a nuvem Microsoft Azure[8], que consiste em um conjunto de ferramentas integradas, modelos pré-compilados e serviços gerenciados, voltados, sobretudo, a aplicações empresariais, móveis, *web* e IoT (*Internet* das Coisas). Aponta, como uma de suas principais vantagens, ser uma plataforma aberta e flexível, dando suporte a uma gama de sistemas operacionais, linguagens de programação, arcabouços, ferramentas, bancos de dados, dentre outros. Fornece ainda facilidades que visam tornar possível a compatibilização dos serviços de TI já existentes na organização com os serviços da nuvem, sem a estrita necessidade de todo o serviço ser migrado para a nuvem.

Como último exemplo de nuvens PaaS, ressalta-se a nuvem Google App Engine[9], a qual é uma plataforma voltada à construção de aplicações *web* escaláveis com *backends* móveis. Tem a capacidade de escalar aplicações automaticamente, atuando em resposta a um aumento de tráfego. Oferece também serviços de balanceamento de carga e descoberta de vulnerabilidades nas aplicações, prometendo baixíssimas taxas de falsos positivos.

Observa-se também a existência de ferramentas de código aberto que permitem a criação de nuvens computacionais. Dentre elas, podem ser citados os projetos Open Nebula[10] e Open Stack[11]. O primeiro provê soluções flexíveis para o gerenciamento de *data centers* virtualizados, nas categorias de nuvens IaaS privadas, públicas e híbridas. Apresenta os seguintes premissas que orientaram a sua construção: arquitetura, *interfaces* e código abertos; possibilidade de aplicação a qualquer tipo de *data center*; interoperabilidade e portabilidade para evitar aprisionamento tecnológico por parte de fornecedores; controle completo sobre os serviços da nuvem; simplicidade e facilidade de implantação, operação e uso; e alta eficiência.

O Open Stack, por sua vez, fornece um sistema operacional que se propõe a controlar um grande conjunto de recursos de computação, armazenamento e rede, fornecidos por um *data center*. Consiste, na realidade, de um conjunto de subprojetos relacionados, dentre os quais o usuário deve escolher quais farão parte dos seus serviços essenciais (*core services*). Exemplos de *core services* são: o NOVA, para serviços de computação; o CINDER, para serviços de armazenamento; o NEUTRON, para serviços de rede; dentre outros. Existem ainda serviços opcionais, tais como o SAHARA, para processamento de dados, o ZAQAR, como serviço de filas, o HEAT, para orquestração de serviços, o MANILA, como sistema de arquivos compartilhado, dentre outros.

Com o sucesso das nuvens comerciais, a tecnologia de nuvens computacionais logo despertou interesse de usuários, pesquisadores e provedores de serviços de computa-

[8] <http://azure.microsoft.com/pt-br/>
[9] <http://appengine.google.com/>
[10] <http://www.opennebula.org/>
[11] <http://www.openstack.org/>

ção de alto desempenho. As iniciativas de pesquisa em computação de alto desempenho em nuvem abrangem os três modelos de serviços, SaaS, PaaS e IaaS. No modelo SaaS, pode-se apontar o oferecimento de aplicações com *interfaces* de alto nível via navegadores *web* voltadas à execução de soluções computacionalmente intensivas de problemas específicos de domínio das ciências e engenharias (CHURCH; WONG; BROCK; GOSCINSKI, 2012). Já no modelo PaaS, podem ser citados ambientes de programação em nuvem equipados com as mais diversas bibliotecas voltadas à construção de aplicações com requisitos de CAD, tais como MPI, OpenMP, CUDA (NICKOLLS, 2007), etc. O exemplo mais promissor dessas nuvens é o Aneka, que é uma solução computacional de origem acadêmica baseada na plataforma .NET e mantida pela empresa Manjrasoft (SUKUMAR; VECCHIOLA; BUYYA, 2010). Por fim, no modelo IaaS, apontam-se provedores de infraestruturas de computação de alto desempenho, ou seja, conjuntos de plataformas de computação paralela, os quais visam oferecer infraestruturas para execução direta de programas paralelos, possuindo interesse especial de cientistas computacionais e engenheiros, tanto da academia quanto da indústria (ZASPEL; GRIEBEL, 2011).

Tendo em vista propor contribuições ao cenário das nuvens de serviços de CAD, o grupo de pesquisa o qual o autor desta Tese integra decidiu construir um sistema de computação em nuvem voltado para aplicações de CAD, baseado em componentes inerentemente paralelos. Essa nuvem busca ofertar as facilidades de abstração e de desempenho satisfatório tal qual faz o HPE. A esse projeto, deu-se o nome de HPC Shelf, sucintamente abordada na seção que se segue.

1.4 A Nuvem HPC Shelf

A HPC Shelf é uma plataforma de oferta de serviços de Computação de Alto Desempenho em nuvem, cujo objetivo final é oferecer, a usuários especialistas de um certo domínio de interesse, aplicações através das quais podem especificar problemas e executar soluções computacionais para esses. Tais soluções computacionais são chamadas doravante de *sistemas de computação paralela*, construídos a partir da composição e orquestração de componentes paralelos compatíveis com o modelo Hash, que representam algoritmos, estruturas de dados, plataformas de computação paralela e propriedades não-funcionais.

A plataforma HPC Shelf tem como base os seguintes elementos arquiteturais:

- Front-End: elemento responsável por possibilitar às aplicações o acesso aos serviços da nuvem. Atualmente, o arcabouço SAFe (*Shelf Application Framework*) faz esse papel, constituindo um sistema gerenciador de *workflows* para orquestração de componentes paralelos (SILVA; CARVALHO JUNIOR, 2016);
- Core: elemento responsável pelo catálogo de componentes disponíveis para a composição do sistema de computação paralela de uma aplicação. Implementa um sistema de resolução de contratos contextuais, para descobrir que componentes, dependendo

de um *contexto* fornecido pela aplicação, são adequados para compor um sistema de computação paralela. Possui também a função de instanciar componentes, o que compreende implantar o componente na plataforma de computação virtual que o acolherá e o tornar pronto para executar;

- Back-End: encarregado de intermediar a comunicação do Core com uma infraestrutura computacional, sendo capaz de gerenciar e monitorar o uso dos recursos físicos na instanciação de plataformas virtuais de computação paralela.

Os atores da nuvem HPC Shelf compõem as partes interessadas no provimento e consumo de serviços na nuvem. Podem ser agrupados de acordo com seus interesses e responsabilidades:

1. *Usuários especialistas*: buscam utilizar aplicações disponíveis na nuvem para resolver problemas específicos do seu domínio;
2. *Provedores de aplicações*: estão interessados em prover aplicações capazes de sintetizar sistemas de computação paralela formados pela orquestração de componentes disponíveis no Core a partir das entradas fornecidas pelos especialistas;
3. *Desenvolvedores de componentes*: são responsáveis por construir componentes da maneira mais otimizada possível, considerando os recursos computacionais disponíveis na plataforma de computação paralela alvo, para serem utilizados em sistemas de computação paralela;
4. *Mantenedores de plataforma*: são responsáveis por gerenciar as plataformas de computação paralela (infraestrutura) da nuvem, a partir das quais criam plataformas virtuais, disponibilizadas utilizando a abstração de componentes.

Com relação à HPC Shelf, o interesse deste trabalho recai sobre o oferecimento de garantias a provedores de aplicações de que componentes críticos usados por sua(s) aplicação(ões) na constituição de sistemas de computação paralela, os quais podem ser desenvolvidos por terceiros (desenvolvedores de componentes), satisfazem certas propriedades, descritas em seus contratos. Doravante, chamaremos de componentes certificados aqueles para os quais são definidos, em seus contratos, propriedades que foram satisfeitas por suas implementações.

1.5 Workflows Científicos e Computação de Alto Desempenho

Computações de longa duração são comuns em aplicações científicas. Devido a esse tipo de computação, o usuário especialista é obrigado, muitas vezes, a segmentar a execução da aplicação, para que, em pontos estratégicos, possa tomar decisões com base nos resultados intermediários, e definir qual será o próximo passo de processamento a ser executado. Certamente, modelos automáticos que permitam guiar os passos da computação em tempo de execução sem a intervenção manual do especialista podem configurar alternativas viáveis.

Como exemplo desses modelos, cita-se a tecnologia de Sistemas de Gerencia-

mento de *Workflows* (SGWf). Desenvolvida para aplicações emergentes no meio comercial, essa tecnologia define um fluxo de tarefas que compõem um processo de negócio de uma organização. Em geral, tais tarefas são elaboradas utilizando *templates* e aplicadas a situações concretas quando necessário. Partem de *scripts* simples que atuam sobre recursos locais até rotinas complexas de orquestração de serviços disponíveis na *internet*. Essas últimas estão intimamente ligadas à evolução das técnicas de Engenharia de *Software* nesse contexto e ao aumento da velocidade dos dispositivos de comunicação.

Devido a suas funcionalidades serem voltadas para a programação com e para reuso, esse modelo logo foi absorvido pela comunidade científica, sob o nome de Sistemas de Gerenciamento de *Workflows* Científicos (SGWfC), a qual está interessada em orquestrar componentes para resolver problemas de domínio dos especialistas. Exemplos dessas iniciativas são Askalon[12] (QIN; FAHRINGER; PLLANA, 2006), BPEL Sedna (WASSERMANN et al., 2007), Kepler[13] (LUDÄSCHER et al., 2006), Pegasus[14] (DEELMAN et al., 2005), Taverna[15] (WOLSTENCROFT et al., 2013) e Triana[16] (HARRISON; TAYLOR; WANG; SHIELDS, 2008). No âmbito de CAD, é possível orquestrar componentes de alto desempenho, como os componentes Hash, configurando os componentes como serviços e programando as interconexões através de *workflows*.

O arcabouço SAFe fornece um conjunto de classes e padrões de projeto que permitem a confecção de aplicações baseadas em *workflows*, os quais, por sua vez, são formados por componentes paralelos disponibilizados como serviços (SILVA; CARVALHO JUNIOR, 2016). Na HPC Shelf, tanto aplicações quanto *workflows* são representados através da abstração de componentes.

1.6 *Softwares* Certificados e Computação de Alto Desempenho

Diferentemente de outras áreas do conhecimento, como as engenharias, a produção de *software*, muitas vezes, não fornece algum tipo de garantia significante de que o artefato de *software* construído está de acordo com a sua especificação (SHAO, 2010). A maioria das corporações e agências governamentais investem seus esforços em mecanismos de descoberta de erros de *software* (*bugs*), deixando de lado a utilização de processos que levem à construção de *softwares* seguros.

Um dos motivos desse desinteresse é a complexidade inerente aos programas. Cada linha do código configura um erro em potencial e programas, sobretudo paralelos, podem levar a um número elevado de estados, fatos que tornam difícil prever o comportamento de uma aplicação antes que ela execute. Essa dificuldade é ainda mais evidente quando se está lidando com um ambiente computacional heterogêneo, como o de CAD, o

[12] <http://askalon.org/>
[13] <http://kepler-project.org/>
[14] <http://pegasus.isi.edu/>
[15] <http://www.taverna.org.uk/>
[16] <http://www.trianacode.org/>

qual conta com uma diversidade de linguagens de programação e plataformas computacionais.

Com base nesse panorama, técnicas para e escrita de *softwares certificados* foram propostas pela comunidade científica, visando a construção de programas confiáveis. Se baseiam na definição de *software* certificado, o qual consiste em um programa de computador associado a uma prova formal, verificável por computador, de que esse programa atende a uma certa especificação. Tomaram notoriedade nos anos 90 após a proposição da técnica *Proof-Carrying Code* (Código Carregando Prova) (NECULA, 1997) por Necula. Tal técnica consiste, em linhas gerais, de um provedor de código compilar um código provido por ele e gerar, nesse momento, também uma prova formal de que o código compilado atende a uma especificação definida *a priori*. O código compilado e sua prova comporão um binário *Proof-Carrying Code* e percorrerão diversos consumidores de código, os quais, antes de usar o código, verificarão se ele realmente corresponde à prova que carrega.

Dentre as técnicas que podem ser empregadas para a construção de *softwares* certificados, aponta-se a **Verificação Formal de *Software***, que é um ramo da área de Métodos Formais para a Ciência da Computação. Nela, um modelo matemático abstrato do sistema deve ser construído, e, a partir dele, podem-se obter provas de que existe uma correspondência entre o modelo e o *software* a ser construído. Existem diferentes técnicas para a verificação formal de *software*, das quais ressaltam-se a *Model Checking* (Verificação de Modelos) (BAIER; KATOEN, 2008) e a *Verificação Dedutiva de Programas* (APT; BOER; OLDEROG, 2009). A primeira se apoia em algoritmos para explorar o espaço dos estados alcançáveis de um programa, verificando propriedades sobre esses estados. A segunda, no que lhe diz respeito, utiliza uma abordagem para verificar propriedades sobre programas baseada no uso de provadores de teoremas, automáticos ou interativos.

Apesar do avanço das técnicas de verificação nas últimas décadas, a aplicação delas na computação de alto desempenho ainda é incipiente. Isso posto, pretende-se, neste trabalho, investigar a aplicação de técnicas de verificação formal de programas para construir *softwares* (componentes) certificados para a HPC Shelf.

Existem diversos tipos de propriedades que podem ser verificadas. Dentre elas, elencam-se as propriedades funcionais, as propriedades de continuidade e as propriedades de segurança. As primeiras pretendem afirmar que um determinado programa implementa a funcionalidade a qual se propôs a fazer. As segundas, por conseguinte, visam garantir que o programa eventualmente alcançará um certo estado de computação desejável. Por fim, as terceiras visam garantir que o programa nunca alcançará um estado de computação indesejável. Existe uma interseção entre esses conjuntos de propriedades. Entretanto, quando uma propriedade de continuidade ou de segurança fizer parte da especificação funcional do programa, a propriedade será considerada como sendo do tipo funcional.

A título de ilustração, considere o cenário que se segue. Tomem-se três compo-

nentes da HPC Shelf: A, o qual representa uma aplicação qualquer; W, o qual representa o *workflow* que compõe A; e C, que representa um componente do tipo Computação que está presente na orquestração imposta por W e implementa operações sobre matrizes utilizando paralelismo. Como exemplo de propriedades funcionais, poderia-se estar interessado em provar que as operações de C condizem com suas respectivas definições matemáticas. Por sua vez, como exemplo de propriedades de continuidade, pode ser interessante para a aplicação a garantia de que a orquestração de W sempre executa uma certa ação computacional que modifica o estado de uma estrutura de dados crítica ou que sempre executa uma certa ação computacional que indica que o processamento pode ser encerrado. Como exemplo de propriedades de segurança, poderia-se estar interessado em garantir que a orquestração de W não leva a um *deadlock* ou que W, ao invocar C, o fará depois que C foi implantado na plataforma de computação virtual alvo. Mais ainda, poderia-se tentar provar que C, caso seja um programa paralelo, não leva a um *deadlock*. Por fim, poderia-se estar interessado em garantir que C não faz nenhum acesso a posições de memória não alocadas, fato que, caso ocorra, no caso de uma computação de longa execução, seria extremamente indesejável, pois a aplicação abortaria e o especialista teria provavelmente que rodar todo o experimento novamente.

1.7 Verificação Formal de *Software* e Computação em Nuvem

Em geral, a aplicação de técnicas de verificação formal de *software*, através de suas ferramentas associadas, demanda muito esforço manual e muito poder computacional. Como uma forma de conter esse problema, diversas técnicas voltadas à redução da complexidade da verificação, tais como raciocínio composicional (KUPFERMAN; VARDI, 1998) e transformações entre modelos (CLARKE; GRUMBERG; LONG, 1994), foram propostas. Essas técnicas permitiram o aumento da aplicabilidade da verificação formal, mas ao requererem intervenção manual, impediram que a verificação formal pudesse ser aplicada a sistemas de *software* em larga escala.

Com base nesse cenário, Schaefer e Sauer (SCHAEFER; SAUER, 2011) propuseram a chamada Verificação como Serviço (*Verification as a Service - VaaS*). A VaaS se baseia em *workflows* de verificação, que podem ser executados em ambientes de computação orientados a serviço, provendo tanto análises quanto a redução da complexidade dos serviços de verificação, através de abstrações que visam exigir menor intervenção do usuário. Assim, a ideia consiste em, partindo de um sistema e propriedades a serem verificadas, um *workflow* de verificação possa ser desencadeado, invocando tarefas de verificação em uma ordem adequada. Os autores argumentam ainda que a Computação em Nuvem, ao prover um ambiente de computação orientado a serviço e com uma infraestrutura de processamento compartilhada, extremamente poderosa, e provida sobre uma *interface* abstrata, torna-se a alternativa mais viável para se implementar o conceito de

verificação como serviço.

Tomando como base a verificação como serviço em nuvem, há muito a ser explorado considerando as diferentes possibilidades de integração de serviços na nuvem. Em especial, até onde se sabe, não existe infraestrutura VaaS construída exclusivamente utilizando técnicas de CAD, o que abre precedente para a investigação em diversas frentes, como paralelismo com memória distribuída, memória compartilhada, aceleradores computacionais, etc. Esse aspecto é investigado nesta Tese.

1.8 Proposta da Tese: Certificação de Componentes na HPC Shelf

Esta Tese tem como principal contribuição um arcabouço de verificação como serviço para componentes paralelos sobre a HPC Shelf (DANTAS; CARVALHO JUNIOR; BARBOSA, 2017a; DANTAS; CARVALHO JUNIOR; BARBOSA; PROENÇA, 2017). A implementação desse arcabouço demanda a criação de duas novas espécies de componentes: *certificadores* e *táticos*.

Os componentes certificadores, quando ligados a outros componentes da HPC Shelf, possuem a habilidade de acessar, de forma reflexiva, as implementações desses componentes e verificá-las, utilizando técnicas de verificação formal de *software*, com relação a um conjunto de propriedade de interesse das aplicações que demandam por esses componentes, tornando-os assim *certificados*.

Os componentes táticos [17], por sua vez, atuam de forma auxiliar aos componentes certificadores, implementando o acesso, de fato, às técnicas de verificação e infraestruturas de verificação associadas, as quais são compostas sobretudo por provadores de teoremas ou *model checkers*, os quais são responsáveis por confrontar os programas com as respectivas propriedades formais. Os componentes certificadores e os componentes táticos, como quaisquer componentes da HPC Shelf, são inerentemente paralelos, podendo ser programados utilizando as diversas técnicas de programação de CAD, visando explorar o máximo desempenho da infraestrutura da nuvem durante o processo de certificação de componentes. Até onde se sabe, a abordagem apresentada neste trabalho constitui a primeira infraestrutura de verificação como serviço em nuvem implementada utilizando exclusivamente técnicas de computação de alto desempenho.

Outro conceito introduzido é o de *sistema de certificação paralela*. Sistemas de certificação paralela são as contrapartes de certificação dos sistemas de computação paralela na HPC Shelf. Em um sentido mais geral, um sistema de certificação paralela é composto por um componente a ser certificado, um componente certificador e um conjunto de componentes táticos conectados ao componente certificador. A orquestração executada no sistema de certificação paralela é definida pelo componente certificador,

[17] O termo "tático" faz alusão a procedimentos chamados de táticas em provadores de teoremas interativos. Nesses provadores, o usuário pode aplicar diversas táticas (estratégias de prova) para se provar os objetivos ou construir novos objetivos a partir dos existentes.

que guia as ativações das ações dos componentes táticos mediante um fragmento escrito em uma linguagem específica de domínio, chamada de TCOL, que quer dizer ***Tactical Component Orchestration Language*** (Linguagem de Orquestração de Componentes Táticos), também proposta na Tese. Através dessa orquestração, os componentes táticos podem ser resolvidos, implantados, instanciados e liberados de acordo com as necessidades do processo de certificação. Observe que componentes certificadores fazem o papel dos componentes *workflow* nos sistemas de computação paralela, e que os componentes táticos fazem os papel dos componentes de solução.

A criação de componentes certificadores e táticos exige conhecimento técnico na área de verificação formal de *software*. Dessa forma, é natural a proposição de um novo ator na nuvem, voltado exclusivamente à criação desses componentes: o *Certificador de Componentes*. Ele é um especialista em métodos formais para a ciência da computação e possui conhecimento sobre técnicas e ferramentas de computação de alto desempenho.

Nesta Tese, são propostas também duas famílias de componentes certificadores. A primeira é chamada de SWC2, que vem do inglês ***Scientific Workflow Certifier Component*** (Componente Certificador de *Workflow* Científico), e seus componentes têm o propósito de verificar propriedades de continuidade e de segurança em *workflows* criados por usuários provedores de aplicação na nuvem (DANTAS; CARVALHO JUNIOR; BARBOSA, 2017b). Neste trabalho, apresentam-se também os principais padrões encontrados nos *workflows* científicos, dentre os quais alguns puderam ser verificados por esses certificadores.

A segunda família é chamada de C4, que vem do inglês ***Computation Component Certifier Component*** (Componente Certificador de Componente de Computação), e seus componentes têm por finalidade verificar propriedades funcionais e de segurança em componentes da espécie Computação providos por usuários desenvolvedores de componentes (DANTAS; CARVALHO JUNIOR; BARBOSA, 2017a; DANTAS; CARVALHO JUNIOR; BARBOSA; PROENÇA, 2017). Para compor componentes táticos para esses componentes certificadores, estudaram-se as principais técnicas de verificação e ferramentas associadas que permitam a verificação de propriedades formais nas diferentes classes de programas paralelos suportadas pela HPC Shelf. Esta Tese também apresenta os achados nesse sentido.

1.8.1 Motivação

As motivações que levaram à proposição desta Tese se resumem nos seguintes itens:

- Falta de iniciativas que permitam a confecção de *softwares* de CAD mais confiáveis. Os *componentes certificadores* utilizam técnicas de verificação para construir provas formais a respeito de propriedades de interesse sobre os componentes, tornando-os certificados. Componentes certificados aumentarão os níveis de confiança nas aplicações fornecidas pela HPC Shelf. Observe também que, ao se associar um

sistema de certificação paralela como a contraparte de um sistema de computação paralela, atribui-se uma importância semelhante a ambos na nuvem;

- Necessidade de aproximação das técnicas oferecidas pela área de Métodos Formais com a área de CAD. Nesse sentido, existem poucos relatos de trabalhos nesse sentido, os quais incluem trabalhos realizados pelo próprio grupo de pesquisa em que este trabalho se insere (PERVEZ et al., 2010; GOPALAKRISHNAN; KIRBY, 2010; SIEGEL; GOPALAKRISHNAN, 2011; VO et al., 2009; MARCILON; CARVALHO JUNIOR, 2013);
- Escassez de arquiteturas de *software* que permitam a aplicação semi-automática de técnicas de verificação aos componentes de um sistema;
- Carência de serviços disponíveis em nuvens voltados exclusivamente para o auxílio na construção de serviços certificados (Verificação como Serviço);
- Escassez de mecanismos semi-automáticos para se aplicarem diversas infraestruturas de prova, aqui representadas pelos componentes táticos, para provar um determinado objetivo de prova. Aqui, essa aplicação semi-automática se dará por meio de uma orquestração descrita em um código escrito em TCOL. Atualmente, os cientistas envolvidos em prova de propriedades de programas realizam aplicações de provadores manualmente;
- Necessidade de se aumentarem em escala as técnicas de modelagem e verificação formal devido à complexidade inerente das aplicações baseadas em nuvem, dados a heterogeneidade dos recursos e o controle descentralizado e aberto que possuem. De fato, a modularidade da arquitetura proposta neste trabalho, a qual permite a agregação de novos componentes certificadores e táticos quando outros tipos de verificação são necessários, e sua implementação, a qual foi construída através de técnicas de programação de CAD, são um passo nesse sentido;
- Necessidade de tornar a certificação integrada ao sistema, ou seja, que nenhum elemento disruptivo necessite ser incorporado à arquitetura. Nesse sentido, a própria nuvem HPC Shelf é um ecossistema em que componentes que encapsulam motores de verificação podem viver e ser invocados para certificar computações de outros componentes e as maneiras como eles interagem;
- Contribuir com o projeto maior, a HPC Shelf, no que diz respeito à escrita de componentes de computação certificados por parte de *desenvolvedores de componentes* e *workflows* certificados por parte de *provedores de aplicações*.

1.8.2 Objetivos e Contribuições

O objetivo principal da Tese proposta é:

> Desenvolver e validar um arcabouço de verificação como serviço para certificação de componentes paralelos em uma plataforma de nuvens computacionais para serviços de computação de alto desempenho.

Os objetivos específicos da Tese proposta são:

- Adaptar a HPC Shelf para acomodar as duas novas espécies de componentes: os certificadores e os táticos;
- Elaborar uma infraestrutura de comunicação entre componentes certificadores e componentes táticos;
- Criar uma linguagem que permita a criação de *workflows* de verificação (TCOL), os quais orquestram um conjunto de componentes táticos de acordo com as necessidades do processo de certificação comandado por um componente certificador;
- Desenvolver um arcabouço de certificação de componentes paralelos, o qual possui como pano de fundo um sistema de certificação paralela e possui como elementos de construção os componentes certificadores;
- Estudar os principais padrões encontrados em *workflows* científicos e estabelecer quais deles são verificáveis;
- Estudar as principais técnicas de verificação de programas e ferramentas relacionadas para estabelecer quais se adequam à verificação dos programas paralelos da HPC Shelf;
- Fornecer duas implementações particulares de componentes certificadores, SWC2 e C4, através dos quais componentes de interesse de aplicações com requisitos de computação de alto desempenho serão certificados;
- Avaliar a arquitetura proposta, por meio da criação de estudos de caso com cenários arquiteturais de certificação para aplicações com requisitos de computação de alto desempenho, as quais terão alguns de seus componentes certificados por componentes SWC2 e C4.

Visa-se contribuir com este trabalho para o estado da arte por meio dos seguintes pontos:

- A primeira arquitetura VaaS que explora técnicas de CAD para esse fim;
- A linguagem TCOL, para orquestração de infraestruturas de prova (componentes táticos) a fim potencializar o uso de provadores nas provas das propriedades;
- O projeto arquitetural dos componentes SWC2 e C4, que devem ser associados a componentes de alto desempenho para que estes possam fazer uso das funcionalidades de certificação;
- Uma análise sobre quais das principais técnicas e ferramentas de verificação formal de *software* disponíveis melhor se adequam a verificar propriedades de aplicações de CAD, através da pesquisa em artigos, livros e pela experimentação;
- Um conjunto de aplicações iniciais certificadas que servirão como base e auxílio para uma gama maior de novas aplicações de Computação de Alto Desempenho certificadas.

1.8.3 Resultados Esperados

Além da versão inicial da arquitetura, que engloba os componentes certificadores, almeja-se alcançar os seguintes resultados:

- Um estudo sobre as técnicas em Verificação Formal de *Software* existentes e a escolha de quais podem ser aplicadas a ambientes distribuídos heterogêneos, como a HPC Shelf;
- Um estudo sobre os principais padrões verificáveis encontrados nos *workflows* científicos e a estratégia de verificação para cada um;
- Um protótipo do arcabouço de certificação e da linguagem de orquestração de componentes táticos;
- Um protótipo de uma implementação específica de um componente SWC2 e um protótipo de uma implementação específica de um componente C4, bem como protótipos das implementações dos componentes táticos associados;
- Um conjunto de aplicações certificadas, as quais terão seus processos de certificação demonstrados. Espera-se que tanto a arquitetura descrita aqui quanto as aplicações certificadas sejam utilizadas em outras iniciativas de construção de aplicações dentro do grupo de CAD do MDCC;
- Trabalhos futuros relativos à construção de aplicações certificadas em nível de mestrado e doutorado no grupo de CAD do MDCC.

A partir desses elementos, espera-se que tanto a arquitetura descrita aqui quanto as aplicações certificadas sejam utilizadas em outras iniciativas de construção de aplicações dentro do grupo de CAD do MDCC.

1.9 Trabalhos Relacionados

Os trabalhos relacionados às contribuições desta Tese podem ser classificados em três vertentes: arquiteturas de verificação como serviço, verificação de *workflows* científicos e estudos sobre técnicas e ferramentas para verificação de programas paralelos.

A verificação como serviço está ainda nos primeiros passos e apenas algumas iniciativas puderam ser encontradas. (MANCINI et al., 2015) propõem um serviço de verificação para mostrar a corretude do sistema com relação a eventos incontroláveis através de uma simulação exaustiva de *hardware*, considerando todos os cenários relevantes. Por sua vez, (BELLETTINI; CAMILLI; CAPRA; MONGA, 2015) propõem um algoritmo MapReduce para verificação de fórmulas CTL em um sistema em nuvem.

O trabalho mais próximo da principal contribuição desta Tese é o arcabouço proposto por (HU; LEI; TSAI, 2016). Eles propõem um arcabouço robusto de verificação como serviço, focando principalmente no dualismo com as principais preocupações de SaaS (Software como Serviço), como o armazenamento de ferramentas e resultados de verificação, problemas de escalabilidade durante a verificação e tolerância a falhas durante

a verificação. Sua abordagem, no entanto, permite a disponibilidade de ferramentas de verificação na nuvem de uma maneira crua, exigindo assim um desenvolvedor de aplicações com habilidades no tipo de verificação suportada pela ferramenta, o que muitas vezes não é verdade. Além disso, sua abordagem não torna possível fazer suposições sobre as plataformas de computação nas quais as verificações serão realizadas. Na verdade, o uso de plataformas de computação paralela não é uma preocupação.

Como dito anteriormente, esta Tese apresenta o primeiro arcabouço VaaS voltado para os requisitos de CAD. Além disso, ela introduz características inovadoras em comparação com o arcabouço proposto por (HU; LEI; TSAI, 2016). Por exemplo, os desenvolvedores de aplicações apenas selecionam no catálogo componentes certificadores para certificar os componentes que desejam e todo o processo de certificação é executado semi-automaticamente, através da orquestração de componentes táticos em um *workflow* predefinido. Além disso, os componentes certificadores e táticos fazem uso de técnicas de CAD, podendo explorar diferentes níveis de paralelismo suportados por plataformas de CAD (memória distribuída, memória compartilhada, *multi/many-core*, aceleradores computacionais, etc.). De fato, acelerar a certificação de componentes certificáveis durante a orquestração da aplicação é uma preocupação relevante, pois é um pré-requisito para a execução desses componentes na aplicação.

Vários estudos relacionados à formalização e verificação de *workflows* de negócio foram propostos, incluindo: regras de Evento-Condição-Ação (gatilhos) (DAYAL; HSU; LADIN, 1990; FU; BULTAN; HULL; SU, 2001; HULL et al., 1999); métodos baseados em lógicas (ATTIE; SINGH; SHETH; RUSINKIEWICZ, 1993; DAVULCU; KIFER; RAMAKRISHNAN; RAMAKRISHNAN, 1998; SENKUL; KIFER; TOROSLU, 2002); Redes de Petri (ADAM; ATLURI; HUANG, 1998; AALST, 1997; AALST, 1998); e Diagramas de Estados (WODTKE; WEIKUM, 1997). No entanto, não foram encontradas iniciativas focadas na formalização e verificação de padrões de *workflows* científicos.

Apesar de outras classes de paralelismo e ferramentas associadas frequentemente serem empregadas na confecção de aplicações com requisitos de Computação Alto Desempenho, o paralelismo com memória distribuída empregado pela biblioteca MPI ainda é um padrão *de facto* nessas aplicações. Dessa forma, tanto por esse fato quanto pelo fato de não existirem muitas ferramentas para verificação de programas MPI, o esforço dispensado neste trabalho, apesar de contemplar todas as classes encontradas, focou principalmente na verificação desses programas. O único trabalho que relatou todas as principais ferramentas de verificação de programas MPI disponíveis na época foi o trabalho de Gopalakrishnan *et al* (GOPALAKRISHNAN et al., 2011). Contudo, todas as ferramentas utilizavam sempre alguma espécie de *model checking*, em geral combinado com outra técnica, como por exemplo a execução simbólica. Como se sabe, um dos principais problemas dos *model checkers* é a explosão espacial, fato que fez com que essas ferramentas fôssem preteridas em uma primeira análise em nossos estudo, priorizando a investigação de ferra-

mentas de verificação dedutiva, que não sofrem desse problema. Nesse sentido, o trabalho desta Tese apresenta inovações quando comparado aos dos referidos autores.

1.10 Estrutura do Documento

Este documento está dividido da seguinte maneira:

No Capítulo 2, apresenta-se o mecanismo de certificação proposto, usando a HPC Shelf como substrato. Isso é feito primeiro mostrando a HPC Shelf como é hoje e depois mostrando as extensões e conceitos do arcabouço de certificação. O Capítulo 3 descreve as técnicas e ferramentas de Verificação Formal de *Software* que foram utilizadas para compor componentes táticos para os componentes certificadores SWC2 (Componente Certificador de *Workflow* Científico) e C4 (Componente Certificador de Componente de Computação). Já o Capítulo 4 descreve as particularidades arquiteturais dos componentes SWC2. O Capítulo 5, por sua vez, apresenta as singularidades da arquitetura dos componentes C4. No Capítulo 6, são mostrados dois estudos de caso para este trabalho, nos quais aplicações reais com requisitos de computação de alto desempenho tiveram seus componentes certificados. Por fim, no Capítulo 7, além das conclusões do estudo, também é incluída uma discussão sobre limitações dos resultados alcançados por esse trabalho, bem como sugestões de trabalhos futuros relacionados aos componentes certificadores.

2 A Plataforma HPC Shelf

Este capítulo apresenta a plataforma HPC Shelf, uma proposta de nuvem computacional para serviços de Computação de Alto Desempenho que visa auxiliar usuários especialistas na construção de aplicações que resolvam problemas que demandem por computação intensiva, possivelmente envolvendo múltiplos *clusters*, constituindo plataformas de processamento paralelo heterogêneas e de larga escala.

Esta Tese de Doutorado insere-se no projeto HPC Shelf propondo um arcabouço para certificação de componentes de diferentes espécies suportadas por essa plataforma. Neste capítulo, inicialmente será apresentado o *status* atual da HPC Shelf. Então, na Seção 2.6, será introduzido o arcabouço de certificação, como uma extensão sobre a arquitetura original da plataforma. Isso servirá de base para a introdução de componentes certificadores para propósitos específicos, respectivamente nos capítulos 4 e 5.

A HPC Shelf visa oferecer um ambiente que permita a construção e execução de aplicações com requisitos de Computação de Alto Desempenho de forma intuitiva, abstraindo dos usuários chamados de *especialistas* preocupações relativas a técnicas e tecnologias de processamento paralelo. Para tanto, a descrição dos problemas a serem resolvidos pelas aplicações deve ser feita utilizando abstrações do domínio de atuação do especialista, permitindo esconder, portanto, os detalhes da implementação paralela e da arquitetura que executará o *software*. Para isso, os seguintes objetivos específicos circundam o seu projeto:

- Fornecer uma plataforma que permita a integração de usuários especialistas de domínio, como engenheiros e cientistas. Esses usuários podem utilizar a nuvem para fornecer soluções a problemas computacionalmente intensivos uns aos outros;
- Propiciar um arcabouço para a montagem de aplicações compostas por componentes de sistema capazes de explorar o processamento paralelo para resolução de problemas descritos pelos usuários especialistas;
- Permitir que aplicações monitorem o progresso de computações de longa duração, visualizando suas saídas e armazenando dados para fins de verificação da qualidade do *software* proposto;
- Prover um mecanismo de seleção dinâmica de componentes, cujas implementações melhor se adequem ao contexto determinado pela aplicação, ou seja, cujo código fonte possa extrair o maior poder de processamento da plataforma de computação paralela sobre a qual o componente irá executar;
- Suportar a execução de código legado em plataformas de computação paralela modernas;
- Viabilizar uma linguagem de descrição arquitetural de *workflows* para orquestrar componentes paralelos visando a construção de sistemas de computação paralela que expressam soluções computacionais de alto desempenho para problemas de interesse

da aplicação.

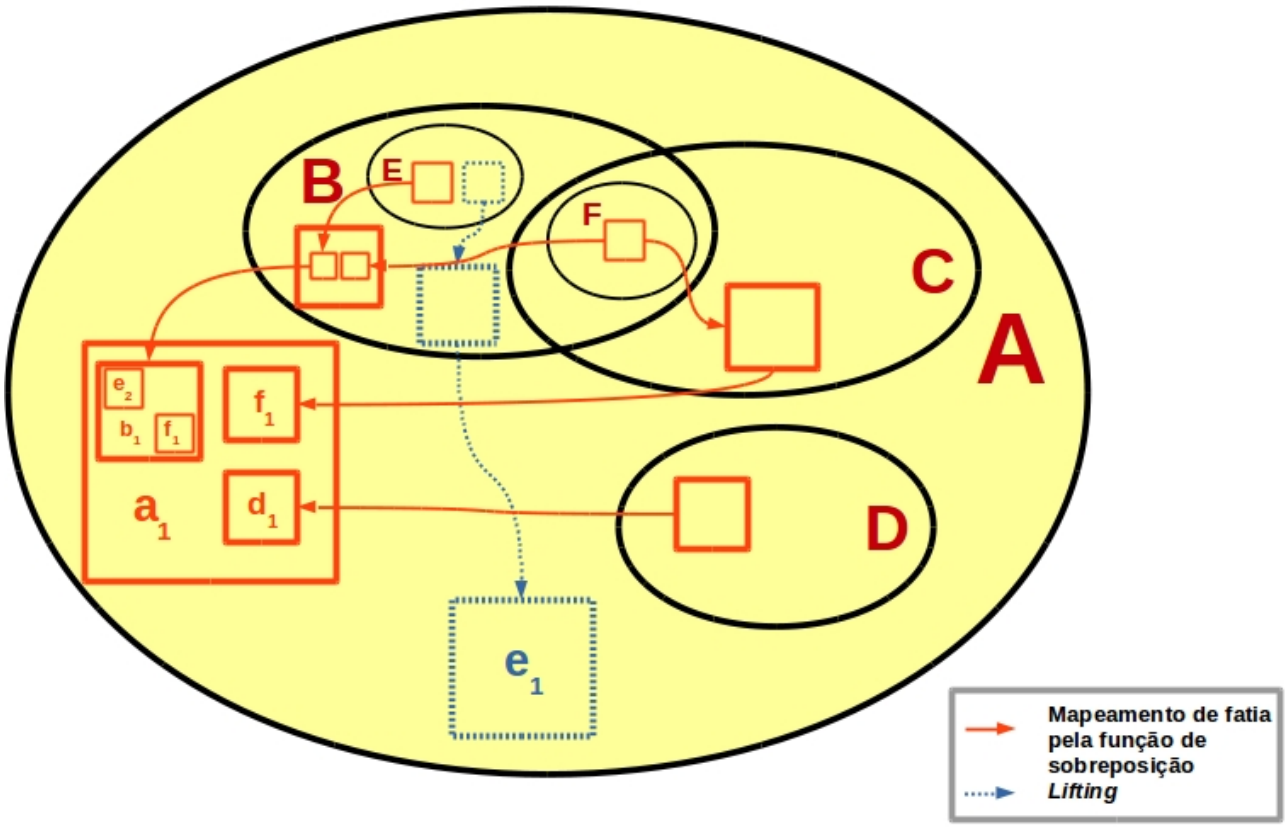
Um *sistema de computação paralela* é uma composição de elementos de *software* com requisitos de Computação de Alto Desempenho com as plataformas de computação paralela (elementos de *hardware*) sobre as quais o *software* deverá executar. Essa visão integrada de *hardware* e *software* é importante dentro do contexto da HPC Shelf, uma vez que é comum que o *software* paralelo seja voltado a explorar o máximo do potencial da plataforma de computação paralela, utilizando algoritmos que fazem suposições sobre características de sua arquitetura, incluindo hierarquias de memória e paralelismo, bem como a presença de aceleradores computacionais (GPUs, MICs, FPGAs, etc.). No caso da HPC Shelf, tanto os elementos de *software* quanto os elementos de *hardware* consistem de um conjunto de componentes paralelos, e é usual utilizar-se o termo *componente de sistema* para se referir à composição de um componente de *software* com o componente que representa a plataforma de computação paralela onde irá executar.

A HPC Shelf faz uso do modelo de componentes paralelos Hash e de uma extensão do seu sistema de tipos HTS (*Hash Type System*) (CARVALHO JUNIOR; REZENDE; SILVA; ALAM, 2013) chamada de *sistema de contratos contextuais*, o qual permite a separação das *interfaces* dos componentes de suas implementações. Com essa separação, pode-se ter, para um mesmo componente, dito *componente abstrato*, várias implementações possíveis cadastradas em uma biblioteca de componentes e acessíveis por meio de um serviço de catálogo, ditas *componentes concretos*, as quais podem ser utilizadas alternativamente de acordo com sua capacidade de melhor atender os requisitos da aplicação e utilizar os recursos oferecidos pela plataforma de computação paralela hospedeira. Detalhes sobre o sistema de contratos contextuais da HPC Shelf serão apresentados na Seção 2.5.

2.1 O Modelo Hash de Componentes

O modelo Hash de componentes (CARVALHO JUNIOR; LINS; CORREA; ARAÚJO, 2007) tem como finalidade prover um formalismo para a criação de componentes inerentemente paralelos, chamados de *componentes-#*, os quais devem executar sobre plataformas de computação paralela de memória distribuída. Cada componente-# é composto por um conjunto de *unidades* (paralelas), em que cada uma representa um agente de processamento (processo) que executa em um nó de processamento distinto, comunicando-se com as demais através de trocas de mensagens. Componentes-# podem ser criados de maneira hierárquica, podendo conter outros componentes *aninhados* a eles. A maneira em que essa composição se dá é através de um mecanismo chamado de *composição por sobreposição*, o qual se vale de uma *função de sobreposição*, a qual mapeia cada unidade de um componente dito *componente aninhado*, que faz parte da sua composição, a uma unidade do componente hospedeiro. Chama-se de *fatia* da unidade do componente hospedeiro, a

Figura 1: Um exemplo de composição por sobreposição



Fonte: Elaborado pelo autor.

unidade do componente aninhado a ela associada através da função de sobreposição.

Considere a ilustração da Figura 1, a qual apresenta um cenário de composição por sobreposição para um componente **A**, o qual possui as unidades a_1 e e_1 e os componentes aninhados **B, C, D, E** e **F**. Para a unidade a_1, são mapeadas pela função de sobreposição as unidades b_1, de **B**, d_1, de **D**, e f_1, de **F**. Mais ainda, cada mapeamento configura uma fatia de a_1. A unidade e_1 de **A** vem diretamente do componente **E** por uma operação chamada de *lifting* (seta tracejada). Observe ainda que **F** é um componente compartilhado por **B** e **C**. Ele pode representar, por exemplo, uma instância de uma estrutura de dados compartilhada entre os dois componentes.

O modelo Hash prevê ainda que plataformas que a ele aderem possuam um conjunto de *espécies de componentes*, as quais agrupam componentes com natureza semelhante. Mais ainda, componentes de uma mesma espécie devem apresentar os mesmos modelos de implantação e interação com o resto do sistema. Assim, qualquer plataforma que implemente o modelo Hash deve fornecer um conjunto de espécies de componentes para usufruto por parte das aplicações.

O HPE *(Hash Programming Environment)* foi a primeira plataforma compa-

tível com o modelo Hash (CARVALHO JUNIOR; REZENDE, 2013), sendo atualmente considerada sua plataforma de referência. O HPE visa a construção de aplicações de alto desempenho utilizando a composição de componentes do modelo Hash, para execução sobre *clusters* computacionais. Com os avanços recentes nessa plataforma, destaca-se sua compatibilização com o modelo de componentes CCA (CARVALHO JUNIOR; CORREA, 2010), fornecendo um conceito geral de componente paralelo que poderia ser adotado por outros *frameworks* computacionais compatíveis com o CCA.

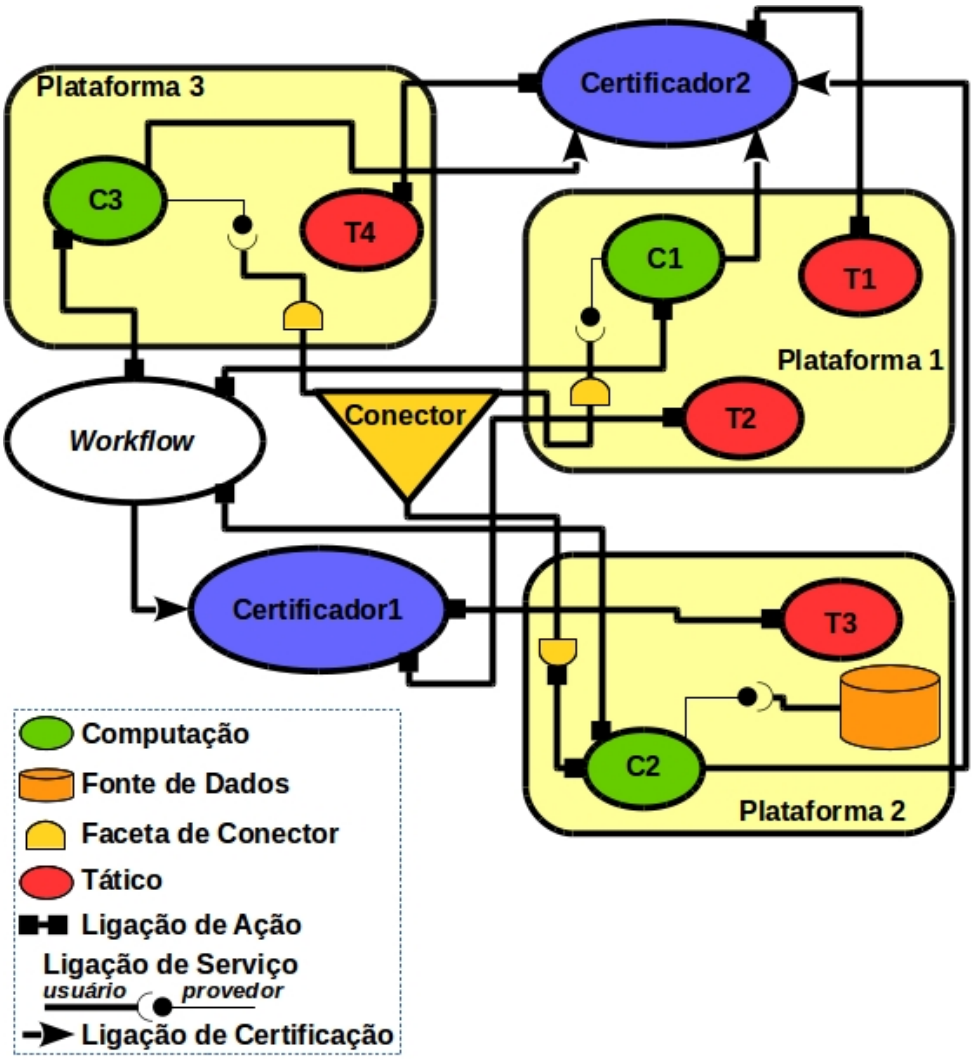
2.2 Espécies de Componentes

As espécies de componentes suportadas pela plataforma HPC Shelf são:

- *Plataformas*: representam plataformas de computação paralela de memória distribuída, compostas por um subconjunto dos recursos de uma infraestrutura para computação paralela em larga escala. São também chamadas, dentro do contexto da HPC Shelf, de *plataformas virtuais*;
- *Computações*: representam implementações de algoritmos paralelos otimizados para explorar características arquiteturais de uma classe de plataformas virtuais;
- *Fontes de Dados*: representam repositórios de dados de interesse de componentes de computação, de acordo com as aplicações;
- *Qualificadores*: são utilizados em *contratos contextuais* para representar interesses não-funcionais de componentes, aqueles que não correspondem a elementos concretos de *software*, tais como, por exemplo, características da plataforma virtual sobre o qual um componente de computação vai executar ou os requisitos/restrições de uma aplicação em relação ao componente que deseja utilizar;
- *Conectores*: acoplam componentes de computação e fontes de dados que se encontram executando em plataformas virtuais distintas, a fim de tornar possível sua orquestração ou servir de meio para a sua coreografia;
- *Bindings*: ligam componentes entre si, a fim de oferecerem, uns para os outros, serviços não-funcionais (*bindings de serviços*) ou funcionais (*bindings de ações*).

A comunicação entre os componentes se dá através de portas, ligadas umas às outras através de *bindings*, de serviços ou de ações. Portas entre *bindings* de serviços podem ser provedoras ou usuárias. Uma *porta provedora* deve se ligar a uma *porta usuária*, fazendo com que os serviços disponibilizados por um componente através da porta provedora possam ser consumidos pelo outro componente através da porta usuária, sendo o *binding* de serviços responsável pela intermediação da comunicação entre elas. Por sua vez, *portas de ações* podem ser conectadas caso exportem o mesmo conjunto de *ações*, identificadas por *nomes*. Dessa maneira, ao invocar uma ação em uma porta, a orquestração é bloqueada até que haja uma invocação da ação de mesmo nome em todas as portas parceiras, ou progride caso já haja invocações pendentes em todas elas. Portas

Figura 2: Um exemplo de componentes e suas ligações



Fonte: Elaborado pelo autor.

de ações são úteis para fins de orquestração. A Figura 2 mostra um cenário que ilustra componentes e suas ligações.

2.3 Atores

Os atores da nuvem HPC Shelf representam as partes interessadas no consumo e provimento de serviços na nuvem. Podem ser agrupados nas seguintes classes, as quais caracterizam interesses e responsabilidades comuns: *usuário especialista, provedor de aplicações, desenvolvedor de componentes* e *mantenedor de plataforma.*

2.3.1 Usuário Especialista

O principal objetivo da HPC Shelf é a disponibilização de serviços para este usuário, sendo considerado o usuário final da nuvem. Em geral, não são especialistas em técnicas

de computação e visam apenas expressar e resolver problemas de seu interesse utilizando uma linguagem de alto nível de abstração, geralmente de propósito especial. Esses usuários possuem as seguintes características:

- Pertencem a um domínio específico das ciências exatas (física, química, biologia, economia, etc.) ou engenharias;
- Além de não precisarem conhecer técnicas de programação paralela, não necessitam ter conhecimento sobre a arquitetura de plataformas de computação paralela. Necessitam apenas de uma *interface* adequada para a especificação dos problemas de seu interesse;
- As aplicações que os servirão na resolução de problemas de seu interesse devem fornecer abstrações voltadas ao campo de atuação do especialista, com uma linguagem intuitiva de propósito especial (DSL) para a descrição de problemas;
- Podem interagir com a aplicação para acompanhar o andamento da solução, podendo, ao final do processo, utilizar os resultados brutos ou realizar algum pós-processamento para a sintetização de dados através de ferramentas disponíveis pela própria aplicação;
- A visão do especialista é de uma nuvem SaaS, ou seja, não tem conhecimento sobre a infraestrutura e as características de implementação do *software* que utiliza. De fato, o usuário especialista não precisa ter consciência de estar utilizando a HPC Shelf, ou mesmo que as soluções usadas pela aplicação são baseadas em componentes. No entanto, caso seja interessante para ele, pode interagir com outros atores da nuvem a fim de ter poder de decisão em nível de estrutura ou na composição dos componentes da aplicação.

O especialista enxerga a aplicação como uma linguagem específica de domínio, que pode ser textual, gráfica ou simplesmente uma entrada de dados em um navegador *web* para realizar computações pré-estabelecidas pela aplicação. Essa linguagem pode ser universalmente expressiva, como uma linguagem de programação completa capaz de descrever computações, ou simplesmente uma entrada de valores a parâmetros que fará com que a aplicação execute uma certa solução computacional para resolver um problema específico. Essa linguagem diz respeito somente a propriedades da aplicação e não tem relação com o *workflow* que orquestrará os componentes para resolver o problema.

O usuário especialista pode ainda monitorar a execução da aplicação, aplicando transformações nos dados de saída, caso a aplicação forneça esse serviço.

2.3.2 Provedor de Aplicações

É um especialista em métodos computacionais para solução de problemas em algum domínio específico, com conhecimentos sobre arquitetura e engenharia de *software*. Está interessado em construir aplicações para usuários especialistas através da composição de componentes. Também não necessitam ter conhecimento sobre técnicas de programação

paralela, mas deve ter a capacidade de configurar componentes pré-instalados na montagem de aplicações. Esses atores possuem as seguintes particularidades:

- São responsáveis por definir a representação dos problemas propostos pelos especialistas através de uma orquestração de componentes descrita em uma linguagem de *workflows* suportada pela HPC Shelf;
- Possuem uma visão PaaS da nuvem, pois ela os permite a construção e a implantação da infraestrutura de *software* das aplicações;
- Definem todas as funcionalidades e controles que serão disponibilizados aos especialistas na aplicação;
- Traduzem as restrições de desempenho e custo energético impostas pelos especialistas sobre as plataformas de computação paralela que atenderão a aplicação em estratégias adequadas que guiarão a HPC Shelf na escolha das plataformas apropriadas no momento da execução da aplicação.

Através de uma IDE (*Integrated Development Environment*) de programação Java ou C# qualquer, o provedor de aplicações pode utilizar classes e padrões de projeto do arcabouço SAFe para construir *interfaces* intuitivas, que podem ser gráficas ou não, utilizando abstrações comuns ao linguajar dos usuários especialistas para os quais as aplicações serão construídas. Nesse mesmo ambiente, ele pode criar também os sistemas de computação paralela e *workflows* associados que resolverão os problemas que as aplicações se propõem a resolver. Cada *workflow* desenvolvido é na realidade um mapeamento do problema submetido pelo usuário especialista em uma estratégia de ativação (orquestração) dos componentes do sistema de computação paralela por ele criado que visa resolver o problema, a qual é escrita na linguagem de *workflows* da HPC Shelf, chamada SAFeSWL (*SAFe Scientific Workflow Language*).

No momento da execução da aplicação, os componentes de solução que compõem o sistema de computação paralela orquestrado pelo *workflow* que resolverá o problema são submetidos pelo SAFe ao Core para que seja feita a escolha das implementações desses componentes. Para cada componente (incluindo plataformas) é retornada à aplicação uma lista de implementações candidatas de acordo com o algoritmo de resolução (ver Seção 2.5). Caso seja interesse do provedor de aplicações, ele pode programar a aplicação para escolher uma implementação candidata específica, diferente daquela que figura em primeiro na lista. Caso não seja interesse, ele deixa que o Core escolha automaticamente a implementação que está em primeiro na lista.

O provedor de aplicações pode ainda acessar a *interface* gráfica do SAFe para monitorar a escolha feita pelo Core para os componentes dos *workflows* submetidos e para monitorar também a escolha final dos componentes feita pela aplicação, caso ele a tenha programado para atuar nessa escolha.

2.3.3 Desenvolvedor de Componentes

É um especialista em computação paralela, incluindo técnicas de programação paralela e detalhes arquiteturais de plataformas de computação paralela. São capazes de criar componentes, definindo suas regras e funcionalidades. Os componentes podem então ser cadastrados na biblioteca de componentes da HPC Shelf a fim de serem utilizados por outros desenvolvedores de componentes e provedores de aplicações. Podem-se citar suas seguintes características:

- Possuem, em geral, formação em Ciência da Computação ou afins e têm especialização em técnicas de computação paralela;
- Podem empregar técnicas e ferramentas de alto nível de especificidade em computação paralela, dependendo da arquitetura em questão. Por exemplo, podem utilizar OpenMP e PThreads em um ambiente de memória compartilhada, MPI e técnicas de RPC em um ambiente de memória distribuída e CUDA para a programação em aceleradores computacionais da classe GPU;
- Da mesma forma que os provedores de aplicações, possuem uma visão PaaS da nuvem;
- Utilizando-se de suas habilidades em técnicas de computação e algoritmos paralelos, podem criar códigos de componentes otimizados para melhor explorar as características arquiteturais de plataformas de computação paralela;
- Para um mesmo componente, podem registrar na HPC Shelf várias implementações possíveis, em que cada uma será propícia a extrair um melhor desempenho quando for utilizada na plataforma de computação paralela para a qual foi construída;
- Têm habilidade na construção de *software* em larga escala, utilizando a composição de componentes a partir de outros existentes.

Esse ator utiliza uma IDE que permite a criação de componentes abstratos e suas assinaturas contextuais e a programação de componentes concretos e seus contratos contextuais. Essa IDE tem comunicação com o Core para os catalogar e deixar os componentes disponíveis para usufruto por parte das aplicações e outros componentes. Essa IDE também deve oferecer linguagens de programação e bibliotecas voltadas à programação paralela, permitindo, assim, que o desenvolvedor possa criar componentes otimizados segundo as características arquiteturais de uma classe de plataformas de computação paralela. Atualmente, essa IDE é uma adaptação do Front-End do HPE, a plataforma de referência do modelo Hash mencionada na Seção 2.1.

2.3.4 Mantenedor de Plataformas

São responsáveis por implantar, configurar, disponibilizar e monitorar as plataformas virtuais da HPC Shelf, as quais são representadas na nuvem como componentes da espécie *plataforma*. Componentes plataforma armazenam todas as informações necessárias para

instanciar plataformas virtuais, as quais devem apresentar as mesmas características de máquinas de computação paralelas reais. Esses usuários apresentam as seguintes características:

- Devem garantir a segurança das aplicações que executam sobre sua infraestrutura, segundo as políticas gerais de segurança da HPC Shelf;
- São responsáveis por instalar e manter todas as bibliotecas necessárias à execução das aplicações;
- Podem possuir infraestruturas de computação de tecnologias diversas, tais como *clusters*, Máquinas Massivamente Paralelas (MPPs), GPUs, dentre outras, e oferecer diversas combinações dessas tecnologias sob a abstração de plataformas virtuais, cada uma registrada no catálogo de componentes da HPC Shelf como um componente da espécie *plataforma*;
- Possuem uma visão IaaS da nuvem, uma vez que ela oferece serviços de infraestrutura de computação paralela para as aplicações.

Esse ator utiliza uma IDE cujo papel é semelhante ao da IDE do Desenvolvedor de Componentes (atualmente, uma adaptação do Front-End do HPE), no entanto tratando sobre especificidades de componentes da espécie *plataforma*. Nela, os mantenedores configuram a infraestrutura da nuvem, representada por plataformas virtuais, nas quais os componentes desenvolvidos serão implantados, instanciados e executados.

Para cada plataforma virtual a ser cadastrada, deve-se especificar um contrato contextual (referido nesse contexto como *perfil de plataforma virtual*), a fim de guiar o Core no cálculo de qual plataforma virtual melhor servirá o componente a ser resolvido. Dada uma plataforma virtual escolhida, ela é instanciada sobre a infraestrutura de computação paralela do mantenedor. O componente é, então, implantado na plataforma virtual e pode executar utilizando somente os recursos que correspondem à definição da plataforma virtual.

2.4 Visão Arquitetural da HPC Shelf

A plataforma HPC Shelf é composta pelos elementos arquiteturais Front-End, Core e Back-End, os quais serão detalhados nas seções seguintes.

2.4.1 Front-End: O Arcabouço SAFe

Esse elemento representa a *interface* através da qual a aplicação acessa os serviços da plataforma. Atualmente, o Front-End é representado pelo arcabouço SAFe.

O SAFe (*Shelf Application Framework*) constitui o arcabouço para a criação de aplicações voltadas ao usuário especialista na HPC Shelf por parte dos provedores de aplicações (SILVA; CARVALHO JUNIOR, 2016).

O SAFe não impõe a forma com que as aplicações devem ser apresentadas aos

usuários, a qual pode ser na forma de aplicações *stand-alone*, *plugins* IDE, *interfaces web*, linha de comando, dentre outras. De fato, a aplicação é meramente uma abstração para esconder os detalhes da arquitetura de componentes paralelos e plataformas virtuais da HPC Shelf. Dessa forma, o trabalho dos provedores de aplicações consiste em mapear os problemas de interesse dos usuários especialistas nos sistemas de computação paralela que irão os resolver.

O SAFe fornece todos os serviços relativos a criação, instanciação e execução de sistemas de computação paralela baseados em componentes e orquestrados por *workflows* científicos para resolver problemas de computação intensiva especificados pelos especialistas através da *interface* da aplicação.

O SAFe se comunica com os módulos Core e Back-End, e essa comunicação se dá por meio serviços *web*. Pode se comunicar com o Core em dois momentos distintos.

O primeiro deles ocorre durante a criação da aplicação, em que o provedor de aplicações acessa a biblioteca de componentes para buscar componentes abstratos que o interessem para compor o sistema de computação paralela da aplicação e o *workflow* associado.

No segundo momento, durante a execução da aplicação, o SAFe dispara a execução da lógica do *workflow*, o qual, por sua vez, acessa o Core na intenção de ativar ações do ciclo de vida dos componentes e ativar ações computacionais nos componentes. As ações do *ciclo de vida dos componentes* são as seguintes:

- **resolve**: recebe um contrato contextual e requer ao Core que o resolva, ou seja, encontre no catálogo componentes concretos compatíveis com o contrato contextual, que melhor representem o contexto expresso pelo contrato. Portanto, o Core retorna uma lista de componentes que satisfazem o contrato fornecido;
- **deploy**: para um componente da espécie *plataforma* escolhido, cria a plataforma virtual com as características especificadas no seu contrato sobre a infraestrutura do mantenedor que registrou o componente. Por sua vez, para componentes de outras espécies, solicita ao Core que o instale na plataforma apropriada, informando a localização do componente para o *workflow*, para que, a partir de agora, ele possa se comunicar diretamente com ela;
- **instantiate**: diz ao Core para solicitar a instanciação do componente junto ao Back-End, ou seja, o tornar pronto para execução, podendo agora ter uma ação computacional invocada;
- **release**: caso o componente não seja mais útil à aplicação, ela pode solicitar ao Core que o libere da plataforma em que ele está instalado ou, no caso de um componente plataforma, que desaloque os recursos de máquina virtual.

O SAFe se comunica com o Back-End diretamente em um só momento. Isso ocorre quando o componente é instanciado e o Core repassa à aplicação o endereço do componente concreto na plataforma gerenciada pelo Back-End. O componente concreto

se comunica com o SAFe (via serviços *web*), o qual fornece à aplicação acesso às portas do componente.

Os provedores de aplicações podem compor aplicações utilizando a biblioteca de classes (API) do SAFe, que possui tanto uma versão para a linguagem Java quanto uma versão para a linguagem C#, e é composta por dois grandes pacotes chamados, respectivamente, de safe-framework e safe-language.

O safe-framework possui todas as classes e *interfaces* necessárias para a criação de aplicações utilizando componentes. A troca de informações em um *workflow* então se dá pela composição de componentes através de objetos do tipo *service*. A utilização desse objeto para a troca de informações entre componentes é inspirada do modelo CCA, introduzido no Capítulo 1 [18].

Cada componente possui um objeto *service*, o qual armazena referências para portas de *serviços* e de *ações*. Cada componente também possui uma porta de ações predefinida para controlar seu ciclo de vida, chamada *LifeCycle*, que exporta as ações cujos nomes são **resolve**, **deploy**, **instantiate** e **release**, cujo significado foi descrito anteriormente.

O pacote safe-language define um conjunto de classes responsáveis por gerenciar a linguagem de orquestração do SAFe, chamada de SAFeSWL (*Scientific Workflow Language*). Essa linguagem é subdividida em dois subconjuntos distintos, uma responsável pelas descrições arquiteturais do *workflow* e a outra responsável pela descrição do fluxo de execução do *workflow*. Ambas as gramáticas são representadas utilizando o formato XSD (*XML Schema Definition*).

2.4.1.1 A Linguagem SAFeSWL

A linguagem SAFeSWL é composta por dois subconjuntos. O subconjunto arquitetural permite a definição de quais componentes farão parte da arquitetura de sistemas de computação paralela, bem como a forma como eles estão relacionados. O subconjunto de orquestração, por sua vez, fornece construtores que permitem a especificação de um *workflow*, ou seja, de um código de orquestração que dita o fluxo de execução das ações de componentes de solução de sistemas de computação paralela. Ambos, o código arquitetural e o código de orquestração, são definidos em arquivos XML, os quais devem ser fornecidos pelo provedor ao SAFe.

A apresentação da gramática do subconjunto arquitetural é dispensável para os propósitos desta Tese, uma vez que se está interessado somente em verificar propriedades formais sobre as orquestrações impostas pelos *workflows*, as quais são definidas utilizando os elementos da gramática do subconjunto de orquestração.

[18] Apesar da utilização de objetos *service*, a HPC Shelf não busca ser um *framework* compatível com o CCA.

Figura 3: Sintaxe abstrata da linguagem de orquestração de SAFeSWL

TASK ::= **skip** | **invoke** *action* | **start** *handle action* | **wait** *handle* | **cancel** *handle* |
sequence TASK+ | **parallel** TASK+ | **repeat** TASK | **break** | **continue** |
select {*action*: TASK}+

Fonte: Elaborado pelo autor.

2.4.1.2 O Subconjunto de Orquestração

Em um sistema de computação paralela, o componente workflow atua como um conector ligado aos componentes que farão parte da orquestração, os quais podem ser componentes de computação ou conectores. Essa ligação se dá através de portas de ações, as quais devem estar registradas nas especificações arquiteturais dos componentes. Cada porta exporta um conjunto de ações e para que uma ação seja ativada ela deve ser invocada por ambos os componentes ligados pela porta. A orquestração executada pelo *workflow* especifica uma lógica de ativação de ações, utilizando diversos combinadores, os quais permitem especificar sequenciamento, alternativas, iterações, concorrência e chamadas assíncronas de ações. A orquestração é definida pela gramática presente na Figura 3.

Uma ação, representada na gramática pela variável *action*, pode ser uma das ações da porta predefinida *LifeCycle* (**resolve**, **deploy**, **instantiate** ou **release**). Pode ser também uma ação computacional declarada em uma porta descrita na descrição arquitetural do *workflow*.

Uma tarefa, representada pela variável TASK, consiste em uma orquestração de um conjunto de ações. O *workflow* é, então, definido por uma tarefa de mais alto nível, que orquestra as tarefas internas. Existem sete tipos primitivos de tarefas e quatro combinadores de tarefas.

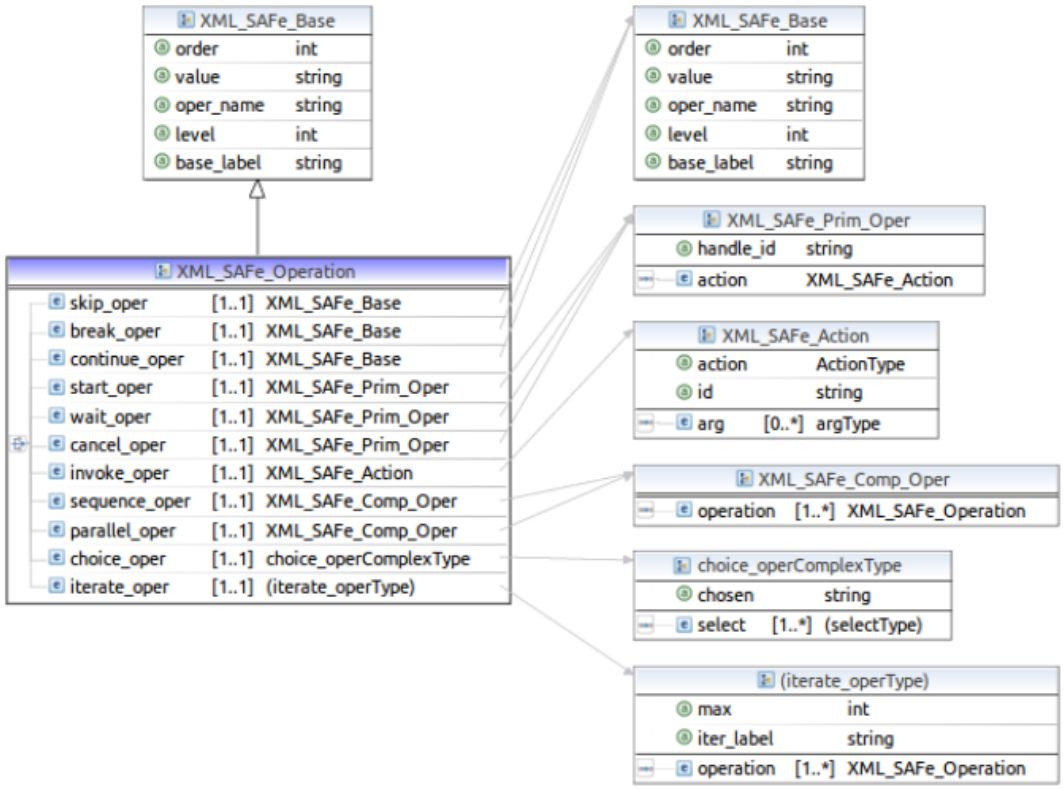
As tarefas primitivas são:

- **skip** denota uma ação vazia;
- **break** termina a iteração na qual se encontra aninhada;
- **continue** volta ao início da iteração;
- **start** denota a ativação assíncrona de uma ação, opcionalmente tratada através de um identificador (*handle_id*);
- **wait** expressa a espera da conclusão de uma ação anteriormente ativada de forma assíncrona, possivelmente bloqueando a *thread* do *workflow* em que foi em que foi executada;
- **cancel** representa o cancelamento da ativação de uma ação previamente ativada de forma assíncrona;
- **invoke** denota a ativação síncrona de uma ação, equivalente a uma invocação assíncrona (**start**) seguida diretamente por uma espera (**wait**).

Os combinadores de tarefas são:

- **sequence** denota a execução sequencial de uma lista de tarefas, na ordem em que

Figura 4: A Gramática XSD para a Linguagem de Orquestração



Fonte: Elaborado pelo autor.

são declaradas;

- **parallel** expressa a execução concorrente de um conjunto de tarefas, em que a tarefa combinadora termina depois de todas as tarefas internas terem sido concluídas (paradigma *fork-join*);
- **select** indica a execução de uma dentre um conjunto de tarefas. A tarefa escolhida é a primeira na sequência cuja ação de guarda está ativada no componente;
- **repeat** representa a execução iterativa de uma tarefa, em que essa iteração é encerrada quando um **break** é executado dentro do seu escopo.

A gramática XSD para especificar a sintaxe concreta do subconjunto de orquestração de SAFeSWL é representada na Figura 4. As operações primitivas da linguagem de orquestração são apresentadas no lado esquerdo. O arcabouço SAFe implementa uma classe para cada *tag* XML. Assim, em cada classe é definido o código que irá executar a semântica da tag XML.

Por exemplo, **invoke**, na sintaxe concreta, é representada por uma *tag* chamada <`invoke`>, do tipo `XML_SAFe_Action`. A Figura 5 exemplifica a linguagem de orquestração de SAFeSWL através do uso da *tag* <`invoke`> em um cenário hipotético que envolve dois componentes, Comp1 e Comp2. A orquestração consiste de uma ativação paralela de dois fluxos de sequência. Cada sequência executa sequencialmente invocações síncronas para *resolução, implantação* e *instanciação* do respectivo componente, através de ações

Figura 5: Exemplo de Orquestração de *Workflow* em XML

```
<parallel>
    <sequence>
        <invoke action="resolve" id_port="life-cycle-Comp1" />
        <invoke action="deploy" id_port="life-cycle-Comp1" />
        <invoke action="instantiate" id_port="life-cycle-Comp1" />
    </sequence>
    <sequence>
        <invoke action="resolve" id_port="life-cycle-Comp2" />
        <invoke action="deploy" id_port="life-cycle-Comp2" />
        <invoke action="instantiate" id_port="life-cycle-Comp2" />
    </sequence>
</parallel>
```

Fonte: Elaborado pelo autor.

disponíveis na sua porta do ciclo de vida, tornando-o pronto para execução.

2.4.2 Core

O Core é o elemento arquitetural responsável por manter o catálogo de componentes, através de serviços aos demais módulos da HPC Shelf (SAFe e Back-End) e a IDE's de programação de componentes. Seu principal serviço, de resolução de componentes, retorna, para cada componente presente em uma orquestração (*workflow*) a ser executado, uma lista de pares, em que cada par consiste em uma implementação do componente (componente concreto) e um componente *plataforma* hábil a executá-lo, ou seja, cada par corresponde a um componente de sistema. O usuário que solicitou a resolução, então, escolhe qual par configura a melhor opção, levando em consideração, por exemplo, no caso de componentes de computação, questões de desempenho, economia monetária, economia energética, etc.

Para a seleção de componentes, o Core implementa um *sistema de contratos contextuais* (ver Seção 2.5), o qual implementa uma estratégia de resolução de contratos contextuais. Esse mecanismo permite descobrir no catálogo, a partir de uma valoração dada à assinatura contextual de um componente abstrato por parte de um usuário, uma lista ordenada de componentes concretos que melhor implementam o contexto representado pela valoração fornecida. Por fim, o Core se comunica com o Back-End para instanciar componentes plataforma e componentes de computação sobre as plataformas reais às quais estão associados.

2.4.3 Back-End

O Back-End é o elemento arquitetural responsável por gerenciar as plataformas virtuais de um mesmo mantenedor, realizando a instanciação de plataformas virtuais sobre a infraestrutura computacional do mantenedor. O Back-End se comunica com o Core para informar os perfis de plataforma disponíveis, os quais são representados pela abstração de componentes da espécie *plataforma*.

De fato, cada mantenedor oferece um Back-End, ou seja, um Back-End para cada infraestrutura computacional mantida por ele disponível sob a nuvem. Cada Back-

End é capaz de alocar os recursos da infraestrutura computacional para instanciar plataformas virtuais sobre as quais componentes serão instanciados e executarão.

2.5 Contratos Contextuais

O Core, além de manter o catálogo dos componentes disponíveis para as aplicações, é responsável por abrigar um sistema de *contratos contextuais* baseado no HTS, sistema de tipos de componentes introduzido junto ao HPE (CARVALHO JUNIOR; REZENDE; SILVA; ALAM, 2016). Esse sistema tem por responsabilidade analisar os componentes abstratos presentes em uma orquestração e escolher, para cada um deles, uma lista ordenada de implementações possíveis (componentes concretos), calculada de acordo com um *algoritmo de resolução*. Para que a execução desse algoritmo seja possível, cada componente concreto deve possuir um *contrato contextual* associado.

Um contrato contextual é uma abstração que versa sobre requisitos funcionais e não-funcionais da aplicação, bem como características das plataformas de execução nas quais os componentes podem ser instanciados.

A respeito de requisitos sobre as plataformas de execução, considere como exemplo o caso em que seja interessante para a aplicação utilizar uma plataforma de computação paralela equipada com processadores gráficos GPU de um modelo e fabricante específicos. Logo, o algoritmo de resolução deve priorizar implementações que sejam capazes de utilizar esses recursos da forma mais otimizada possível. A título de ilustração desse cenário, considere a seguir a *assinatura contextual* do componente plataforma abstrato CLUSTER, o qual representa uma família de *clusters* computacionais na HPC Shelf:

CLUSTER [***processor_type*** $= P$: PROCESSORTYPE,
number_processors $= N$: INTEGER,
multicore_support $= M$: MULTICORESUPPORT,
accelerator_type $= A$: ACCELERATORTYPE [***multicore_support*** $= M$],
topology_type $= T$: TOPOLOGYTYPE,
memory_type $= MT$: MEMORYTYPE,
operating_system $= O$: OSTYPE]

Essa assinatura declara um conjunto de *parâmetros de contexto*, os quais são descritos a seguir:

- ***processor_type***: denota a classe de processadores presentes nos nós do *cluster*;
- ***number_processors***: contém a quantidade de nós de processamento presentes no *cluster*;
- ***multicore_support***: diz se os processadores dos nós suportam múltiplos fluxos reais de execução (*threads*);
- ***accelerator_type***: representa a família de aceleradores computacionais GPU presentes no *cluster*. Observe que esse parâmetro está associado ao parâmetro ***multi-***

core_ support, pela *variável de contexto M*, de forma que se os nós de processamento suportem execuções *multicore*, os aceleradores GPU presentes também devam suportar o disparo concorrente de múltiplos *kernels*, e vice-versa;

- ***topology_ type***: simboliza a topologia do barramento de interconexão dos processadores do *cluster*;
- ***memory_ type***: denota a classe de memórias principais disponíveis aos processadores do *cluster* (DRAM, EDO RAM, SDRAM, DDR SDRAM, RDRAM, etc. (STERLING; LUSK; GROPP, 2003));
- ***operating_ system***: delimita o sistema operacional que executa sobre o *cluster*.

ProcessorType, Integer, Float, MultiCoreSupport, AcceleratorType, TopologyType, MemoryType e OSType são qualificadores e, na assinatura contextual, são chamados de *limites* dos parâmetros de contexto.

Um contrato contextual (perfil de plataforma virtual) para essa assinatura contextual é retratado a seguir, o qual é associado a um componente plataforma concreto ClusterImpl:

ClusterImpl : Cluster [***processor_ type*** = XeonE5,
number_ processors = 128,
multicore_ support = MultipleCores,
accelerator_ type = Fermi [***multicore_ support*** = MultipleCores],
topology_ type = FatTree,
operating_ system = Linux]

Nesse contrato contextual, faz-se uma atribuição (valoração) aos parâmetros de contexto da assinatura contextual Cluster para representar uma plataforma virtual específica. Como se pode ver, ocorre uma valoração parcial, ou seja, nem todos os parâmetros de contexto foram valorados, fato que faz com que o algoritmo de resolução do não leve em consideração os parâmetros omitidos no cálculo de que implementações de um componente de computação melhor executarão sobre essa plataforma virtual. De forma resumida, esse contrato contextual representa uma plataforma virtual (*cluster*) que possui 128 processadores, os quais pertencem à família *XeonE5* e têm suporte *multicore*. Os aceleradores computacionais GPU pertencem à família *Fermi*[19]. Por fim, a topologia da rede de interconexão é *Fat Tree* e o sistema operacional é Linux.

Como se pode ver através do contrato contextual ClusterImpl, requisitos arquiteturais podem influenciar na escolha de componentes que implementam algoritmos computacionais. Em certos casos, a quantidade de memória disponível na plataforma pode privilegiar componentes que utilizem menos memória para armazenar suas estruturas de dados durante o processamento. Em outros casos, a pré-disposição, por parte da plataforma, de bibliotecas de passagem de mensagens ou de bibliotecas de computação intensiva pré-paralelizadas pode ser indispensável para alguns componentes. A topologia

[19] <http://www.nvidia.com.br/object/fermi-architecture-br.html>

de interconexão da rede também pode ser explorada, no sentido em que o componente paralelo possa privilegiar a troca de mensagens entre nós mais próximos na topologia. Por fim, a quantidade de nós de processamento disponível deve ser adequada, considerando as características de escalabilidade do algoritmo implementado pelo componente.

Por sua vez, requisitos da aplicação também podem influenciar na escolha da implementação de um componente. A exemplo disso, considere a assinatura contextual do componente de computação abstrato BLAS3, o qual representa componentes que implementam operações de terceiro nível (matriz-matriz) da biblioteca *Basic Linear Algebra Subprograms* (BLAS)[20](DONGARRA, 2002), utilizando paralelismo:

BLAS3 [***matrix_pattern*** = P : MatrixPattern,
matrix_format = F : MatrixFormat [***matrix_pattern*** = P],
number_processors = N : Integer,
multicore_support = M : MultiCoreSupport,
accelerator_type = A : AcceleratorType [***multicore_support*** = M]]

Dentre seus parâmetros de contexto, ressaltam-se ***matrix_pattern***, o qual afirma que implementações para esse componente devem explorar o padrão de elementos não-zero das matrizes para alcançar um melhor desempenho; ***matrix_format***, o qual representa o formato em que a matriz é representada e depende do padrão dos elementos da matriz; e ***number_processors***, que representa o número de unidades de processamento engajadas na execução, de forma que possam coexistir diferentes componentes, implementando algoritmos distintos de acordo com a quantidade de unidades de processamento empregadas na computação paralela.

Como um contrato contextual para BLAS3, tem-se, a seguir, BLAS3SparseBTUsingGPU, o qual representa um componente de computação concreto para BLAS3 voltado para matrizes esparsas, bloco-tridiagonais e armazenadas utilizando o formato CRS (*Compressed Row Storage*)[21]:

BLAS3SparseBTUsingGPU :
BLAS3 [***matrix_pattern*** = BlockTridiagonalMatrix,
matrix_format = CRSFormat [***matrix_pattern*** = BlockTridiagonalMatrix],
number_processors = 64,
multicore_support = MultipleCores,
accelerator_type = FermiArchitecture [***multicore_support*** = MultipleCores]

Pela definição do contrato, tem-se que os nós da plataforma de computação paralela que abrigará o componente BLAS3SparseBTUsingGPU devem ser no mínimo 64, ter múltiplos núcleos e serem equipados com GPUs pertencentes à arquitetura *Fermi*. Dessa maneira, o componente pode empregar OpenMP e CUDA para tirar vantagem das características arquiteturais.

Como se vê ainda nesse contrato, a definição do tipo de estrutura de dados

[20] <http://www.netlib.org/blas/>
[21] <http://netlib.org/linalg/html_templates/node91.html>

que será utilizada pela aplicação (matriz) é relevante na escolha da implementação do componente. Mais ainda, as propriedades da estrutura de dados podem ser levadas em consideração. Por exemplo, matrizes podem ser densas ou esparsas e, neste último caso, podem possuir padrões em relação à posição dos elementos não-nulos (diagonal, tridiagonal, bloco-diagonal, penta-diagonal, etc.). Outras questões que podem influenciar a escolha da implementação do componente dizem respeito ao problema que se deseja tratar, como, por exemplo, o tamanho da instância (ou da entrada), tendo em vista questões de escalabilidade, e a necessidade de algum algoritmo específico que o resolva por imposição da própria aplicação.

Para ilustrar melhor esse cenário, considere adiante um contrato contextual (valoração) para BLAS3 que foi criado pelo provedor de aplicações e submetido ao Core, a fim de encontrar no catálogo um componente que melhor se encaixasse ao contexto pretendido por ele:

BLAS3 [***matrix_pattern*** = TridiagonalSparseMatrix,
multicore_support = MultipleCores,
accelerator_type = KeplerArchitecture [***multicore_support*** = MultipleCores]]

Observando esse contrato, pode-se dizer que o provedor de aplicações busca uma implementação para BLAS3 que trabalhe com matrizes bloco-tridiagonais (esparsas), utilizando unidades de processamento *multicore*, cada uma equipada com um acelerador computacional GPU da família *Kepler*[22].

Observe que BLAS3SparseBTUsingGPU e a plataforma virtual ClusterImpl configuram um par válido para satisfazer o contrato submetido pelo provedor e estarão presentes na lista de pares obtida através do algoritmo de resolução do sistema de contratos contextuais. No caso de BLAS3SparseBTUsingGPU, veja que matrizes tridiagonais esparsas são casos especiais de matrizes bloco-tridiagonais, onde o tamanho do bloco é unitário. Logo, TridiagonalSparseMatrix é subtipo de BlockTridiagonalMatrix. Além disso, a arquitetura *Kepler* é compatível com a arquitetura *Fermi*, de forma que KeplerArchitecture é subtipo de FermiArchtecture.

Observe que o parâmetro de contexto numérico ***number_processors*** não teve valor especificado pelo provedor de aplicações, mas existe tanto no contrato BLAS3SparseBTUsingGPU quanto no contrato ClusterImpl. Por se tratar de um parâmetro de contexto numérico, não existe uma relação de subtipos para ele, mas sim uma ordem crescente entre números inteiros ($<$). Dessa forma, como BLAS3SparseBTUsingGPU requer 64 unidades de processamento e ClusterImpl possui 128 disponíveis, isto faz com que os dois possam formar um *componente de sistema* ($64 < 128$) candidato para a requisição do provedor de aplicações.

[22] <http://www.nvidia.com/object/nvidia-kepler.html>

2.6 Um Arcabouço de Certificação de Componentes para a HPC Shelf

Nesta seção, descrevem-se as contribuições deste trabalho sobre a plataforma HPC Shelf. Inicialmente, um novo objetivo foi proposto à nuvem:

> Fornecer um mecanismo que permita a construção de aplicações mais seguras, no que tange a fazerem o que se propõem a fazer e não apresentarem comportamentos inesperados durante a execução.

Esse objetivo é alcançado por meio de um arcabouço de certificação de componentes (DANTAS; CARVALHO JUNIOR; BARBOSA, 2017a; DANTAS; CARVALHO JUNIOR; BARBOSA; PROENÇA, 2017) através do qual componentes certificadores podem ser criados e empregados como serviço para verificação de propriedades de interesse em outros componentes de interesse de aplicações.

O mecanismo de certificação proposto introduz uma nova espécie de componentes, chamados componentes *certificadores*, os quais, quando ligados a componentes de outras espécies no código SAFeSWL de um sistema de computação paralela, permitem a verificação de suas propriedades formais. A ligação entre um componente certificador e um componente que deve certificar tem o nome de *binding de certificação*, um novo tipo de *binding*, que se junta aos *bindings* de serviços e de ações, e será melhor contextualizada nas seções que se seguem. Uma outra espécie de componentes, chamados de componentes *táticos*, também é proposta para suprir as necessidades de componentes certificadores de acessar as funcionalidades de diferentes infraestruturas de verificação. Portanto, os componentes táticos encapsulam e oferecem acesso, de fato, a infraestruturas de prova e verificação que podem ser orquestradas pelos certificadores com o intuito de verificar propriedades formais de componentes.

2.6.1 Sistema de Certificação Paralela

Um *sistema de certificação paralela* é composto pelos seguintes elementos:

- Um conjunto de plataformas virtuais, ditas *plataformas de certificação*;
- Um conjunto de um ou mais componentes táticos, possivelmente alocados a plataformas de certificação distintas;
- Um componente certificador, responsável pela orquestração dos componentes táticos através de um fragmento da linguagem TCOL (descrita posteriormente) em uma tarefa de certificação;
- Um conjunto de componentes a serem certificados, ligados ao componente certificador através de ligações de certificação.

Sistemas de certificação paralela são calculados a partir do código SAFeSWL de sistemas de computação paralela, e são responsáveis pela execução de procedimentos de certificação sobre um conjunto de componentes ligados a um mesmo componente

certificador. Note sua analogia com sistemas de computação paralela, encontrando-se o componente certificador em posição análoga ao componente workflow e os componentes táticos aos componentes de solução. Por isso, o componente certificador não está associado a uma plataforma virtual, sendo executado diretamente no SAFe, assim como os componentes *workflow*.

2.6.2 Componentes Táticos

Um componente tático representa uma infraestrutura de prova, ou seja, pode ser composto por uma combinação adequada das ferramentas de verificação que serão elencadas no Capítulo 3 e é capaz de realizar um fluxo de execução que vai do recebimento de um código escrito na linguagem que compreende, executar validações, conversões e, por fim, verificar propriedades sobre tal código.

2.6.2.1 Portas de Componentes Táticos

Um componente tático possui por padrão as seguintes portas:

- uma porta de serviços usuária, com operações para receber do componente certificador os programas do componente a ser certificado, possivelmente previamente traduzidos pelo componente certificador para a linguagem que a ferramenta de verificação do componente tático compreende; receber do componente certificador propriedades formais a serem verificadas nesses programas; permitir ao componente certificador monitorar o progresso da verificação das propriedades; e retornar para o certificador o resultado do processo de verificação das propriedades;
- uma porta de ações, chamada de *porta de verificação*, ou *Verify*, contendo as ações **verify_perform**, **verify_conclusive** e **verify_inconclusive**;
- a porta de ações padrão que permite que o componente certificador guie seu ciclo de vida, que exporta as ações **resolve**, **deploy**, **instantiate** e **release**.

Quando **verify_perform** é ativada, o componente tático inicia o fluxo de execução do seu processo de verificação com relação às propriedades a ele atribuídas. Ao fim desse processo, é ativada **verify_conclusive**, quando o resultado da verificação foi conclusivo para todas as propriedades (true ou false), ou **verify_inconclusive**, significando que uma ou mais propriedades tiveram verificação inconclusiva (null). A verificação de uma propriedade é inconclusiva quando o componente tático é impedido, de alguma forma, de aplicar sua técnica de verificação para provar ou refutar a propriedade. Exemplos disso podem ser quando há alguma falha de infraestrutura na plataforma que abriga o componente tático, quando a propriedade é escrita em um formato que não é compreendido pelo componente tático ou quando o *timeout* de verificação configurado na ferramenta de verificação é atingido.

2.6.3 Portas e Ligações de Componentes Certificadores

Para que um componente seja certificado, deve ser escolhido no catálogo um ou mais componentes certificadores e realizar entre eles *ligações de certificação*. Essas ligações são estabelecidas dentro da definição arquitetural do componente no SAFe.

Para se comunicar com o componente a ser certificado, um componente certificador possui uma única porta provedora com as seguintes características:

- através dessa porta, o SAFe passa para o componente certificador os programas do componente, propriedades formais *ad hoc* (não geradas pelo próprio componente certificador nem anotadas diretamente nos programas), caso existam, e outras informações relevantes ao processo de certificação;
- através de métodos disponíveis nesta porta, o processo de certificação do componente feito pelo componente certificador pode ser iniciado pelo SAFe.

Um componente certificador possui também portas que são contrapartes das portas de verificação dos seus componentes táticos associados. Quando o SAFe solicita ao componente certificador que inicie o processo de certificação, o componente certificador executa a orquestração definida no fragmento TCOL relacionado, a qual, em algum momento, ativará ações **verify_perform** relativas aos componentes táticos associados. Quando o componente tático ativa a ação **verify_perform**, é iniciado o processo de verificação das propriedades formais. Quando um componente tático ativa **verify_conclusive** e a ação de mesmo nome é ativada no certificador, o certificador acessa a porta de serviços do componente tático a fim de obter o resultado da verificação das propriedades e fazer a devida contabilização de quais foram provadas ou não.

Por fim, quando **verify_inconclusive** é ativada no componente tático e o mesmo ocorre no certificador, o certificador acesse a porta de serviços do componente tático para obter o diagnóstico relatado. Ressalta-se que o componente certificador, caso tenha sido programado com essa capacidade, pode solicitar, automaticamente ou com a permissão da aplicação, ao Core que, se possível, instancie o mesmo componente tático em outra plataforma virtual. Caso isso seja possível, o certificador então reinicia o processo de verificação relativo ao componente tático que falhou. Caso não, o certificador emite uma mensagem à aplicação e continua seu fluxo de execução normal, porém considerando as propriedades de responsabilidade do componente tático que falhou como *inconclusivas*.

2.6.4 A Linguagem de Orquestração de Componentes Táticos (TCOL)

A aplicação automática de diferentes componentes táticos para verificar propriedades sobre um conjunto de programas pode potencializar o processo de certificação. Por exemplo, uma propriedade específica que não pôde ser provada por um componente tático pode ser provada por outros. Mais ainda, a ativação paralela de componentes táticos para verificar diferentes propriedades pode diminuir o tempo final de verificação.

Figura 6: Sintaxe abstrata da linguagem TCOL

TASK ::= **skip** | **invoke** *action* | **start** *handle action* | **wait** *handle* | **cancel** *handle* |
sequence TASK+ | **parallel** TASK+ | **repeat** TASK | **break** | **continue** |
select {*action*: TASK}+ | **switch** {*condition*: TASK}+

Fonte: Elaborado pelo autor.

Com base nesse cenário, propõe-se uma linguagem que permita a orquestração de um conjunto de componentes táticos por um componente certificador, denominada *Linguagem de Orquestração de Componentes Táticos* (TCOL). A ideia é que, no momento do registro do componente certificador concreto no Core, seja também fornecido um código de orquestração escrito nessa linguagem, o qual guiará a ativação das ações **verify_perform** dos componentes táticos por parte do certificador.

No protótipo corrente da arquitetura proposta, TCOL é uma extensão da linguagem de *workflows* da HPC Shelf, SAFeSWL. A extensão que se faz é para adicionar uma nova tarefa de seleção, **switch**, a qual é composta por uma sequência de pares <*condição*, *tarefa*>, em que a tarefa escolhida é a primeira na sequência cuja condição avalie para true. A Figura 6 apresenta a sintaxe abstrata de TCOL.

A gramática de *condições* de TCOL é similar às gramáticas de condições de linguagens orientadas a objetos no estilo C-*like*, tais como C++, Java ou C#. Logo, optou-se por não mostrá-la aqui. Adiante, apresenta-se o conjunto inicial de variáveis que podem aparecer nas condições. Essas variáveis são gerenciadas pelo componente certificador e têm seus valores atualizados por ele durante o processo de certificação. São elas:

- **formal_properties**: um vetor associativo que, para cada código (*script* SAFeSWL do *workflow* ou programa de componente computação) e propriedade formal verificada sobre ele, associa um dos valores: null, para dizer que a propriedade é inconclusiva, ou seja, que nenhum componente tático tentou ou conseguiu verificar (provar ou refutar) a propriedade; false, se a propriedade foi verificada por algum componente tático mas foi refutada; e true, se algum componente tático conseguiu prová-la;
- **tc_applied**: um vetor associativo que, para cada nome de componente tático presente na orquestração do certificador, afirma se ele já foi acionado para verificar as propriedades formais que lhe foram atribuídas (false ou true);
- Parâmetros de Contexto: é possível utilizar nas condições os valores atribuídos aos parâmetros de contexto do componente certificador abstrato pelo contrato contextual do componente certificador concreto. É possível utilizar ainda os valores atribuídos aos parâmetros de contexto dos componentes táticos abstratos associados ao componente certificador concreto pelos contratos contextuais dos respectivos componentes táticos concretos.

2.6.5 Atores

Nessa seção, discute-se sobre papéis dos atores já existentes da HPC Shelf com relação ao mecanismo de certificação de componentes. Além disso, propõe-se um novo ator: o *certificador de componentes*.

2.6.5.1 Usuário Especialista

Uma vez que usuário especialista não precisa ter consciência do caráter baseado em componentes da aplicação da HPC Shelf a qual está interessado, tampouco do fato de que essa aplicação usa a HPC Shelf como plataforma de execução das soluções computacionais por ela construídas, o mecanismo de certificação é completamente transparente para ele.

Entretanto, provedores de aplicações que expõem suas aplicações enfatizando o caráter baseado em componentes, ou mesmo levando a abstração de componentes ao nível do usuário especialista, podem utilizar a característica de certificação para convencer os especialistas de que sua aplicação é mais segura, podendo ser esse um critério usado pelo especialista para escolher entre aplicações alternativas para um mesmo propósito.

2.6.5.2 Provedor de Aplicações

Ao desenvolver sistemas de computação paralela descritos em SAFeSWL, os quais são chamados de *workflows*, o provedor pode decidir utilizar componentes ditos *certificáveis*, ou seja, que só executam após passarem por um processo de certificação antes da execução através de um componente certificador conectado à sua porta de certificação através do código SAFeSWL.

A certificação de um componente de computação por parte de um certo componente certificador é uma operação idempotente. Portanto, não precisa ser realizada várias vezes em diversas aplicações. De fato, a certificação de um componente certificável por parte de um certificador é registrada pelo Core por meio de um certificado inforjável, de modo que não precisa ser realizada posteriormente.

A execução do procedimento de certificação de um componente é realizada através da inclusão de uma nova ação, chamada *ação de certificação*, ou **certify**, na porta de ciclo de vida do componente, quando se trata de um componente certificável. Obviamente, a ativação da ação de certificação, no código SAFeSWL, deve ser realizada após a resolução do contrato contextual componente (ativação da ação **resolve**). Caso o componente escolhido ainda não tenha sido certificado pelo certificador escolhido, o procedimento de certificação é executado através da inclusão do componente no sistema de certificação paralela do componente certificador, caso já exista um instanciado, ou na criação e instanciação de um, caso contrário.

2.6.5.3 Desenvolvedor de Componentes

O desenvolvedor pode agora fornecer componentes que podem ser certificados através de sistemas de certificação paralela, ditos componentes certificáveis. Para tanto, ele deve possuir conhecimento sobre técnicas de verificação formal de *software* suportadas pelos componentes certificadores que deseja que certifiquem os componentes certificáveis que deseja que sejam registrados na HPC Shelf.

Para tornar um componente certificável, o desenvolvedor deve aninhar à configuração do componente em questão um componente certificador adequado para a espécie do componente certificável, determinando seu contrato contextual. Para um sistema de computação paralela, esse componente certificador aninhado será entendido como uma ligação de certificação, ficando a cargo do provedor de aplicações conectar um certificador cujo contrato seja compatível com o contrato do certificador aninhado.

Um componente criado como certificável pode ser certificado tanto em tempo de execução do sistema de computação paralela em produção (*workflow*), quanto em tempo de desenvolvimento, através de um sistema de computação paralela criado através de uma aplicação da HPC Shelf especialmente destinada à prototipação e certificação (aplicação de certificação).

Na aplicação de certificação, o desenvolvedor pode certificar o componente usando diferentes certificadores, bem como modificar a implementação do componente de forma a satisfazer vários certificadores. Além disso, o desenvolvedor pode interagir com componentes táticos durante a prova, nos casos desses demandarem por auxílio interativo de um especialista durante o processo de verificação. Essa é uma funcionalidade importante que, em geral, não poderá ser utilizada pelos provedores de aplicações, que, mesmo interessados em usar componentes certificáveis, não são obrigados a serem especialistas em certificação. Por outro lado, os desenvolvedores de componentes certificáveis, além de terem conhecimento sobre técnicas de verificação formal, possuem o conhecimento sobre o código do componente, necessário para provas interativas, o que não é acessível por parte dos provedores.

2.6.5.4 Certificador de Componentes

Esse novo ator é um especialista em métodos formais de verificação de *software*, além de possuir conhecimento sobre a arquitetura dos componentes certificadores. Seu principal objetivo é criar componentes certificadores e componentes táticos. Suas especificidades se resumem a:

- Criar componentes táticos que encapsulam ferramentas de verificação, as quais já devem estar instaladas nas plataforma virtuais alvo no momento da sua implantação. Os parâmetros de contexto dos componentes táticos abstratos ficam a critério do certificador de componentes, que pode, inclusive, não incluir nenhum parâmetro adicional além dos parâmetros de plataforma comuns a todos os componentes da

HPC Shelf, fazendo com que para o componente tático abstrato exista um único componente tático concreto, sempre escolhido pelo mecanismo de resolução de contratos contextuais. Como exemplo de parâmetros que usuários certificadores podem acrescentar ao contexto de componentes táticos, pode-se citar a versão da ferramenta de verificação utilizada pelo componente tático. Dessa forma, poderia existir um componente tático abstrato representando uma determinada ferramenta de verificação e diversos componentes táticos concretos, representando diferentes versões dessa ferramenta. Através da relação de subtipos do sistema de contratos contextuais, é possível expressar a retrocompatibilidade entre diferentes versões. Por exemplo, se um ator necessita de um componente certificador com um componente tático específico cuja ferramenta de verificação possui versão 5.0.0, mas que não foi escolhido pelo sistema de resolução a partir do contexto fornecido, o sistema pode escolher outro componente com ferramenta de verificação de versão compatível, como por exemplo a 6.2.2 ($6.2.2 :< 5.0.0$), admitindo-se que haja retrocompatibilidade entre as versões 6.x.x e 5.x.x da ferramenta;

- Associar componentes táticos ao componente certificador concreto que está a criar. Ao criar um componente certificador concreto, componentes táticos abstratos podem ser associados e, para cada um, deve ser fornecida uma valoração à sua assinatura contextual de forma que os componentes táticos concretos de interesse estejam presentes na lista retornada quando o componente tático for resolvido no SAFe durante a orquestração feita pelo certificador concreto. A ordem de ativação dos componentes táticos para verificar as propriedades que lhes competem é guiada por um código de orquestração na linguagem TCOL, que deve ser incorporado ao certificador concreto;
- Criar componentes certificadores, capazes de orquestrar um conjunto de componentes táticos por meio de TCOL;
- Durante o processo de certificação de um componente, o conjunto de propriedades a serem verificadas, gerado pelo componente certificador, é dividido em *pacotes* e distribuído entre seus componentes táticos. A implementação de quais propriedades serão geradas e como será a distribuição desses pacotes é de responsabilidade do certificador de componentes. Cada componente tático, por sua vez, ao receber seu pacote, pode distribuí-lo entre seu conjunto de unidades a fim de acelerar a verificação. Na prática, o componente certificador já divide o pacote de propriedades do componente tático e as submete diretamente às unidades dos componentes táticos envolvidos. Dado que foi escolhida para o componente tático uma plataforma virtual com n unidades de processamento, o Back-End responsável pela plataforma virtual instancia n unidades do componente tático, as quais recebem do certificador o conjunto de propriedades formais a serem verificadas. De posse de um conjunto de propriedades formais, cada unidade do componente tático pode disparar em paralelo

instâncias da ferramenta de verificação associada ao componente tático para verificar cada propriedade. Assim, caso as unidades de processamento da plataforma virtual possuam m núcleos, cada unidade do componente tático pode disparar m instâncias da ferramenta de verificação paralelamente, através de *threads*, cada uma verificando uma propriedade distinta.

2.6.6 Elementos Arquiteturais da HPC Shelf

Dentre os elementos arquiteturais da HPC Shelf, o SAFe e o Core terão novas funcionalidades necessárias à implementação da arquitetura dos componentes certificadores.

2.6.6.1 SAFe

Neste trabalho, atribui-se mais uma função ao SAFe, que é a criação de sistemas de certificação paralela para certificar componentes que fazem parte do sistema de computação paralela especificado em SAFeSWL. Para isso, para cada componente certificável, o qual possui uma porta de certificação e uma ação adicional, chamada **certify**, na sua porta de ações *LifeCycle*, deve ser associado um componente certificador registrado no catálogo de componentes da HPC Shelf, cujo contrato contextual deve ser compatível (subtipo) com o contrato contextual determinado pelo desenvolvedor do componente.

A ativação da ação **certify** pode ocorrer em qualquer momento do ciclo de vida do componente de computação, dando a liberdade de escolha desse momento ao ator que programou a aplicação. Porém, caso não seja ativada antes da ação de instanciação do mesmo componente, a ação de instanciação permanece bloqueada até que alguma *thread* da orquestração ative a ação de certificação do componente em questão. Observe, portanto, que a certificação é estática, uma vez que não acontece no momento da execução das ações computacionais dos componentes. O momento mais comum para ativação da ação **certify** de um componente ocorre depois que a ação **resolve** foi ativada nesse componente. Após a ativação de **resolve**, o componente computação abstrato já foi resolvido, o componente computação concreto pretendido foi escolhido e foi calculada uma plataforma virtual para ele. Sabe-se que serão instanciadas tantas unidades paralelas do componente computação concreto quantas forem as unidades de processamento da plataforma virtual alvo. Essa informação é importante ao mecanismo de certificação, uma vez que alguns componentes certificadores que utilizam componentes táticos equipados com certos *model checkers* necessitam saber previamente quantas unidades dos componentes que certificam serão instanciadas, uma vez que tais ferramentas exigem essa informação para reduzir o espaço de busca do *model checking* durante a verificação de propriedades formais nos programas.

A execução da ação de certificação de um componente certificável causa a inclusão desse componente no sistema de certificação paralela do componente certificador, caso já exista um instanciado, ou cria e instancia um, caso não. Caso a certificação

termine com sucesso (todas as propriedades *obrigatórias* foram provadas), o componente certificável recebe um certificado para aquele certificador, que é lembrada no catálogo de componentes da HPC Shelf, tornando-se um componente certificado segundo o certificador em questão. Caso a certificação falhe, por ser inconclusiva ou alguma das propriedades obrigatórias serem falsas, a tentativa de instanciação do componente falha, causando um erro na execução do *workflow* e seu abortamento [23].

2.6.6.2 Core

O Core passa a incluir no catálogo componentes certificadores e táticos. Por consequência, sua funcionalidade de resolução de componentes passa agora também a atuar em sistemas de certificação paralela, através da resolução de componentes certificadores abstratos presentes neles, e da resolução de cada componente tático abstrato orquestrado por um componente certificador.

O Core assume também a responsabilidade pelo gerenciamento da certificação de componentes certificáveis, de forma que o processo de certificação seja aplicado apenas uma vez para um determinado componente por um certo componente certificador. Isso se dá através de um sistema de emissão de *certificados* associados a componentes certificáveis que passaram anteriormente, com sucesso, pelo processo de certificação.

2.7 Considerações Finais

Neste capítulo, apresentaram-se os conceitos, atores envolvidos e os elementos arquiteturais da nuvem HPC Shelf. Dentre os conceitos abordados, ressalta-se o modelo de componentes inerentemente paralelos Hash, o qual permite a composição de aplicações utilizando componentes de diversas espécies hierarquicamente. Destaca-se também o sistema de contratos contextuais, o qual é capaz de aplicar um algoritmo de resolução para descobrir dentre os componentes catalogados no sistema os que melhor se adequam a contextos requeridos pelos usuários, os quais são dispostos através da abstração de contratos contextuais.

Dentre os atores, evidenciam-se os papéis principais do provedor de aplicações, do desenvolvedor de componentes e do certificador. O primeiro busca utilizar sua IDE de programação (Java ou C# + SAFe) para implementar uma orquestração utilizando a linguagem SAFeSWL que seja apta a resolver um problema de interesse da aplicação. O segundo, por sua vez, utiliza o HPE para desenvolver componentes abstratos (assinaturas contextuais) e componentes concretos (contratos contextuais). O terceiro, enfim, possui incumbência semelhante à do segundo, entretanto, criando componentes certificadores e táticos que podem atuar na certificação de outros componentes.

[23] Para lidar com essas situações, pretende-se desenvolver, em trabalhos futuros, um sistema de tratamento de exceções para orquestrações SAFeSWL.

Dentre os elementos arquiteturais, ressaltam-se o Core e o SAFe. O primeiro realiza as tarefas mais importantes da nuvem, que são armazenar o catálogo dos componentes disponíveis para as aplicações, guiar o ciclo de vida dos componentes nas plataformas computacionais alvo e abrigar o sistema de contratos contextuais. Já o segundo propicia um ambiente robusto para a construção de aplicações padronizadas que acessam os componentes de alto desempenho, bem como rege a lógica de execução de *workflows* e certificadores utilizados nas aplicações.

Por fim, as contribuições deste trabalho para a HPC Shelf foram introduzidas, elencando os impactos do arcabouço de certificação proposto para os seus diferentes atores e elementos arquiteturais. Nos capítulos 4 e 5, serão apresentadas, respectivamente, as especificidades de dois componentes certificadores propostos como contribuição desta Tese, chamados SWC2 e C4, respectivamente para *workflows* e componentes de computação. Antes disso, o Capítulo 3 apresentará uma revisão geral sobre técnicas e ferramentas para verificação formal de *software*, necessárias para a proposta de componentes certificadores.

3 Verificação Formal de *Software*

O interesse em desenvolver componentes táticos para os componentes certificadores SWC2 e C4, os quais são voltados, respectivamente, à certificação de *workflows* e componentes de computação na HPC Shelf, incutiu a necessidade de realizar-se um estudo exploratório relacionado à Verificação Formal de *Software*. Esse estudo se baseou na revisão de artigos, livros e relatórios disponíveis nas bases de artigos científicos ACM Digital Library[24], IEEE Explorer[25], Science Direct[26] e Scopus[27], e também na *internet*.

Trabalhos em três áreas foram analisados: *semântica de programas*, *lógicas para especificação de programas* e *técnicas de verificação*. Todavia, para as finalidades desta Tese, optou-se por dividir este capítulo em duas seções, uma voltada à técnicas de verificação formal de propriedades sobre *workflows* e outra voltada à verificação formal de propriedades sobre programas paralelos contidos em componentes de computação.

3.1 Verificação Propriedades Formais em *Workflows*

Existem diversos formalismos com semânticas bem definidas que permitem raciocinar sobre *workflows*. Dentre eles, podem-se destacar as Redes de *Petri* (REISIG, 1985; PETRI, 1962) e os *Sistemas de Transição* (WINSKEL; NIELSEN, 1995). Para capturar a semântica dos *workflows* da HPC Shelf, adotaram-se os últimos. A justificativa para essa escolha será apresentada ao longo do texto.

3.1.1 Semântica de Programas

Do ponto de vista da implementação, as tarefas (orquestrações de conjuntos de ações) dos *workflows* da HPC Shelf são vistas como processos (programas) concorrentes. A semântica de programas concorrentes é objeto de estudo das *álgebras de processos*.

Álgebras de processos representam processos de um sistema computacional como termos algébricos a fim de permitir um raciocínio matemático sobre eles (PARROW, 2001; BAETEN; BASTEN; RENIERS, 2010; FOKKINK, 2013). Tais álgebras focam na especificação e manipulação de termos (processos), utilizando uma coleção de operadores, os quais podem ser *básicos*, para construir processos finitos, operadores de *comunicação*, para expressar concorrência, e operadores de *recursão*, para expressar comportamentos infinitos. Uma grande potencialidade da álgebra de processos recai no fato de permitir expressar nativamente algumas propriedades interessantes, tais como a ausência de *deadlock* e a verificação de que um comportamento observável e saídas do sistema estão de acordo, respectivamente, com algum estímulo externo e entradas a ele fornecidas.

[24] <http://dl.acm.org>
[25] <http://ieeexplore.ieee.org>
[26] <http://www.sciencedirect.com>
[27] <http://www.scopus.com>

Os fundamentos da álgebra de processos foram desenvolvidos paralelamente por Milner, Hoare, Bergstra e Klop, entre as décadas de 70 e 80. Milner criou a álgebra de processos CCS (*Calculus of Communicating Systems*) (MILNER, 1989). Por sua vez, Hoare introduziu o CSP (*Communicating Sequential Processes*) (HOARE, 1985; ROSCOE, 2010). Bergstra e Klop, finalmente, propuseram a ACP (*Algebra of Communicating Processes*) (BERGSTRA; KLOP, 1984), a qual é fortemente relacionada à CCS. As álgebras de processos são aplicadas geralmente como pano de fundo para outros formalismos, como por exemplo os sistemas de transição, a fim de melhor capturar a semântica de interação entre processos.

Além das álgebras de processos, outro fato que faz com que a semântica dos *workflows* da HPC Shelf seja melhor capturada pelos sistemas de transição é que esses sistemas fornecem formalismos para expressar a semântica comportamental e composicional de sistemas reativos. De fato, os processos dos *workflows* da HPC Shelf (tarefas) compõem um sistema reativo, uma vez que apenas reagem a estímulos provocados por outros processos.

Como ferramenta baseada em sistemas de transição, escolheu-se o *software* mCRL2 (*Micro Common Representation Language 2*)[28], que utiliza uma álgebra de processos reminiscente da ACP (BAETEN; BASTEN; RENIERS, 2010) para modelar o comportamento de sistemas concorrentes e protocolos. Em mCRL2, processos são construídos a partir de um conjunto de ações declaradas pelo usuário e um conjunto de combinadores de ações e processos. Os últimos incluem a composição de *multi-ações*, composições *sequenciais*, *alternativas* e *paralelas* e operadores de abstração (*reetiquetagem*, *ocultação* e *restrição*). Ações podem ser parametrizadas com dados (números inteiros, por exemplo) e construtores condicionais tornam possível especificar comportamentos condicionais baseados nesses dados em processos. Isso permite modelar sistemas cujo comportamento dependa crucialmente dos dados trocados. Os dados são definidos em termos de tipos abstratos de dados equacionais (SANNELLA; TARLECKI, 2011). Por fim, a semântica dos processos é dada pela associação de um sistema de transições, chamado de Sistema de Transições Etiquetadas (ou LTS - *Labelled Transition System*, abordado mais adiante) a cada expressão da linguagem.

O mCRL2 provê uma lógica modal expressiva com operadores de ponto fixo, estendendo o μ-calculus modal proposicional de Kozen (KOZEN, 1983), com variáveis de dados e quantificação sobre domínios de dados (ver Seção 4.5). A flexibilidade alcançada pela combinação dos operadores de ponto fixo minimal e maximal com combinadores modais permite a especificação de propriedades complexas, entretanto geralmente através de fórmulas complicadas de se entender. Felizmente, o mCRL2 permite o uso de expressões regulares sobre o conjunto de ações como possíveis etiquetas de ambas as modalidades de necessidade e eventualidade. Por exemplo, a expressão $\langle c.(a+b)^* \rangle \phi$ afirma a existência

[28] <http://mcrl2.org/>

de uma sequência possivelmente infinita de ações casando com a expressão regular (c, seguido de uma sequência possivelmente infinita de a's e b's).

O uso de expressões regulares provê um conjunto de macros para especificação de propriedades que são suficientes na maioria dos casos práticos. Por exemplo, uma propriedade de segurança afirmando que um *workflow* não deve exibir uma sequência de ações casando com a expressão regular e é simplesmente escrita como $[e]\,\mathsf{false}$. Dualmente, para se afirmar que deve existir tal sequência (propriedade de continuidade) pode-se escrever $\langle e \rangle\,\mathsf{true}$. Note que a fórmula do parágrafo anterior poderia ter sido escrita como $\nu X\,.\,\langle c.(a+b)^* \rangle\, X$, utilizando o operador de ponto fixo maximal ν. As propriedades padrão dos certificadores de *workflows* (Capítulo 4) ilustram o uso do mCRL2 na especificação de propriedades fundamentais dos *workflows* da HPC Shelf.

Outros fatos que levaram à escolha do mCRL2 são:

- O mCRL2 foi construído especialmente para trabalhar com sistemas concorrentes, como os *workflows* da HPC Shelf;
- É altamente difundido na comunidade acadêmica de Métodos Formais;
- É constituído de um conjunto de mais de 60 ferramentas que podem ser utilizadas em conjunto para visualização, simulação, minimização e *model checking* de sistemas concorrentes. Essa integração de ferramentas se mostrou bastante produtiva pelo fato de evitar conversões desnecessárias entre modelos e linguagens.

Uma observação importante que deve ser feita é que, devido a questões de espaço, não será feita aqui uma apresentação das sintaxes da álgebra de processos e da lógica modal do mCRL2. Dessa forma, o leitor é referido a (GROOTE et al., 2007) e a uma vasta documentação contida no *website* da ferramenta.

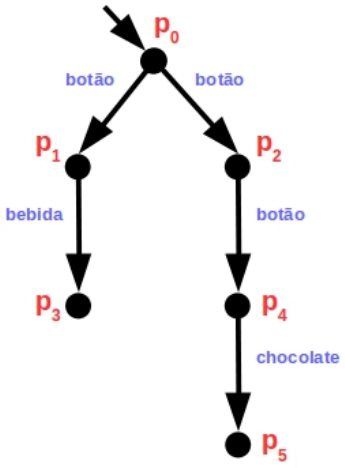
3.1.1.1 Sistemas de Transições Etiquetadas (LTS)

Sistemas de Transições Etiquetadas (LTS) constituem uma semântica poderosa para raciocinar sobre processos, permitindo especificação, implementação e testes (TRETMANS, 2008). Um LTS consiste de uma estrutura composta por estados com transições entre eles, etiquetadas com ações. Os estados modelam os estados do sistema e as transições etiquetadas modelam as ações que o sistema pode executar. Formalmente, um LTS é definido por uma quíntupla $A = (S, Act, \longrightarrow, s, T)$, em que:

- S é um conjunto contável, não-vazio de estados;
- Act é um conjunto contável de ações, sendo possivelmente multiações (ações que podem executar paralelamente);
- $\longrightarrow \subseteq S \times Act \times S$ é a relação de transição;
- $s \in S$ é o estado inicial;
- $T \subseteq S$ é o conjunto de estados terminais.

É comum representar $t \xrightarrow{a} t'$ para dizer que $(t, a, t') \in \longrightarrow$. No caso em que se tenha uma transição $t \xrightarrow{a} t'$ seguida de uma transição $t' \xrightarrow{a'} t''$, pode-se uti-

Figura 7: Um Sistema de Transições Etiquetadas



Fonte: Elaborado pelo autor.

lizar a notação de composição de transições, na forma $t \xrightarrow{a.a'} t''$. A título de ilustração, considere a Figura 7, a qual representa um sistema de transições etiquetadas $\langle\{p_0, p_1, p_2, p_3, p_4, p_5\}, \{botão, bebida, chocolate\}, \{\langle p_0, botão, p_1\rangle, \langle p_0, botão, p_2\rangle, \langle p_1, bebida, p_3\rangle, \langle p_2, bot$ para uma máquina de vendas. Nele, tem-se, por exemplo, $p_0 \xrightarrow{botão.botão.chocolate} p_5$ e $p_0 \xrightarrow{botão.chocolate} p_5$.

Destaca-se que, comumente, o conjunto de ações *Act* contém uma ação τ cujo comportamento não é observado pelo ambiente do sistema (ação interna). Nesses casos, *Act* é composto por $L \cup \{\tau\}$, em que L representa o conjunto das ações observáveis pelo sistema. Como exemplo, suponha que se tivesse, no LTS da Figura 7, a configuração $p_0 \xrightarrow{botão.\tau.bebida} p_3$. Ela seria percebida pelo ambiente como $p_0 \xrightarrow{botão.bebida} p_3$.

Exceto para processos triviais, uma representação de um LTS através de grafos, como o da Figura 7, ou através da quíntupla definida anteriormente, não é adequada. Em sistemas reais, é comum se observar bilhões de estados, fato que impossibilita sua enumeração. Uma maneira mais eficiente para expressar sistemas LTS é através de sua semântica operacional. Essa semântica operacional é definida por uma linguagem, com termos, operadores e expressões e é chamada de *linguagem de processos* ou de *linguagem das expressões de comportamento*. Essa linguagem é definida como se segue:

$$B \ ::= \ a\,;B \mid i\,;B \mid \mathbf{\Sigma}\mathcal{B} \mid B\,|[G]|\,B \mid \mathbf{hide}\ G\ \mathbf{in}\ \ B \mid P$$

- $a\,;B$ corresponde a uma *expressão com prefixo de ação* e define o comportamento que, primeiro executa a ação observável $a \in L$, depois se comporta como B. Em outras palavras, $a\,;B$ define um LTS que executa a ação a no LTS B.
- $i\,;B$ é o mesmo que $a\,;B$, com exceção de que i denota a ação interna τ.
- $\mathbf{\Sigma}\mathcal{B}$ é a expressão que denota a *escolha de comportamento*. Se comporta como um dos processos contidos no conjunto $\mathcal{B}$. Quando se tem somente dois processos, é comum usar $B_1 \square B_2$ em vez de $\mathbf{\Sigma}\{B_1, B_2\}$. Usa-se também a abreviação **stop** para dizer que não há nenhuma ação a tomar, ou seja, tem-se $\mathbf{\Sigma}\emptyset$.

- $B_1 \,|\, [G] \,|\, B_2$ representa a *expressão paralela* e denota a execução paralela de B_1 e B_2. O conjunto G de ações denota as ações em que B_1 e B_2 devem sincronizar. Por exemplo, se B_1 está a executar uma ação a, ele deve esperar até que B_2 também execute a. Quando $G = \emptyset$, tem-se uma intercalação completa de ações e quando $G = L$, todas as ações observáveis sincronizam. Utiliza-se $|||$ para representar a primeira situação e $||$ para representar a última.
- **hide** G **in** B significa a *expressão de ocultação* e quer dizer que todas as ações em G não são observáveis, ou seja, são substituídas por τ.
- P é a *expressão de definição e instanciação de processos*. Pode ser usada na forma $P :== B$ para associar um nome de processo P a uma expressão observável B. Pode ser utilizada na forma P para caracterizar uma instanciação de um processo dentro de uma expressão comportamental.

Parênteses podem ser usados para evitar ambiguidades e, no caso em que não se deseje utilizá-los, pode-se usar a seguinte ordem de prioridades de avaliação dos construtores: $; > \Box > |[G]| >$ **hide**. Mais ainda, os operadores paralelos devem ser lidos da esquerda para a direita e só pode ser aplicada a associatividade entre eles caso eles tenham os mesmos conjuntos de ações a sincronizar. Mais uma vez por questões de espaço, omitirá-se as regras da semântica operacional dessa linguagem. Caso seja de interesse do leitor, sugere-se consultar (TRETMANS, 2008; GROOTE; RENIERS, 2009).

Como exemplo para a linguagem de processos acima, cita-se *botão* **;** *bebida* **; stop** $\Box$ *botão* a qual corresponde à expressão referente ao LTS da Figura 7. Uma variação da máquina de vendas pode ser P := *botão* ; (*bebida* ; $P \Box i$; P). Nela, pressiona-se o botão, em seguida escolhe-se uma bebida e executa-se novamente P, ou é realizada a ação interna e executa-se novamente P. Por fim, outro exemplo que se cita é **hide** a, b *in* a ; $B_1 \,||\, b$; $B_2 \,\Box\, c$; B_3 ||
Nele, pelas regras de precedência, executa-se (**hide** a, b *in* $(((a$; $B_1) \,||\, ((b$; $B_2) \,\Box\, (c$; $B_3))) \,|||\, (d$;

3.1.2 Lógicas para Especificação de Programas

Um dos primeiros estudos sobre corretude de programas é creditado a Alan Turing, quando, em 1949, propôs o artigo *"Checking a Large Routine"*, o qual continha provas rigorosas de um programa que computa o fatorial através de repetidas adições (FILLIÂTRE, 2011). Todavia, os esforços da comunidade acadêmica nesse sentido se tornaram evidentes na década de 60, quando os três problemas seguintes foram propostos:

- *Especificação de Programas*: "como expressar o que o programa deve fazer?";
- *Desenvolvimento de Programas*: "é possível encontrar um programa que satisfaça a especificação?";
- *Verificação de Programas*: "dado um programa e uma especificação, como garantir que o primeiro satisfaz a segunda?".

Respostas a todos esses questionamentos foram dadas na mesma década e se basearam sempre nas lógicas formais como linguagem para especificação de programas.

Como exemplo dessas lógicas, cita-se a lógica de Floyd-Hoare (FLOYD, 1967; HOARE, 1969) (ver Seção 3.2.2). Inicialmente, Floyd criou o "método das asserções indutivas" para verificar que um programa fluxograma para computar uma função satisfazia a "corretude parcial" (se o programa termina recebendo uma entrada, então a saída satisfaz a especificação). Hoare então transformou o método em um cálculo lógico baseado nas triplas de Hoare (detalhadas mais adiante). É importante ressaltar que a lógica de Floyd-Hoare inicialmente foi proposta para lidar com programas sequenciais, tipicamente interativos. Todavia, Owicki e Gries (OWICKI; GRIES, 1976) estenderam essa lógica para verificar programas paralelos disjuntos, paralelos com variáveis compartilhadas e paralelos com sincronização; e Apt (APT, 1986) a estendeu para verificar programas paralelos com memória distribuída.

Outro exemplo dessas lógicas é a Lógica Modal, que apesar de ter sido esquecida por centenas de anos foi trazida de volta e aplicada para este fim por Engeler (ENGELER, 1967) e diversos outros pesquisadores. A Lógica Modal é adequada para a especificação formal de sistemas reativos e concorrentes, como os *workflows* da HPC Shelf.

3.1.2.1 Lógicas Modais

A Lógica Modal é uma área da Lógica para Ciência da Computação que estuda proposições modais e as relações lógicas entre elas. Os exemplos mais corriqueiros dessas proposições são a possibilidade de um fato e a necessidade de um fato. Como exemplo, têm-se as seguintes proposições:

- É possível que chova amanhã;
- É necessário que chova amanhã ou não chova amanhã;
- Uma proposição p não é possível se e somente se sua negação é necessária.

Os operadores *é possível que* e *é necessário que*, representados por $\Diamond$ (diamante) e $\Box$ (caixa), respectivamente, são chamados de operadores modais, uma vez que especificam a maneira (modalidade) pela qual o resto de uma proposição pode ser considerado verdade. Existem outros diversos tipos de operadores modais que versam sobre tempo, probabilidades, processos, dentre outros. Através desses operadores, a lógica modal busca prover a matemáticos, cientistas da computação e filósofos, símbolos e semânticas necessárias para realizarem provas rigorosas sobre modalidades de interesse (MCCANCE, 1999). A lógica modal remonta à civilização Grega, na *persona* do filósofo Aristóteles, o qual buscava estabelecer um relacionamento lógico sobre *o que é*, *o que é possível* e *o que é necessário*. Por muitos anos, devido a querelas históricas (MCCANCE, 1999), esteve latente. No entanto, nos últimos cinquenta anos, obteve novamente interesse devido a novas categorias de modalidades terem sido propostas.

Certamente, proposições modais não são interpretadas facilmente utilizando tabelas-verdade e variáveis proposicionais tal qual faz o calculo das proposições da lógica clássica. Contudo, é possível definir formalmente uma lógica modal básica como exten-

são da lógica proposicional através da seguinte linguagem (BLACKBURN; BENTHEM; WOLTER, 2006):

- Um conjunto infinito enumerável de letras A, X, $P1$, $P2$, etc., chamadas de variáveis (símbolos) proposicionais;
- As constantes booleanas $\top$ e $\bot$;
- Os operadores unários $\neg$, $\Diamond$ e $\Box$;
- Os operadores binários $\vee$, $\wedge$, $\Longrightarrow$ e $\Longleftrightarrow$;
- Os parênteses (e).

É importante ressaltar que, por simplicidade, nessa linguagem, especifica-se somente uma *modalidade*, ou seja um par de operadores $\Diamond$ e $\Box$. Em outras situações, pode existir um conjunto de modalidades, chamado de MOD, tal que para todo $m \in MOD$ têm-se esses operadores relacionados a ele na forma $\langle m \rangle$ e $[m]$, respectivamente, com significados particulares.

A definição de *Modelo de Kripke* para uma linguagem modal é uma tripla $\mathfrak{M} = (W, \{R^m\}_{m \in MOD}, V)$. W é um conjunto não-vazio chamado de *domínio* e contém elementos chamados de *pontos*, os quais podem representar tempos, situações, estados, mundos e outras coisas. Cada R^m representa uma relação binária em W. V é uma função de avaliação que associa a cada símbolo proposicional p um subconjunto $V(p)$ de W que representa os pontos em que p é verdade. O par $(W, \{R^m\}_{m \in MOD})$ representa o *frame* do modelo $\mathfrak{M}$. Se existe apenas uma modalidade, adota-se R em vez de $\{R^m\}_{m \in MOD}$. Modelos de Kripke podem ser representados através de *Digrafos* (Grafos Direcionados).

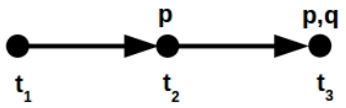
A noção de verdade para as fórmulas compostas pelos operadores oriundos da lógica proposicional na linguagem modal básica é a mesma. No entanto, para os operadores modais, a noção de verdade necessita de uma interpretação mais apurada. A mais comum dessas interpretações é a semântica de múltiplos mundos de Kripke. Sob essa interpretação, a noção de verdade é relativa ao "mundo" em questão. Seja P uma proposição, então $\Box P$ é definida como verdade se é verdade em todos os mundos "acessíveis". Já $\Diamond P$ é verdade em ao menos um mundo "acessível".

A título de exemplificação, considere o modelo temporal de Kripke da Figura 8. Nele, W representa tempos e R é uma relação de tempo que se dá pelo fecho transitivo da relação indicada pelas setas. A fórmula $\Diamond(p \wedge q)$ é verdadeira nos tempos t_1 e t_2, uma vez que neles é possível que p e q sejam verdade em um mundo acessível futuro (tempo em t_3), mas não é valida em t_3. Pelo mesmo raciocínio, $\Diamond p$ é válida em t_1. Mais ainda, para qualquer fórmula modal básica φ, $\Box\varphi$ é verdadeira em t_3, pois nenhum ponto é acessível a partir de t_3 (vacuidade).

3.1.2.2 μ-Calculus Modal

Uma das lógicas modais que obtiveram mais notoriedade nas últimas duas dé-

Figura 8: Um modelo temporal simples



Fonte: Elaborado pelo autor.

cadas foi o μ-Calculus Modal. A partir dele, é possível denotar propriedades sobre sistemas reativos. Sistemas reativos são aqueles cujos elementos geralmente trocam mensagens contendo dados entre eles próprios e com o ambiente e, partir dos valores dessas mensagens, as decisões sobre o desenrolar da computação são tomadas. O μ-calculus modal é importante neste trabalho no sentido de que, através dele, podem-se especificar propriedades de interesse sobre os *workflows* da HPC Shelf.

A lógica modal de *Henessy-Milner* é uma variação da lógica modal básica em que não existem variáveis proposicionais. Além disso, a modalidade diamante $\langle a\rangle\varphi$ é válida quando uma ação a pode ser executada e φ é válida após isso. Por exemplo, a fórmula $\langle a\rangle(\langle b\rangle\top \wedge \langle c\rangle\top)$ afirma que depois da ação a, as ações b e c são possíveis. No caso da modalidade caixa, $[a]\varphi$ expressa que para toda ação a que pode ser executada, φ vale após executar a. Como mais um exemplo, tome a fórmula $[a][b]\bot$. Ela afirma que não existe uma sequência de ações ab.

É interessante que em algumas situações tenha-se mais de uma ação dentro de uma modalidade. Isto se dá pela definição de *fórmulas regulares* dentro de modalidades. A definição de fórmulas regulares depende da definição de *fórmulas de ação*, as quais permitem expressar *conjuntos de ações*, o que é feito mediante a sintaxe a seguir:

$$\alpha ::= a_1|...|a_n \mid \top \mid \bot \mid \overline{\alpha} \mid \alpha \cap \alpha \mid \alpha \cup \alpha.$$

- $a_1|...|a_n$: denota um conjunto unitário representando uma *multiação*, a qual consiste na execução de todas as ações a_1, ..., a_n em paralelo;
- $\top$: representa o conjunto de todas as ações;
- $\bot$: representa o conjunto vazio de ações;
- $-$, $\cap$ e $\cup$: representam os operadores convencionais de conjuntos (complemento, intersecção e união) sobre conjuntos de ações.

Como exemplo disso, considere a fórmula modal $\langle\top\rangle\langle a\rangle\top$, que expressa que depois de uma ação arbitrária uma ação a pode ser executada. Mais ainda, considere $\langle\overline{a}\rangle\langle b\cup c\rangle$. Nela, afirma-se que uma ação diferente de a pode ser executada seguida de b ou c. Por questões de espaço, não serão exemplificados todos os construtores dessa sintaxe e, caso seja interesse do leitor, recomenda-se a consulta de (BLACKBURN; BENTHEM; WOLTER, 2006; LENZI, 2005; GROOTE; RENIERS, 2009).

A definição de fórmula regular estende a noção de fórmula de ação para expressar **sequências de ações** em modalidades e é definida pela seguinte sintaxe:

$$R ::= \epsilon \mid \alpha \mid R \cdot R \mid R + R \mid R^{\star} \mid R^{+}.$$

- ϵ representa a sequência vazia de ações;
- α representa uma fórmula de ação;
- $\cdot$ representa a concatenação de duas sequências de ações;
- $+$ denota a união de sequências;
- * denota a repetição de zero ou mais sequências de ações;
- $^+$ significa repetição de uma ou mais sequências de ações.

Como exemplos de fórmulas regulares, podem-se citar: $\langle a \cdot b \cdot c \rangle \top$, $\langle a^\star \rangle \top$, $[a^+]\varphi$ e $[\top^\star]\langle \top \rangle \top$. A primeira afirma que uma sequência das ações a, b e c pode ser executada. A segunda, por sua vez, diz que qualquer sequência de a's é possível. Já a terceira garante que φ deve ser válida em qualquer estado alcançável a partir de uma sequência não vazia de a's. A última consiste na propriedade de segurança que afirma que não existe *deadlock* em nenhum estado alcançável, ou seja, após de qualquer sequência de estados, é sempre possível ir para um outro estado qualquer.

Por fim, introduzem-se as modalidades de ponto fixo, μ e ν, e a variável X, as quais, junto com lógica modal de Henessy-Milner e as fórmulas regulares, formam o μ-calculus modal. A fórmula $\mu X.\varphi$ representa o ponto fixo minimal e $\nu X.\varphi$ quer dizer o ponto fixo maximal. Considere X como sendo um conjunto de estados. Assim, $\mu X.\varphi$ é válida para todos os estados que pertencem ao menor conjunto X que satisfaz a equação $X = \varphi$. Analogamente, $\nu X.\varphi$ é válida para todos os estados que pertencem ao maior conjunto X que satisfaz a equação $X = \varphi$. X pode aparecer em φ.

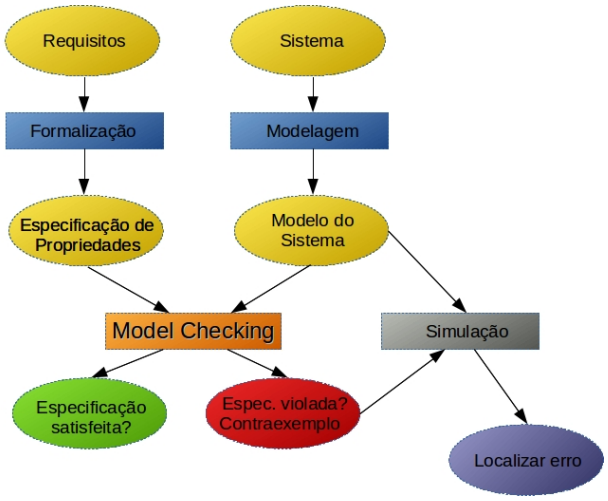
A título de ilustração, considere a fórmula $\mu X.([\top]X \vee \varphi)$. Essa fórmula é válida para todos os estados pertencentes ao menor conjunto X tal que $X = [\top]X \vee \varphi$. Seja X' este menor conjunto. Assim, tem-se $X' = [\top]X' \vee \varphi$. Seja $e_1 \in X'$ um estado em que φ não é válida. Logo, $[\top]X'$ vale em e_1, ou seja, existe uma ação qualquer que o leva a outro estado $e_2 \in X'$. Suponha também que φ não é válida em e_2. Dessa forma, $[\top]X'$ também vale em e_2. Logo existe uma ação qualquer que o leva para outro estado $e_3 \in X'$, e assim sucessivamente. Observe que esse comportamento "recursivo" será repetido até que φ seja válida em um estado de X' (considerando que o sistema não apresente *deadlock*). Dessa maneira, a fórmula $\mu X.([\top]X \vee \varphi)$ afirma que φ se tornará eventualmente válida ao longo de qualquer caminho, considerando que não se tem *deadlock* no sistema.

Para verificar propriedades formais (como as fórmulas do μ-calculus modal) sobre modelos de Kripke, pode-se fazer uso da técnica de *Model Checking* (Verificação de Modelos). Formalmente, consiste em verificar se, dado um modelo (de Kripke) $\mathfrak{M}$, um ponto w de $\mathfrak{M}$ e uma fórmula modal φ, φ é satisfeita por $\mathfrak{M}$ em w.

3.1.3 Técnicas de Verificação: *Model Checking*

De maneira geral, a técnica de *model checking* é uma técnica automática que, dado um modelo finito de um sistema e uma propriedade formal, sistematicamente checa se essa propriedade é válida para (um estado) (n)aquele modelo (BAIER; KATOEN, 2008). Essa

Figura 9: Abordagem *Model Checking*



Fonte: Elaborado pelo autor.

técnica faz uso da exploração de todos os estados possíveis de um sistema, utilizando "força bruta", e está dividida nas seguintes fases:

- Modelagem: deve-se modelar o sistema utilizando a descrição da linguagem fornecida pela ferramenta de verificação de modelos disponível. Então, deve-se formalizar as propriedades a serem checadas utilizando a linguagem de especificação de propriedades da ferramenta;
- Execução: executa-se o verificador de modelos para checar a validade das propriedades no modelo do sistema;
- Análise: para cada propriedade checada, verifica-se se ela foi provada ou violada. No segundo caso, deve-se analisar o contraexemplo fornecido pela simulação, refinar o modelo ou a propriedade, e repetir todo o procedimento;
- Estouro de Memória: caso ocorra, deve-se tentar reduzir o modelo e tentar novamente.

Esse esquema pode ser melhor compreendido através da Figura 9. O modelo do sistema é geralmente gerado automaticamente a partir de uma descrição do modelo que é especificada em alguma linguagem de programação como C, Java, alguma linguagem específica de domínio, como a SAFeSWL, ou até mesmo alguma linguagem de descrição de *hardware*, como Verilog ou VHDL. A especificação de propriedades diz *o que* o sistema deve fazer, enquanto a descrição do modelo diz *como* o sistema deve se comportar.

A ferramenta de verificação de modelos, então, examina todos os estados relevantes possíveis do sistema e, caso encontre um contraexemplo em um estado, indica como o modelo pode chegar a esse estado. O contraexemplo representa um caminho que parte do estado inicial até o estado que viola a propriedade. Daí, com a ajuda de um simulador, o usuário pode repetir o caminho até esse estado indesejável, fazendo uso de ferramentas de depuração. Ao encontrar o problema, que pode ser na propriedade ou no modelo, é possível corrigi-lo e a verificação pode ser realizada novamente.

A técnica de *model checking* é uma das técnicas de verificação cujas aplicações a sistemas de computação obtiveram mais sucesso. Teve início com o trabalho de Clarke e Emerson em 1980 (EMERSON; CLARKE, 1980), o qual utilizou o formalismo da lógica temporal para provar propriedades sobre programas paralelos. Atualmente, é bastante aplicada em sistemas de aviação, protocolos de comércio eletrônico e em diversos padrões de comunicação do IEEE. Possui diversos pontos fortes, dos quais se podem elencar sua possibilidade de aplicação aos mais diversos tipos de sistemas de *software* e de *hardware*; a possibilidade de verificação de propriedades isoladas, em detrimento de outras; o fato de não exigir alto grau de conhecimento do usuário para aplicar a técnica; e um pano de fundo matemático correto, o qual recai sobre uma gama de algoritmos em grafos, estruturas de dados avançadas e lógica.

Dentre os pontos fracos da técnica de *model checking*, pode-se citar que é uma técnica voltada a aplicações de *controle intensivo* de processos, não sendo aconselhada a aplicações com alto volume de dados (*data intensive*). Outro ponto fraco é que sua aplicação é condicionada a questões de *decidibilidade*, ou seja, para sistemas com um número potencialmente infinito de estados ou que raciocina sobre tipos abstratos de dados definidos sobre lógicas não decidíveis, a *model checking* em geral não é efetiva. O último ponto fraco que se cita, e o mais importante deles, é o *problema da explosão de estados*. Ocorre quando o número de estados necessários para modelar o sistema excede a quantidade de memória disponível no computador. Embora seja comum, existem diversas formas para tentar se lidar com esse problema (BAIER; KATOEN, 2008).

Para realizar *model checking* de uma propriedade μ-calculus modal sobre um programa mCRL2, o mCRL2 primeiro converte o programa mCRL2 para uma especificação de processo linear (LPS - *Linear Process Specification*)[29] (GROOTE; RENIERS, 2009), a qual consiste em uma especificação com somente um processo, sem paralelismo, comunicação ou operadores de visibilidade. Em seguida, o mCRL2 realiza uma conversão da fórmula da propriedade junto com a especificação LPS do programa para um sistema de equações chamado PBES - *Parametrized Boolean Equation System* (Sistema de Equações Booleanas Parametrizadas) (GROOTE; RENIERS, 2009). Esse sistema é adequado para aplicar *model checking*, pois existem inúmeros algoritmos que o resolvem em tempo aceitável, a despeito da complexidade exponencial.

3.2 Verificação Formal em Componentes de Computação

Nesta seção, serão apresentadas a lógica de especificação e as técnicas de verificação estudadas com a finalidade de serem aplicadas a programas paralelos dos componentes de computação da HPC Shelf. Entretanto, antes disso, é interessante que sejam categorizadas as diferentes classes de programas paralelos que os componentes podem conter e quais

[29] <http://www.mcrl2.org/release/user_manual/language_reference/lps.html>

técnicas estão aptas a verificar propriedades formais sobre eles.

3.2.1 Programas Paralelos e Verificação Formal de *Software*

Como dito no Capítulo 2, cada componente de computação da HPC Shelf possui um conjunto de *unidades*, as quais representam processos que executam em nós de processamento distintos da plataforma de computação que abrigará o componente. Comumente, os programas que executam nas unidades de componentes de computação são regidos pelo padrão *SPMD* (*Single Program Multiple Data*), ou seja, um mesmo código é executado em diferentes unidades, trabalhando em porções diferentes dos dados e trocando mensagens com outras unidades. Esse é o padrão de programação da biblioteca MPI (FORUM, 2009; Message Passing Interface Forum, 1994). Pode haver um grau de paralelismo a mais quando uma unidade possui mais de um núcleo de processamento, permitindo assim que o código MPI que roda na unidade possa disparar *threads* nesses núcleos. Esse tipo de paralelismo é comumente alcançado pela combinação das bibliotecas MPI e OpenMP (OPENMP, 1997).

É ainda possível que cada unidade do componente execute um programa paralelo diferente, e que esses possam trocar mensagens entre si. Para tanto, basta que o desenvolvedor do componente forneça os diversos programas e associe a cada um deles a unidade em que ele irá executar.

Um pouco menos comum em programas ou algoritmos ditos embaraçosamente paralelos, é o caso em que determinadas unidades executam programas sequenciais (iguais ou não) paralelamente, ou seja, os programas são disparados paralelamente mas não trocam mensagens entre si. Por fim, é ainda possível que cada unidade seja um programa paralelo exclusivamente com memória compartilhada, ou seja, cada um é disparado paralelamente e então gerencia um conjunto de *threads* no modelo *fork-join*, não trocando mensagens com as outras unidades.

Dentre essas classes de programas paralelos, o interesse maior de verificação neste trabalho recai sobre os programas MPI, uma vez que ele é um padrão *de facto* em aplicações de computação de alto desempenho que fazem uso de plataformas de computação paralela de memória distribuída, o alvo de componentes paralelos do modelo Hash. Com relação às propriedades, o interesse maior deste trabalho recai naquelas ditas funcionais, uma vez que permitem verificar se, de fato, o programa faz o que se propõe. No tocante às ferramentas de verificação, está-se interessado nas que possuem tempo de verificação não proporcional ao número de unidades paralelas engajadas na computação. As ferramentas de verificação dedutiva (ver Seção 3.2.3), diferentemente das ferramentas de *model checking* (ver Seção 3.2.4), possuem tempos de verificação independentes do números de unidades (LÓPEZ et al., 2015).

Apesar de existir uma técnica distinta baseada na Lógica de Hoare (apresentada na Seção 3.2.2) que permite verificar propriedades funcionais sobre os programas de

cada uma das classes anteriores (APT; BOER; OLDEROG, 2009), a grande maioria das ferramentas de verificação dedutiva, as quais implementam a Lógica de Hoare, é capaz de verificar propriedades funcionais somente sobre programas sequenciais. No tocante a programas paralelos com memória compartilhada, observou-se o emprego de ferramentas de verificação dedutiva de programas apenas na verificação de propriedades de segurança. Com relação a programas paralelos com memória distribuída, a única ferramenta de verificação dedutiva encontrada foi o ParTypes (LÓPEZ et al., 2015)[30], o qual permite apenas confrontar funcionalmente o programa com um protocolo de alto nível que reflete suas operações de comunicação.

3.2.2 Lógicas para Especificação de Programas: Lógica de Floyd-Hoare

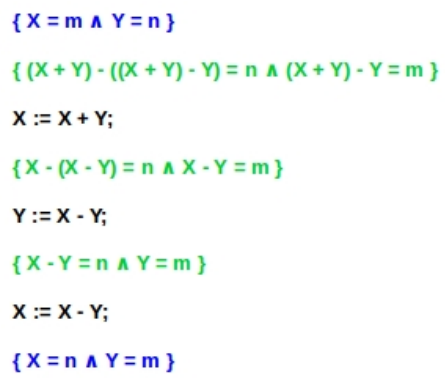
A *Lógica de Floyd-Hoare*, ou simplesmente *Lógica de Hoare*, é um formalismo que permite raciocinar sobre o comportamento de programas imperativos. Nela, cada construtor sintático (comando) da linguagem de programação é equipado com uma regra de prova, que permite inferir fatos sobre a corretude de programas que contém aquele construtor (PIERCE et al., 2014). Foi criada nos anos 60 e aperfeiçoada ao longo dos anos, fazendo parte, hoje, em uma infinidade de ferramentas que especificam e verificam sistemas de *software* reais na academia e na indústria. Suas principais vantagens recaem sobre uma maneira natural de se escrever especificações de programas através de *asserções* e sobre sua *composicionalidade*, no sentido de que a organização do programa reflete a organização da prova de que ele está de acordo com a sua especificação.

Essa lógica se baseia na definição de *asserções*, que são fórmulas da lógica de predicados que versam sobre os valores das variáveis (estado) do programa em determinados momentos da sua execução. Além das asserções, utiliza a noção de *Tripla de Hoare*. Uma Tripla de Hoare é uma tripla da forma $\{Pre\}\ C_i\ \{Post\}$, onde C_i é um comando ou um bloco de comandos, Pre é uma asserção chamada de pré-condição e $Post$ é uma asserção chamada de pós-condição. É interpretada da seguinte forma: supondo que Pre é válida antes de executar C_i, então, se C_i terminar, $Post$ será válida.

Usando-se triplas de Hoare, pode-se "decorar" um programa, ou seja, encontrar, para cada comando do programa, uma pré- e uma pós-condição adequadas. Dado um programa P devidamente decorado, pode-se tentar provar que ele está de acordo com uma especificação, a qual consiste de uma pré-condição $EsPre_i$ e uma pós-condição $EsPost_i$ para cada método M_i do programa. A união do conjunto de todas as pré-condições $EsPre_i$, $EsPre$, com o conjunto de todas as pós-condições $EsPost_i$, $EsPost$, resulta no chamado *conjunto das asserções de especificação* do programa. Assim, verificar se P está de acordo com sua especificação consiste em fornecer uma decoração adequada a cada método M_i de P de forma que cada tripla $\{EsPre_i\}M_i\{EsPost_i\}$ seja válida. A

[30] <http://gloss.di.fc.ul.pt/tryit/tools/ParTypes>

Figura 10: Um programa decorado



Fonte: Elaborado pelo autor.

decoração de um programa é feita utilizando um conjunto de axiomas e regras de inferência que podem ser vistos em (PIERCE et al., 2014; APT; BOER; OLDEROG, 2009). Para se aplicar as regras de inferência, algumas implicações lógicas devem ser provadas. Tais implicações são chamadas de *Condições de Verificação* (*VCs*). Inicialmente, Hoare propôs essas regras para uma linguagem imperativa simples. Outros pesquisadores posteriormente desenvolveram axiomas e regras de inferência para o raciocínio sobre programas concorrentes e ponteiros, os quais podem ser encontrados em (APT; BOER; OLDEROG, 2009; REYNOLDS, 2002; KASSIOS, 2011).

A Figura 10 apresenta um exemplo simples de programa decorado. Nele, é feita somente a troca dos valores das variáveis X e Y. Sua especificação corresponde às asserções em azul. Quando se tem uma asserção seguida de outra, quer-se dizer que a primeira implica logicamente a segunda (*regra da consequência*).

O algoritmo *Weakest Precondition Algorithm* (WP) (Pré-condição Mais Fraca) é uma forma sistemática para decorar um programa (MARCHÉ; TAFAT, 2012). Em vez de se decorar todo o programa, são fornecidas apenas a especificação de corretude do programa (conjuntos $EsPre$ e $EsPost$) e outras asserções "chave" como invariantes de laço, dentre outras. Para decorar um programa P, o WP, dentre outras coisas, calcula a pré-condição mais fraca para cada comando C_i baseada em uma pós-condição $Post$, ou seja, ele encontra uma asserção Pre tal que $\{Pre\}\ C_i\ \{Post\}$ e, para toda asserção Pre' tal que $\{Pre'\}\ C_i\ \{Post\}$, $Pre' \implies Pre$. Existem diversas ferramentas que calculam pré-condições mais fracas, dentre as quais podem ser citadas os *plugins* WP[31] e Jessie[32] para a ferramenta de verificação Frama-C, a qual será vista mais adiante.

Caso o desenvolvedor de um componente de computação deseje que seja verificado nele um conjunto de propriedades funcionais e de segurança utilizando a lógica de Hoare, ele deve fornecer, para certos métodos de programas, pré- e pós-condições de especificação ($EsPre$ e $EsPost$) que reflitam as propriedades desejadas. Para tais programas,

[31] <http://frama-c.com/wp.html>
[32] <http://frama-c.com/jessie.html>

devem ser fornecidas ainda asserções que provem que, supondo que as pré-condições de especificação são válidas, as pós-condições de especificação são válidas. Este processo será melhor detalhado no Capítulo 5.

3.2.2.1 Lógica de Separação

Existe uma extensão da Lógica de Hoare hábil a tratar de estruturas de dados mutáveis, referenciadas através de ponteiros. Essa lógica é chamada de *Lógica de Separação* e através dela é possível estabelecer e provar especificações que exigem, dentre outras coisas, que programas não façam acessos a posições de memória não alocadas.

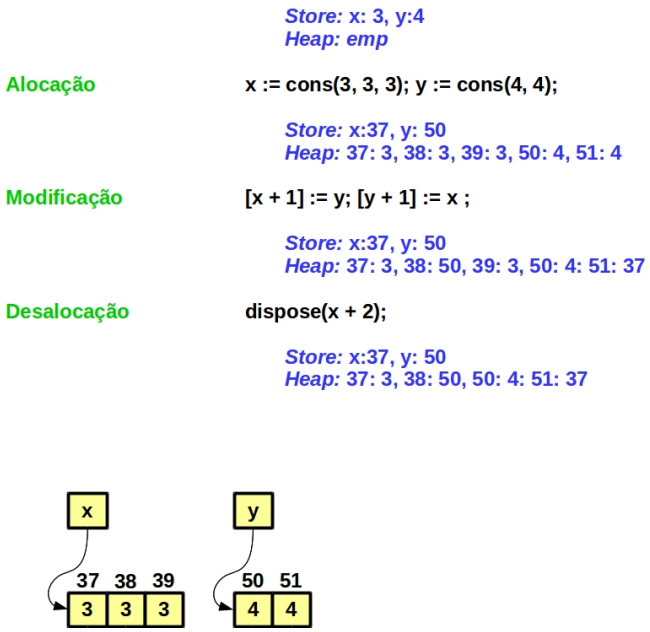
A *Lógica de Separação* foi desenvolvida por John C. Reynolds, Peter O'Hearn, e outros no final dos anos 90 (REYNOLDS, 2002). É baseada na *conjunção de separação* $P * Q$, que afirma que P e Q valem para para porções separadas de memória, e em outras regras de prova sobre programas que exploram a separação para prover um raciocínio modular sobre programas. Descreve programas com comandos para alocar, acessar, modificar e desalocar estruturas de dados, sem fazer o uso de *garbage collection*. Uma característica interessante dessa linguagem é que qualquer tentativa de derreferenciar um ponteiro quebrado faz com que o programa aborte. Tem sido largamente usada para raciocinar sobre situações envolvendo recursos compartilhados, como, por exemplo, concorrência com memória compartilhada.

Suas asserções descrevem estados que são compostos por uma *store*, que corresponde a estados de variáveis locais (alocadas na pilha), e por um *heap*, que corresponde a estados de objetos alocados dinamicamente. Uma *store* é uma função que associa variáveis do programa a valores e um *heap* é uma função parcial que associa endereços de memórias ativos a valores. Além disso, introduz algumas notações especiais, que merecem ser destacadas:

- $[x]$ denota o conteúdo do endereço de memória x (derreferenciação);
- $x \mapsto e$ significa que na posição de memória x está armazenado o valor e;
- $x \mapsto -$ quer dizer que x foi alocado, mas não especifica o valor apontado por x;
- *emp* representa o *heap* vazio;
- $P * Q$ significa que o *heap* contém partes disjuntas tais que P vale em uma e Q vale em outra (Conjunção de Separação);
- $P -\!\!* Q$ significa que se o *heap* for estendido com uma parte disjunta tal que P vale, então Q vale no novo *heap* (Implicação de Separação).

Considere o exemplo da Figura 11. Nele, são executados cinco comandos na linguagem da lógica de separação e são mostrados a *store* e o *heap* a cada momento. No primeiro comando, a operação **cons** aumenta o domínio do *heap* com três endereços consecutivos que não estão no domínio. Similarmente, no segundo comando, o domínio é aumentado com dois endereços consecutivos. O comando **dispose** desaloca uma célula de memória. Já na Figura 12, é representada a memória desse programa graficamente

Figura 11: Um exemplo de programa na Lógica de Separação



Fonte: Elaborado pelo autor.

de acordo com cada comando executado. Ao final desse programa, a asserção $(x \mapsto 3) * (x + 1 \mapsto y) * (y \mapsto 4) * (y + 1 \mapsto x)$ é válida.

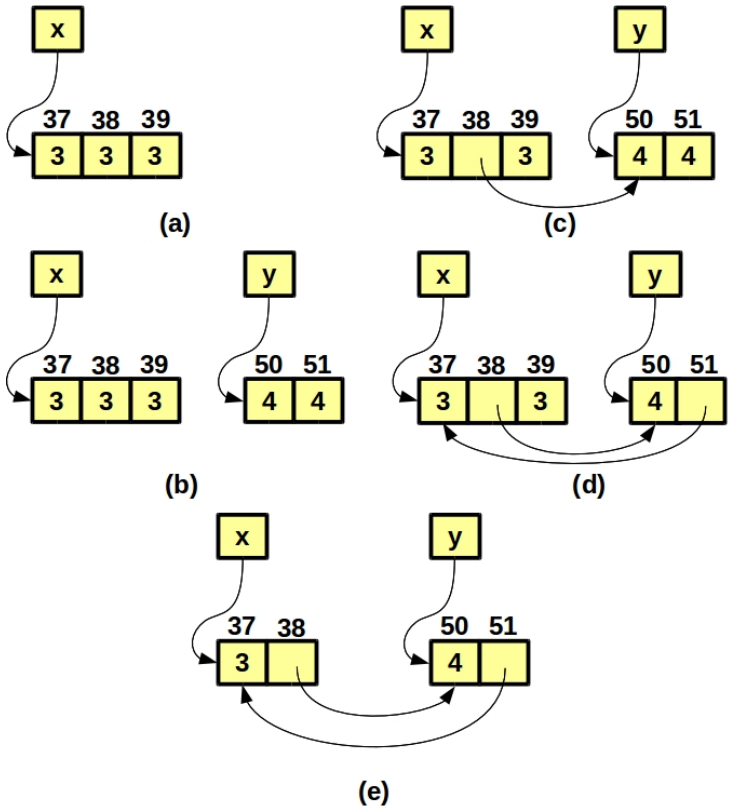
Novamente, para decorar programas com asserções na lógica de separação, é necessário utilizar um conjunto de axiomas e regras de inferência que podem ser encontradas em (REYNOLDS, 2002; O'HEARN, 2012; REYNOLDS, 2008).

Existem algumas ferramentas que implementam a lógica de separação, dentre as quais se cita a biblioteca Ynot[33] (NANEVSKI et al., 2008; CHLIPALA et al., 2009) para o provador de teoremas Coq, que será abordado mais adiante. Ynot axiomatiza uma mônada parametrizada para computações imperativas (tal qual faz Haskell para a mônada IO), a fim de tornar a escrita de programas mais fácil. Além da lógica de separação e esse estilo de escrita imperativo dentro da linguagem funcional do Coq (Galina), se vale de todo o poder oriundo do Coq, que é um dos provadores de teoremas mais expressivos.

Outro exemplo de implementação da lógica de separação é a ferramenta de verificação VeriFast[34] (JACOBS et al., 2011; JACOBS; PIESSENS, 2008; JACOBS; SMANS; PIESSENS, 2010b; JACOBS; SMANS; PIESSENS, 2010a). O VeriFast visa a verificação de programas *multithread* escritos em C e Java. Incorpora a essas linguagens de propósito geral uma rica especificação através da definição de tipos de dados, funções recursivas primitivas puras, predicados abstratos na lógica de separação e funções lemas, as quais servem como provas de que suas pré-condições implicam suas pós-condições. É capaz de enviar as condições de verificação geradas para serem provadas nos provadores automáticos Redux e Z3, os quais serão abordados em breve. O VeriFast é utilizado na arquitetura

[33] <http://ynot.cs.harvard.edu/>
[34] <http://people.cs.kuleuven.be/~bart.jacobs/verifast/>

Figura 12: Representação da memória do programa da Figura 11



Fonte: Elaborado pelo autor.

proposta por este trabalho para compor componentes táticos, a fim de se verificarem propriedades de segurança em componentes de computação, quando fizerem uso da lógica de separação.

Uma abordagem mais geral em relação à técnica de *model checking* para automatizar a verificação de *software* é a *Verificação Dedutiva de Programas*. Em vez de explorar o espaço dos estados de um programa, o que faz com que se tenha que usar abstrações finitas sobre os estados, essa técnica tenta verificar programas através de provas executadas em provadores de teoremas.

3.2.3 Técnicas de Verificação: Verificação Dedutiva de Programas

A *Verificação Dedutiva de Programas* consiste em gerar, a partir do sistema e sua especificação, uma coleção de obrigações de provas matemáticas, cuja verdade implica na conformidade do sistema com sua especificação, e descarregar essas obrigações usando provadores de teoremas interativos ou automáticos. Essa abordagem tem a desvantagem de normalmente requerer que o utilizador compreenda em pormenores o porquê do correto funcionamento do sistema e transmita essas informações ao sistema de verificação, quer sob a forma de uma sequência de teoremas a provar, quer sob a forma de especificações de componentes do sistema (por exemplo, funções ou procedimentos) e talvez subcomponentes (como laços ou estruturas de dados). Apesar de ser empregada para diversos fins,

é muito comum o seu uso para automatizar a abordagem axiomática para verificação de programas da lógica de Hoare e suas derivadas.

Dentre os provadores interativos, podem-se citar os provadores da lógica de alta ordem, como Coq (The Coq development team, 2004), Isabelle/HOL (NIPKOW; WENZEL; PAULSON, 2002) e PVS (OWRE; RUSHBY; SHANKAR, 1992). Para aplicar esses provadores na verificação de programas, a semântica do programa e o sistema de prova devem ser embutidos na lógica de alta ordem do provador. Tendo em vista minimizar a necessidade de interação do usuário com o provador, um conjunto de táticas pode ser formalizado para automatizar a aplicação de regras de prova e procedimentos de decisão podem ser aplicados para checar automaticamente implicações lógicas necessárias nas premissas das regras de prova.

Além de provadores de teoremas interativos, diversas outras ferramentas foram propostas a fim de auxiliar o processo de verificação dedutiva de programas. A seguir, são tratadas todas as ferramentas de verificação dedutiva que foram utilizadas neste trabalho, as quais podem ser utilizadas em composição para formar componentes táticos. O uso dessas ferramentas em conjunto exige, em geral, um fluxo que se inicia na aplicação de um *Frontend de Verificação*, depois a aplicação de uma *Linguagem de Verificação Intermediária* e, por fim, a aplicação de um provador de teoremas, que pode ser automático ou interativo. Dessa forma, componentes táticos que utilizem tais ferramentas são, na maioria das vezes, uma composição de um *frontend* de verificação, uma linguagem de verificação e um provador. Em certos casos, podem ser compostos somente por um *frontend* de verificação e um provador, ou somente por uma linguagem de verificação intermediária e um provador.

3.2.3.1 *Frontends* de Verificação

Em geral, essas ferramentas permitem a verificação de programas anotados com asserções e escritos em uma linguagem de alto nível, tais como C, Java ou C#. O código deve ser anotado com asserções consistindo de pré-condições, pós-condições, invariantes de laço, invariantes de objetos e outros elementos (ALMEIDA et al., 2014). Tais ferramentas são geralmente hábeis a efetuar a tradução dos códigos escritos na linguagem de alto nível para uma linguagem de verificação intermediária, a qual gerará condições de verificação (VCs) e as descarregará em algum provador. Exemplos delas são Frama-C (CUOQ et al., 2012; KIRCHNER et al., 2015) e VCC (COHEN et al., 2009), para código C, Krakatoa[35] (FILLIÂTRE; MARCHÉ, 2007; MARCHÉ; MOHRING; URBAIN, 2004), para código Java, e Dafny[36] (LEINO, 2010), que é uma ferramenta de verificação moderna para uma linguagem similar à linguagem C#. Em outros casos, essas ferramentas são por si só capazes de fazer chamadas diretas a provadores. Esse é o caso da ferramenta de lógica de

[35] <http://krakatoa.lri.fr/>
[36] <http://research.microsoft.com/en-us/projects/dafny/>

separação VeriFast, a qual pode descarregar VCs diretamente para os provadores Redux e Z3.

Frontends de verificação podem incorporar novas características de verificação quando utilizam *plugins* de verificação. Esse é o caso do Frama-C, ao qual podem ser incorporados os *plugins* Jessie e WP, em que cada um oferece um conjunto distinto de construtores que podem ser utilizados dentro das asserções. Outro caso é o VCC que pode ser incorporado do *plugin* ParTypes, o qual permite confrontar um programa C/MPI anotado na sintaxe do VCC com um protocolo de alto nível fornecido que reflete suas operações de comunicação (ponto a ponto e coletivas).

3.2.3.2 Linguagens de Verificação Intermediárias

Essas ferramentas atuam como camadas sobre as quais é possível construir verificadores para outras linguagens. Oferecem linguagens de especificação e programação ricas para as quais são gerados códigos a partir do código original do programa a ser verificado. A partir do código gerado na linguagem intermediária, é possível gerar obrigações de provas (VCs) para diversos provadores, que podem ser automáticos ou interativos. Por exemplo, a partir de programa C (decorado ou não), pode ser gerado um programa em uma linguagem intermediária, e ela pode gerar VCs que visam provar a corretude do programa com relação a um certo aspecto e repassá-las a um provador específico. Observe que isso evita que tenha que ser gerado um código diferente para cada provador que se deseja utilizar.

Why3[37] (BOBOT; FILLIÂTRE; MARCHÉ; PASKEVICH, 2011) e Boogie[38] (BARNETT et al., 2006; LEINO, 2008) são exemplos de ferramentas baseadas em linguagem de verificação intermediária. Why3 oferece muitas características de alto nível oriundas da linguagem ML, tais como tipos algébricos e polimórficos, definições recursivas, predicados (co-)indutivos, casamento de padrões, expressões *let*, dentre outras. Why3 é utilizada como linguagem intermediária para código C anotado, com o *frontend* Frama-C, e para código Java anotado, com o *frontend* Krakatoa, dentre outros. Why3 permite que sejam descarregadas obrigações de provas em muitos provadores automáticos e interativos, tais como Alt-Ergo, CVC3, Z3, CVC4, E, Simplify, SPASS, Vampire, Yices, Coq, Isabelle/HOL, PVS e outros (FILLIÂTRE; PASKEVICH, 2013).

Boogie, por sua vez, possui construtores matemáticos, como construtores de tipos, constantes, funções, axiomas, mapas, quantificadores, vetores de bits, etc.; e construtores imperativos, os quais introduzem variáveis globais, declarações de procedimentos, e implementações de procedimentos (LEINO, 2008). Verificadores construídos sobre Boogie devem compilar para o formato CIL (BARNETT et al., 2006), o formato executável da máquina virtual .NET. Boogie, então, traduz o código CIL para o código da sua lin-

[37] <http://why3.lri.fr/>
[38] <http://research.microsoft.com/en-us/projects/boogie/>

guagem interna e, então, descarrega as VCs usando o provador automático Z3. Boogie é a linguagem intermediária utilizada pelo Dafny.

3.2.3.3 Provadores Automáticos

Existe uma grande variedade de provadores automáticos. A partir deles, podem-se extrair duas classes principais: *Resolvedores SMT* (*Satisfiability Modulo Theories*) e *Provadores ATP* (*Automated Theorem Provers*). Resolvedores SMT determinam quando fórmulas na lógica de primeira ordem, nas quais alguns símbolos de funções e predicados têm interpretações adicionais, são satisfatíveis[39]. Provadores ATP lidam com a tarefa de provar que uma conjectura é uma consequência lógica de um conjunto de axiomas e hipóteses[40]. Serão utilizados neste trabalho os resolvedores SMT Alt-Ergo, CVC3, CVC4 e Z3; e os provadores ATP E, SPASS e Vampire. Será utilizado também um provador especializado na verificação de fórmulas numéricas incluindo ponto-flutuante, chamado Gappa. Este conjunto é o subconjunto dos provadores disponíveis em Why3 que revela os melhores resultados na prática (BOBOT; FILLIÂTRE; MARCHÉ; PASKEVICH, 2014; AYAD; MARCHÉ, 2010).

3.2.3.4 Provadores Interativos

Tais ferramentas são também chamadas de assistentes de prova e auxiliam o desenvolvimento de prova formais através da colaboração humano-máquina. São equipados com uma *interface* de edição que facilita a busca por procedimentos de provas armazenados e a aplicação de táticas. Uma tática é um procedimento que embute um raciocínio lógico, que pode ser simples ou elaborado, o qual serve para resolver, total ou parcialmente, objetivos de prova. Como exemplo, cita-se a tática omega, do Coq, que implementa um procedimento automático de decisão para a aritmética de Presburger. Ela resolve fórmulas livres de quantificadores construídas com negações, conjunções, disjunções e implicações em igualdades e desigualdades envolvendo números naturais e inteiros. Coq, Isabelle/HOL e PVS são suportados por Why3. De acordo com (CHLIPALA, 2011; WIEDIJK, 2003), Coq e Isabelle/HOL são superiores a PVS.

Coq é um provador de teoremas de alta ordem baseado no *Cálculo das Construções* (*Co-*)*Indutivas*, um λ-Calculus tipado superior, no qual definições de objetos matemáticos podem ser dadas e provas sobre proposições os envolvendo podem ser executadas. A lógica do Coq é uma lógica construtiva de alta ordem que se baseia no isomorfismo de Curry-Howard (SØRENSEN; URZYCZYN, 2006). Esse isomorfismo preza que uma especificação S pode ser vista tanto como um teorema, de um ponto de vista matemático, ou como um tipo definido em uma linguagem de programação, de um ponto de vista computacional. Similarmente, um programa C que implementa o tipo S pode ser visto

[39] <http://en.wikipedia.org/wiki/Satisfiability_modulo_theories>
[40] <http://www.cs.miami.edu/~tptp/OverviewOfATP.html>

matematicamente como uma prova P do teorema S. A checagem da prova P é, portanto, a checagem de tipos de C.

Isabelle/HOL é uma especialização do assistente de prova genérico Isabelle para uma lógica de alta ordem. Isabelle/HOL e Coq compartilham algumas caraterísticas, mas divergem, dentre outras coisas, no sentido de que a lógica do Isabelle/HOL é clássica e a lógica do Coq não é. Além disso, Coq possui tipos dependentes e Isabelle/HOL, não (WIEDIJK, 2003). Esses dois provadores de teoremas pertencem ao *estado-da-arte* em Métodos Formais e são utilizados em muitas áreas importantes como criptografia, compiladores certificados, *kernels* certificados e compiladores certificadores (AFFELDT; NOWAK; YAMADA, 2012; BLECH; GRÉGOIRE, 2011; KLEIN, 2014; BARTHE; GRÉGOIRE; KUNZ; REZK, 2009; KLEIN et al., 2009; LEROY, 2006).

3.2.4 Técnicas de Verificação: *Model Checking*

Devido à escassez de ferramentas de verificação dedutiva para programas paralelos com memória distribuída, mais especificamente programas MPI, foram investigados *model checkers* que podem ser utilizados para esse fim. Os *model checkers* mais relevantes encontrados foram o ISP (*In-situ Partial Order*) (VO et al., 2009; VAKKALANKA et al., 2008) e o CIVL (*Concurrency Intermediate Verification Language*) (SIEGEL et al., 2015)[41]. Ambos basicamente se propõem a verificar um conjunto padrão (fixo), porém expressivo, de propriedades de segurança. Nesse sentido, a que mais se destaca, tanto pelo menor tempo de verificação quanto pelo portfólio de aplicações já verificadas é o ISP.

O ISP é capaz de verificar propriedades sobre programas MPI, escritos em C, C++, C# ou Java, podendo portar diretivas OpenMP. O ISP verifica *deadlocks*, violações de asserções, falhas de acesso a objetos MPI e condições de corrida em comunicações (casamentos de comunicações inesperados), dentre outras. Para tanto, utiliza o algoritmo POE (*Partial Order avoiding Elusive interleavings*) (VAKKALANKA et al., 2008), o qual explora somente os entrelaçamentos relevantes do programa utilizando uma técnica *model checking* conhecida como *redução de ordem parcial* (CLARKE JR.; GRUMBERG; PELED, 1999). Uma importante característica do ISP é seu tempo baixo de verificação, o que é alcançado, dentre outras coisas, pelo fato de necessitar saber, no momento da verificação, a quantidade de nós que será empregada na computação, o que diminui razoavelmente o espaço de possibilidades. Como ponto fraco, no entanto, pode ser citada a necessidade do ISP de reiniciar o ambiente MPI a cada verificação.

O CIVL é a evolução do TASS (SIEGEL; ZIRKEL, 2011), outro *model checker* bem conceituado. CIVL é na verdade um arcabouço que compreende:

- Uma linguagem de programação intermediária própria (CIVL-C), composta por um conjunto de *primitivas básicas*, voltadas, sobretudo, à implementação de paralelismo

[41] <http://vsl.cis.udel.edu/civl/>

(memória distribuída e compartilhada);

- Um conjunto de tradutores de programas C que portam diretivas das bibliotecas MPI, OpenMP, PThreads e CUDA para CIVL-C;
- Um conjunto de *primitivas de refinamento* que podem ser anotadas em programas C que portam diretivas de algumas das bibliotecas anteriores, para fins de verificação. Essas primitivas não se baseiam no raciocínio da Lógica de Floyd-Hoare, e dizem respeito somente a que procedimentos de refinamento disponíveis na ferramenta devem ser aplicados automaticamente ao programa por ela até que ela encontre um programa implementado somente por primitivas básicas CIVL-C, que então está pronto para ser verificado (HAWBLITZEL; PETRANK; QADEER; TASIRAN, 2015). A verificação de segurança do programa concorrente, entretanto, é feita através do raciocínio de segurança de Owicki e Gries (OWICKI; GRIES, 1976) que se baseia na *verificação sequencial* de cada componente do programa e na verificação de *não-interferência* entre as componentes do programa;
- Um *model checker* que utiliza *execução simbólica* para verificar um conjunto fixo de propriedades de segurança em programas CIVL-C, tendo a habilidade de enviar condições de verificação para os provadores CVC3, CVC4 e Z3. Possui ainda a capacidade de verificar se dois programas CIVL-C são equivalentes.

Tanto o ISP quanto o CIVL foram adotados pela arquitetura descrita nesta Tese.

3.3 Considerações Finais

Este capítulo apresentou as semânticas de programas, lógicas para especificação de programas e técnicas de verificação formal de *software* utilizadas nesta Tese para composição de componentes táticos para os componentes SWC2 e C4, apresentados nos próximos capítulos. As técnicas de verificação formal estudadas foram a *model checking* e a verificação dedutiva de programas.

A primeira é utilizada neste trabalho para a verificação de propriedades em componentes da espécie *workflow*. As propriedades a serem verificadas são de continuidade e de segurança e são especificadas utilizando uma lógica modal chamada de μ-calculus modal. Os *workflows* em si são especificados formalmente utilizando sistemas de transições etiquetadas. É empregada também na verificação de componentes da espécie *computação*, quando fazem uso de funções da biblioteca MPI. Nesse caso, o conjunto de propriedades de segurança que podem ser verificadas é o conjunto suportado pelas ferramentas de verificação ISP e CIVL. É ainda possível verificar a equivalência funcional entre dois programas MPI com o CIVL.

A segunda foi empregada somente na verificação de propriedades em programas anotados com asserções da lógica de Hoare ou da lógica de separação de componentes da

espécie computação. As propriedades a serem verificadas sobre os programas são dos tipos funcionais e de segurança. Elas podem ser descritas diretamente nos programas, através de asserções especiais de especificação (*EsPre* e *EsPost*) para seus métodos, ou fornecidas diretamente pelo usuário. O programa, junto com as propriedades, é enviado ao *frontend* de verificação, o qual cria um programa anotado para uma linguagem de verificação intermediária. Essa, por sua vez, é capaz de gerar um conjunto de condições de verificação (VCs), as quais visam garantir que o programa que recebeu atende às asserções que contém. Essas VCs podem ser repassadas a inúmeros provadores, aumentando, assim, a chance de se provar mais VCs. As VCs então podem ser provadas sem a interação humana, através de provadores automáticos, ou com a colaboração do programador, no caso de provadores interativos. Por fim, ressalta-se que a única ferramenta que permite a verificação dedutiva de programas MPI é o ParTypes, o qual permite verificar se o programa está de acordo com um protocolo de comunicação fornecido pelo usuário.

4 Componente Certificador de *Workflow* Científico (SWC2)

Este capítulo tem por finalidade descrever a arquitetura de um componente certificador para orquestrações de componentes em sistemas de computação paralela descritos em SAFeSWL, associadas ao componente especial, presente nesses sistemas, chamado `workflow`. Esse componente certificador será chamado de SWC2 (DANTAS; CARVALHO JUNIOR; BARBOSA, 2017b), um acrônimo em inglês para *Componente Certificador de Workflow Científico*, e será particularmente voltado à verificação de propriedades de continuidade e de segurança. Seguindo o arcabouço de certificação da HPC Shelf proposto na Seção 2.6, no código SAFeSWL, uma instância de um componente SWC2 deve ser conectada ao componente `workflow` através de uma ligação de certificação. Deve-se observar que o processo de certificação através de componentes SWC2 fica a cargo do provedor de aplicações, uma vez que o código SAFeSWL é criado por ele no momento de construção da aplicação.

Os objetivos específicos da proposta da arquitetura de componente certificador para *workflows* científicos, chamado SWC2, são:

- Proporcionar a criação de componentes certificadores, por parte dos usuários certificadores de componentes, que estendam SWC2, utilizando a abstração de contratos contextuais para expressar as propriedades que esses componentes verificam, de forma a possibilitar a existência de diversos certificadores de *workflows*, que possam se distinguir de acordo com as propriedades que verificam;
- Permitir que usuários provedores de aplicações possam verificar as orquestrações que compõem em relação a um conjunto de propriedades de continuidade e de segurança, aumentando assim os níveis de confiança nos sistemas de computação paralela por eles criados;
- Fazer com que, para um mesmo domínio de problemas para usuários especialistas, caso existam diversas aplicações que possam ser utilizadas para especificar problemas que os interessam, o critério "certificação" possa influenciar na escolha entre eles de qual aplicação poderá ser utilizada;
- Suportar a ligação de mais de um certificador ao componente workflow de um sistema de computação paralela, permitindo assim a criação de diversos sistemas de certificação paralela, cada um fazendo uso das funcionalidades de verificação de um certificador específico;
- Permitir que o provedor de aplicações, caso possua conhecimento sobre a ferramenta mCRL2, bem como sobre propriedades do μ-calculus modal, possa fornecer propriedades *ad hoc* (que não fazem parte do conjunto inicial proposto por essa arquitetura) a serem verificadas por um componente certificador de *workflow*.

Na Seção 4.1, apresentam-se as principais características dos *workflows* científicos descritos na literatura, através de um mapeamento sistemático, dentre as quais

destacam-se aquelas que puderam ser verificadas por certificadores. Já na Seção 4.2, discutem-se os detalhes particulares sobre os *componentes táticos* utilizados pelo componente SWC2. Na Seção 4.3, é apresentada a definição do contrato contextual para o certificador SWC2. Na Seção 4.4, apresenta-se o *algoritmo de tradução de SAFeSWL para mCRL2* (S2m), o qual é executado pelo certificador para a tradução de *scripts* de *workflows* escritos no subconjunto de orquestração de SAFeSWL para códigos compreendidos pela ferramenta de verificação mCRL2. Finalmente, na Seção 4.5, apresentam-se as especificações formais de um conjunto inicial de propriedades de continuidade e de segurança, descritas utilizando o μ-calculus modal, previamente disponibilizadas por essa arquitetura.

4.1 Caracterização de *Workflows* Científicos

Sistemas de Gerenciamento de *Workflows* Científicos (SGWfC) têm sido amplamente aplicados por cientistas e engenheiros para o projeto, execução e monitoramento de *pipelines* reutilizáveis de tarefas de processamento de dados em descobertas científicas e análises de dados (TAYLOR; DEELMAN; GANNON; SHIELDS, 2006). A partir de um estudo sistemático sobre os SGWfC mais difundidos (Askalon (QIN; FAHRINGER; PLLANA, 2006), BPEL Sedna (WASSERMANN et al., 2007), Kepler (LUDÄSCHER et al., 2006), Pegasus (DEELMAN et al., 2005), Taverna (WOLSTENCROFT et al., 2013) e Triana (HARRISON; TAYLOR; WANG; SHIELDS, 2008)), bem como o próprio SAFe (SILVA; CARVALHO JUNIOR, 2016), conseguiu-se extrair características relevantes, no contexto deste trabalho, dos *workflows* construídos nessas ferramentas, tendo em vista verificá-los através de componentes SWC2 na HPC Shelf:

1. **Componentes de grossa granularidade**: *workflows* científicos diferem dos *workflows* computacionais convencionais no sentido de que seus componentes são tipicamente de grossa granularidade, devido à natureza dos algoritmos especializados que executam. Tais algoritmos exigem cálculos intensivos, possivelmente fazendo uso de técnicas e infraestruturas de CAD;
2. **Poucos construtores de orquestração**: componentes de grossa granularidade encapsulam a maior parte da complexidade computacional dos *workflows* científicos. Assim, as linguagens de orquestração destinadas à criação desses *workflows* geralmente oferecem poucos construtores, como sequenciamento, ramificação, iteração, paralelismo e assincronismo;
3. ***Workflows* abstratos**: *workflows* científicos são comumente representados por componentes e relações de dependência de execução entre eles, sem levar em consideração as plataformas de computação nas quais os componentes serão executados. Todavia, antes da execução de um componente no *workflow*, um *procedimento de resolução de plataforma* pode ser aplicado para descobrir qual plataforma de computação melhor se adequa ao contexto de implantação do componente. Com relação

a esse item, pode ser verificado estaticamente no *workflow*, por exemplo, se as ações computacionais dos componentes são sempre ativadas após suas plataformas de computação já terem sido resolvidas;

4. **Separação de *interface* e implementação dos componentes**: *workflows* científicos usam geralmente uma descrição abstrata dos componentes, ou seja, somente *interfaces* expondo as operações disponíveis, sem associar o componente a uma implementação específica dessas operações. Em um momento apropriado da execução do *workflow*, é então acionado um *procedimento de resolução de componente*, que, a partir de uma representação abstrata do componente e um contexto que reflete as características da execução atual do componente, descobre no catálogo um componente específico que melhor implementa o contexto fornecido. Em termos de verificação, pode-se verificar, por exemplo, se as ativações de ações computacionais de componentes durante a execução do *workflow* são feitas depois que os componentes já foram resolvidos;
5. **Instanciação incremental dos recursos computacionais**: em geral, as plataformas de computação que alojam os componentes orquestrados são instanciadas apenas quando os componentes são de fato úteis para a computação realizada pelo *workflow* e são liberadas quando os componentes não são mais necessários. Esse padrão envolve basicamente três operações: a *implantação* do componente na plataforma de computação de destino, que faz com que a implementação do componente e possíveis bibliotecas necessárias sejam transferidas remotamente do catálogo central de componentes; a operação de *instanciação* da plataforma, que consiste em alocar os recursos computacionais necessários ao componente e configurar o ambiente de execução para ele; e, finalmente, a operação de *liberação* do componente da plataforma de computação, desalocando os recursos atribuídos a ele. No que diz respeito à verificação, a consistência da ordem de ativação destas operações pode ser verificada estaticamente a partir do *script* do *workflow*;
6. ***Workflows* internos dos componentes**: além do *workflow* da aplicação, que ativa externamente ações de componentes, cada componente pode ter um *workflow* interno que, a partir de ativações externas de ações do componente pelo *workflow* da aplicação, ativa ou habilita/desabilita ações do componente. A *composição* do *workflow* da aplicação com os *workflows* internos dos componentes pode permitir a verificação de propriedades mais realistas no sistema computacional;
7. **Clusterização**: a orquestração de componentes de fina granularidade é considerada custosa em *workflows* científicos. Em geral, o tempo necessário para tornar o componente pronto (resoluções de plataformas e componentes, implantação e instanciação) excede seus tempos efetivos de computação. Assim, componentes de fina granularidade com características semelhantes podem ser agrupados em um único componente de grossa granularidade, chamado de componente *cluster*. Dessa forma,

a ativação de uma ação de um componente *cluster* é, na verdade, a ativação de um *workflow* que ativa ações dos componentes de fina granularidade associados. Esse comportamento pode ser incorporado na modelagem de verificação do *workflow* para gerar uma especificação mais precisa do sistema computacional;

8. ***Timeouts***: pode haver um atraso entre o tempo em que uma ação de um componente é ativada no *workflow* e o tempo em que a ação realmente começa no componente. Assim, sistemas de *timeouts* são comumente usados em *workflows* para definir o tempo máximo que o *workflow* deve esperar até que a ação do componente realmente seja iniciada. Quando o tempo limite for atingido, o *workflow* pode ativar a ação novamente ou reportar uma exceção para a aplicação. Uma modelagem temporizada ou estocástica, que leve em consideração a composição do *workflow* da aplicação com os *workflows* internos dos componentes, pode permitir a verificação de violações de *timeouts*.

As verificações indicadas nos itens **iii**, **iv** e **v** referem-se à consistência de ativações de ações da porta padrão do ciclo de vida dos componentes (*LifeCycle*) e já são suportadas por padrão pelo certificador de *workflows* abstrato descrito na Seção 4.3. Mais especificamente, as ações desta porta devem ser ativadas na seguinte ordem estrita: **resolve**, **deploy**, **instantiate** e **release**. Além disso, as ações computacionais de qualquer porta de um dado componente devem ocorrer entre ativações de **instantiate** e **release**.

Na HPC Shelf, cada componente computação ou conector C tem um *workflow* interno que habilita/desabilita suas ações computacionais. Cada *workflow de componente* consiste em um conjunto de regras da forma:

$$W_C ::= act_1 \rightarrow act_2\downarrow \mid \top \rightarrow act_2\downarrow \mid act_1 \rightarrow act_2\nmid \qquad \text{(regra de componente)}$$

Act_C é o das ações de um componente C, incluindo todas as ações do ciclo de vida, e $act_1,\ act_2 \in Act_C$. A regra $\top \rightarrow act_2\downarrow$ significa que act_2 está sempre habilitada, enquanto $act_1 \rightarrow act_2\downarrow$ indica que, quando act_1 for concluída, act_1 se torna desabilitada e act_2 se torna habilitada. Por sua vez, a regra $act_1 \rightarrow act_2\nmid$ indica que, depois de act_1 ser concluída, act_1 e act_2 se tornam desabilitadas. O certificador de *workflows* concreto descrito na Seção 4.3 é capaz de compor o *workflow* da aplicação com os *workflows* dos componentes para fins de verificação (do item **vi**). Um exemplo de *workflow* de aplicação, *workflows* internos de habilitação/desabilitação de ações de componentes e a tradução da composição desses *workflows* feita pelo referido certificador são apresentados na descrição completa das regras de tradução (Apêndice A.1). Outros exemplos de *workflows* internos de componentes são apresentados nos estudos de caso relatados no Capítulo 6.

Serão chamados de componentes *cluster* na HPC Shelf aqueles componentes que possuem mais de uma unidade. Neste caso, pode-se dizer que cada unidade de um

componente *cluster* representa um componente de fina granularidade, e a ativação de uma ação do componente *cluster* corresponde, de fato, a ativações em paralelo das ações correspondentes em todos os componentes de fina granularidade relacionados. Assim, se o provedor de aplicações deseja realizar a verificação descrita no item **vii** (clusterização), ele, antes de solicitar a certificação do *workflow* ao SAFe, deve pedir ao SAFe que resolva particularmente os componentes *cluster*, calculando, assim, plataformas virtuais para eles, as quais determinam seus números de unidades. Assim, no momento da tradução, os componentes *clusters* serão substituídos pelos seus componentes de fina granularidade pelo referido certificador concreto. Um exemplo de componente *cluster* e sua tradução são apresentados nos estudos de caso.

A verificação no item 8 (*timeouts*) não foi possível por limitações da ferramenta mCRL2, portanto ainda não é suportada pelo certificador concreto definido neste capítulo. No entanto, experimentos com Uppaal (BEHRMANN et al., 2006), para modelagem temporizada, e cadeias de Markov interativas (HERMANNS, 2002), para modelagem estocástica, tendo em vista definir novos componentes táticos, fazem parte de planos para trabalhos futuros.

4.2 Componentes Táticos de Certificadores SWC2

Atualmente, certificadores de *workflows* utilizam somente o componente tático que encapsula o *toolset* de verificação mCRL2. Dessa forma, podem-se usar somente dois tipos de paralelismo. O primeiro consiste em lançar, em cada unidade de processamento da plataforma virtual que abriga o componente tático, uma unidade paralela do componente tático mCRL2, dividindo igualmente entre elas o conjunto total de propriedades gerado. O segundo consiste em cada unidade do componente tático disparar em paralelo *threads* nos núcleos de processamento disponíveis na respectiva unidade de processamento da plataforma virtual, dividindo entre elas o conjunto de propriedades atribuído a cada unidade do componente tático. Todavia, quando novos componentes táticos, que permitam verificar outras classes de propriedades, forem incorporados à arquitetura, como por exemplo Uppaal, eles poderão ser lançados paralelamente com o mCRL2, sendo regidos por um código de orquestração na linguagem TCOL.

A seguir, descrevem-se os contratos contextuais dos certificadores SWC2.

4.3 Contratos Contextuais

Como quaisquer componentes da HPC Shelf, certificadores de *workflows* também são subdivididos em componentes abstratos e componentes concretos. Como descrito no Capítulo 2, componentes abstratos podem ser representados através de *assinaturas contextuais*, através dos quais componentes concretos que os implementam podem ser diferenciados entre si pelo mecanismo de resolução, ou seleção, de componentes.

Em geral, certificadores de *workflows* se diferenciam pelo conjunto de propriedades que verificam, as quais podem ser de continuidade ou de segurança. Existem três tipos de propriedades que certificadores de *workflows* podem verificar:

- propriedade *padrão*;
- propriedade *de aplicação*;
- propriedade *ad hoc*.

Uma propriedade *padrão* é comum a quaisquer *workflows* SAFeSWL, tais como ausência de *deadlock*, ausência de laços infinitos, verificação da consistência de ativações de ações da porta do ciclo de vida dos componentes, etc. Essas propriedades já são suportadas por padrão pela arquitetura dos certificadores e fazem parte da assinatura contextual do certificador abstrato (descrita mais adiante). Dessa forma, caso um certificador concreto deseje utilizá-las, basta valorá-las em seu contrato contextual.

O certificador de componentes pode estender o certificador de *workflow* abstrato para adicionar propriedades relevantes a uma dada aplicação ou uma classe delas. Tais propriedades são chamadas de propriedades *da aplicação* e também podem ser representadas como parâmetros de contexto. Entretanto, um provedor de aplicações que deseje verificá-las em seu *workflow* deve se referir diretamente ao componente abstrato específico que as suporta.

Propriedades *ad hoc*, por sua vez, podem ser especificadas por usuários provedores de aplicações e repassadas juntamente com o XML que descreve o código do *workflow* em SAFeSWL. Nem todo certificador aceita propriedades desse tipo. Por isso, é necessário também um parâmetro de contexto na assinatura contextual que reflita esta opção. Um dos estudo de caso do Capítulo 6 traz exemplos dessas propriedades.

Como se sabe, propriedades podem ser também divididas entre *obrigatórias* ou *opcionais*. No caso de propriedades padrão ou da aplicação, o componente certificador determina quais propriedades que ele verifica são obrigatórias e quais são opcionais. No caso de propriedades *ad hoc*, o provedor de aplicações é quem faz essa distinção. Novamente, um componente certificador considera um componente como certificado quando ao menos todas as propriedades obrigatórias forem provadas sobre ele.

Cada certificador mantém o tempo médio de prova de cada propriedade que verifica, e essa informação fica disponível na descrição do componente no Core. Dessa forma, o provedor de aplicações pode avaliar o custo/benefício de verificar uma determinada propriedade. Mais ainda, o tempo máximo de verificação (incluindo todas as propriedades) pode ser expresso como um parâmetro de contexto de *qualidade* na assinatura contextual do certificador, permitindo que o provedor de aplicações possa guiar o sistema de resolução de contratos contextuais na escolha de um componente cujo tempo máximo de verificação não ultrapasse um valor determinado por ele. No momento da criação do componente certificador, o certificador de componentes atribui um valor experimental para esse parâmetro de contexto e, a partir de então, ele é atualizado ao final

de cada verificação. O certificador de componentes pode ainda estender o certificador de *workflow* abstrato para incorporar novos parâmetros de qualidade, devendo, entretanto, implementar a lógica de programação da métrica associada a cada um.

Por fim, observa-se que um componente certificador pode lançar diferentes unidades de um componente tático associado a ele paralelamente nas unidades de processamento da plataforma virtual que abriga o componente tático, reduzindo, potencialmente, o tempo de verificação final das propriedades no componente a ser certificado.

A seguir, apresenta-se a assinatura contextual do certificador de *workflow*:

SWC2 [***deadlock_absence*** = D : DATYPE,
infinite_loop_absence = I : ILATYPE,
life_cycle_verification = L : LCVTYPE,
ad_hoc_properties = A : AHTYPE,
max_verification_time = M : INTEGER]

Os parâmetros de contexto ***deadlock_absence*** (ausência de *deadlock*), ***infinite_loop_absence*** (ausência de *loop* infinito) e ***life_cycle_verification*** (verificação da consistência de ativações de ações da porta do ciclo de vida dos componentes do *workflow*) são parâmetros de contexto que dizem respeito a propriedades de continuidade e de segurança padrão que o certificador pode verificar e serão melhor discutidas nas seções que se seguem. Para o parâmetro ***life_cycle_verification***, existe até o momento uma única valoração, LIFECICLEVERIFICATIONMODELCHECKER, que realiza uma verificação de consistência no uso das ações do ciclo de vida dos componentes através de *model checking* de propriedades do μ-calculus modal.

O parâmetro de contexto ***ad_hoc_properties*** afirma se o componente certificador aceita que, juntamente com código de orquestração do *workflow*, sejam repassadas a ele propriedades formais criadas, com qualquer propósito que seja, pelo provedor de aplicações. Exemplos dessas propriedades serão mostrados no Capítulo 6.

max_verification_time representa o tempo máximo (em segundos) de verificação que o provedor de aplicações está disposto a aceitar. Inicialmente, o certificador de componentes o calcula experimentalmente para um determinado perfil de plataforma virtual, e a cada verificação ele é atualizado. No caso de um certificador sequencial, esse tempo é obtido pela soma dos tempos médios de verificação de cada propriedade. No caso de um certificador paralelo, é o maior tempo médio de verificação.

Um exemplo de contrato contextual para a assinatura contextual anterior pode ser descrito como se segue:

SWC2Impl : SWC2 [***deadlock_absence*** = DEADLOCKABSENCE,
infinite_loop_absence = INFINITELOOPABSENCE,
life_cicle_verification = LIFECICLEVERIFICATIONMODELCHECKER,
ad_hoc_properties = ADHOCPROPERTIES,
max_verification_time = 20]

A partir desse contrato, pode-se afirmar que o componente concreto SWC2Impl verifica ausência de *deadlocks*, verifica existência de *loops* infinitos, requer que o ciclo de vida dos componentes seja verificado por *model checking*, aceita propriedades *ad hoc* e o tempo máximo de verificação inicial é 20s. A prova é realizada paralelamente, ou seja, diversas unidades do componente tático associado (MCRL2, descrito adiante) são lançadas nas unidades de processamento da plataforma virtual que o abriga e são divididas igualmente entre elas o conjunto de propriedades formais de responsabilidade do componente tático. Por fim, observa-se que as propriedades padrão em SWC2Impl foram configuradas como obrigatórias.

Um componente tático abstrato pode ser descrito como se segue:

TACTICAL [***message_passing_interface*** = M : MPITYPE,
number_of_nodes = N : INTEGER,
number_of_cores = C : INTEGER]

O parâmetro de contexto ***message_passing_interface*** refere-se à característica da plataforma de execução que delimita qual é a *interface* de passagem de mensagens utilizada pelo componente tático concreto. Desse modo, a plataforma virtual a ser escolhida pelo Core para executar o componente tático deve possuir, previamente instalada, a *interface* especificada pelo argumento atribuído a ***message_passing_interface***. A *interface* padrão mais utilizada é o MPI (*Message Passing Interface*), a qual possui várias distribuições, com características peculiares que podem ser aproveitadas na implementação de um determinado componente. Esse parâmetro permite escolher distribuições alternativas de MPI, ou deixar a escolha em aberto quando a implementação for portável. Por sua vez, os parâmetros ***number_of_nodes*** e ***number_of_cores*** delimitam, respectivamente, a quantidade de unidades de processamento distribuídos e de núcleos de processamento presentes em cada unidade da plataforma virtual escolhida.

Um componente tático abstrato que representa implementações de componentes táticos que encapsulam o *toolset* mCRL2 pode ser criado estendendo o componente tático abstrato TACTICAL para adicionar o parâmetro ***version***, o qual informa qual a é a versão do mCRL2 utilizada no componente tático:

MCRL2 [***version*** = V : VERSIONTYPE,
message_passing_interface = M : MPITYPE,
number_of_nodes = N : INTEGER,
number_of_cores = C : INTEGER] extends TACTICAL[***message_passing_interface*** = M,
number_of_nodes = N,
number_of_cores = C]

O certificador de componentes que criou o certificador concreto SWC2Impl o associou ao componente tático abstrato MCRL2 e forneceu a orquestração descrita na

Figura 13: Código de orquestração TCOL do componente certificador SWC2Impl

```
<sequence>
      <invoke action="resolve" id_port="life-cycle-mCRL2" />
      <invoke action="deploy" id_port="life-cycle-mCRL2" />
      <invoke action="instantiate" id_port="life-cycle-mCRL2" />
      <invoke action="verify_perform"  id_port="verify-mCRL2" />
      <invoke action="release" id_port="life-cycle-mCRL2" />
</sequence>
```

Fonte: Elaborado pelo autor.

Figura 13.

O certificador de componentes forneceu também a seguinte valoração (contexto) à assinatura MCRL2, a qual visa escolher um componente tático MCRL2 concreto implementado em MPI:

$$\text{MCRL2}\ [\boldsymbol{message\_passing\_interface} = \text{MPI}]$$

Um contrato contextual para a assinatura MCRL2 é o seguinte:

$$\begin{aligned}\text{mCRL2Impl} : \text{MCRL2}\ [&\boldsymbol{version} = 201409.1,\\ &\boldsymbol{message\_passing\_interface} = \text{MPICH2},\\ &\boldsymbol{number\_of\_cores} = 4]\end{aligned}$$

Através desse contrato contextual, pode-se dizer que o componente concreto mCRL2Impl foi implementado utilizando a biblioteca MPI, mais especificamente utilizando a distribuição MPICH2[42], requer que cada unidade de processamento da plataforma virtual alvo possua no mínimo 4 núcleos de processamento e que a versão do *toolset* mCRL2 utilizada foi a 201409.1. Veja que, através do parâmetro ***number_of_cores***, nos núcleos de processamento de cada unidade de processamento da plataforma virtual alvo serão lançadas *threads* e dividido igualmente entre elas o conjunto de propriedades formais de responsabilidade da unidade do componente tático que executa na unidade de processamento.

Observe que, de acordo com o contexto para MCRL2 fornecido no certificador concreto SWC2Impl, o componente tático mCRL2Impl pode figurar como candidato, considerando que o valor MPI é supertipo de MPICH2 (MPICH2 :< MPI).

O provedor de aplicações pode submeter ao Core uma valoração como a seguinte:

$$\begin{aligned}\text{SWC2}\ [&\boldsymbol{deadlock\_absence} = \text{DeadlockAbsence},\\ &\boldsymbol{ad\_hoc\_properties} = \text{AdHocProperties},\\ &\boldsymbol{max\_verification\_time} = 30]\end{aligned}$$

Essa valoração diz que o provedor de aplicações busca um certificador que faça verificação de ausência de *deadlocks*, aceite propriedades *ad hoc* e que tenha tempo médio de verificação de no máximo 30s. Claramente, o componente certificador concreto SWC2Impl é um candidato para essa valoração.

[42] <http://www.mpich.org/>

Figura 14: Gramática formal para o subconjunto de orquestração de SAFeSWL

$$T ::= L \mid G \mid T_1; T_2 \mid T_1 || T_2 \mid \texttt{repeat}\ T \qquad \text{(tarefa)}$$
$$L ::= act \mid \texttt{break} \mid \texttt{continue} \mid \texttt{start}(h, act) \mid \texttt{wait}(h) \mid \texttt{cancel}(h) \qquad \text{(literal)}$$
$$G ::= act{\downarrow}T \ \mid \ act{\downarrow}T + G \qquad \text{(tarefas guardadas)}$$

Fonte: Elaborado pelo autor.

Para que se possa realizar *model checking* das propriedades formais em componentes *workflow*, o componente certificador realiza a tradução do código de orquestração do *workflow*, escrito em SAFeSWL, para um sistema LTS escrito na linguagem mCRL2.

4.4 Tradução de SAFeSWL para mCRL2 (S2m)

Ao conjunto de regras de tradução de um código de orquestração SAFeSWL para um código mCRL2, dá-se o nome de *algoritmo de tradução de SAFeSWL para mCRL2*, ou S2m. Aqui, será feita apenas uma apresentação do esquema geral de tradução e da semântica operacional definida neste trabalho para o subconjunto de orquestração de SAFeSWL, a partir da qual foram criadas as referidas regras. O leitor, entretanto, pode obter no Apêndice A.1 a descrição formal completa das regras de tradução que definem o algoritmo.

4.4.1 Semântica Operacional do Subconjunto de Orquestração de SAFeSWL

A semântica operacional em questão é melhor visualizada a partir de uma nova gramática para o subconjunto de orquestração de SAFeSWL, representada na Figura 14, aqui chamada *gramática formal* para o subconjunto de orquestração de SAFeSWL.

Seja W uma referência para o componente workflow do sistema de computação paralela. Nessa gramática, c varia sobre identificadores de componentes, h (*handle*) varia sobre os naturais, e $act \in Act_W$. Assume-se um número mínimo de ações, que inclui as ações usadas por SAFeSWL para gerenciar o ciclo de vida dos componentes: $\{\texttt{resolve}(c), \texttt{deploy}(c), \texttt{instantiate}(c), \texttt{release}(c)\} \subseteq Act_W$.

A semântica de um *workflow* SAFeSWL W consistindo de uma tarefa T_W é dada pelo conjunto de regras definido na Figura 15, partindo do estado $\langle T_W, \texttt{stop}, \emptyset, \emptyset, \emptyset, \emptyset \rangle$. Aqui, stop é um símbolo dedicado utilizado pelas regras de execução, denotando a conclusão de uma tarefa. Cada estado de execução consiste de uma tupla $\langle T_1, T_2, E, L, S, F \rangle$, em que:

- T_1 é a próxima tarefa a ser evoluída;
- T_2 é a tarefa seguinte a ser evoluída;
- E é o conjunto das ações atualmente habilitadas nos componentes;
- L é uma pilha de pares contendo o começo e o fim de cada bloco de repetição que

envolve a tarefa corrente;

- S é um conjunto de pares contendo ações disparadas assincronamente que ainda não foram concluídas e os manipuladores (*handles*) associados a elas;
- F é um conjunto de manipuladores (*handles*) associados a ações assíncronas que já terminaram de executar.

Observe que, por simplicidade, as regras apresentadas ainda não levam em consideração o comportamento imposto pelos *workflows* internos dos componentes, que habilitam/desabilitam suas ações e manipulam diretamente o conjunto E. Para capturar esse comportamento, serão apresentadas adiante alterações necessárias em algumas regras.

Eis uma breve descrição das regras de execução. A regra (`big-step`) estabelece uma relação de transição em passo largo entre estados de execução. A regra (`action`) afirma que um estado que contém a ativação uma ação SAFeSWL habilitada faz com que o sistema observe a ação e vá para o estado em que se avalia a tarefa seguinte. A regra (`sequence`) indica que a sequência de duas tarefas leva à avaliação em sequência dessas tarefas. A regra (`parallel-left`) diz que se um estado X com uma tarefa T_1 leva a um estado Y qualquer em um número qualquer de passos, a paralelização de T_1 com uma tarefa T_2, partindo do estado X, leva ao estado Y, contudo propagando o paralelismo para a tarefa seguinte. A regra (`stop-par-left`) possibilita a terminação do paralelismo. A regra (`select-left`) indica que a ação ativada deve estar habilitada no componente. A regra (`select-right`) estabelece que uma ação desabilitada no componente não pode executar. A regra (`repeat`) executa uma tarefa T_1, recordando na pilha L as tarefas de início e fim da iteração. As regras (`continue`) e (`break`) executam, respectivamente, a tarefa de início e fim do laço mais interno. A regra (`start`) diz que uma ação SAFeSWL habilitada e um *handle* ainda não usado podem ser associados e adicionados ao conjunto S, emitindo também uma ação para o sistema ($start(a, h)$). A regra (`finish`) indica que uma ação ativada assincronamente pode de fato ocorrer, tendo seu *handle* registrado em F e emitindo uma ação para o sistema ((a,h)). A regra (`wait`) afirma que a espera por uma ação assincronamente ativada que já foi concluída não tem efeito para o sistema. Por fim, a regra (`cancel`) diz que uma ação ativada assincronamente pendente pode ser cancelada.

4.4.1.1 *Workflows* Internos dos Componentes

A semântica de um *workflow* é adaptada para levar em conta as regras impostas pelos componentes, que adicionarão e removerão elementos do conjunto de ações habilitadas (E). Assim, um sistema em execução consiste não apenas em uma tarefa T_W que descreve o *workflow* da aplicação, mas também em um conjunto fixo de regras de componente R, oferecidas pelos componentes envolvidos. Dado um *workflow* T_W e um conjunto de regras R, a semântica de T_W sob R é dada pelas regras anteriores (Figura 15), substituindo as regras (`action`) e (`finish`) por aquelas na Figura 16. A função *update* é definida como:

Figura 15: Semântica operacional do subconjunto de orquestração de SAFeSWL ((`parallel-right`) e (`stop-par-right`) são simétricas, respectivamente, a (`parallel-left`) e (`stop-par-left`), e omitidas aqui)

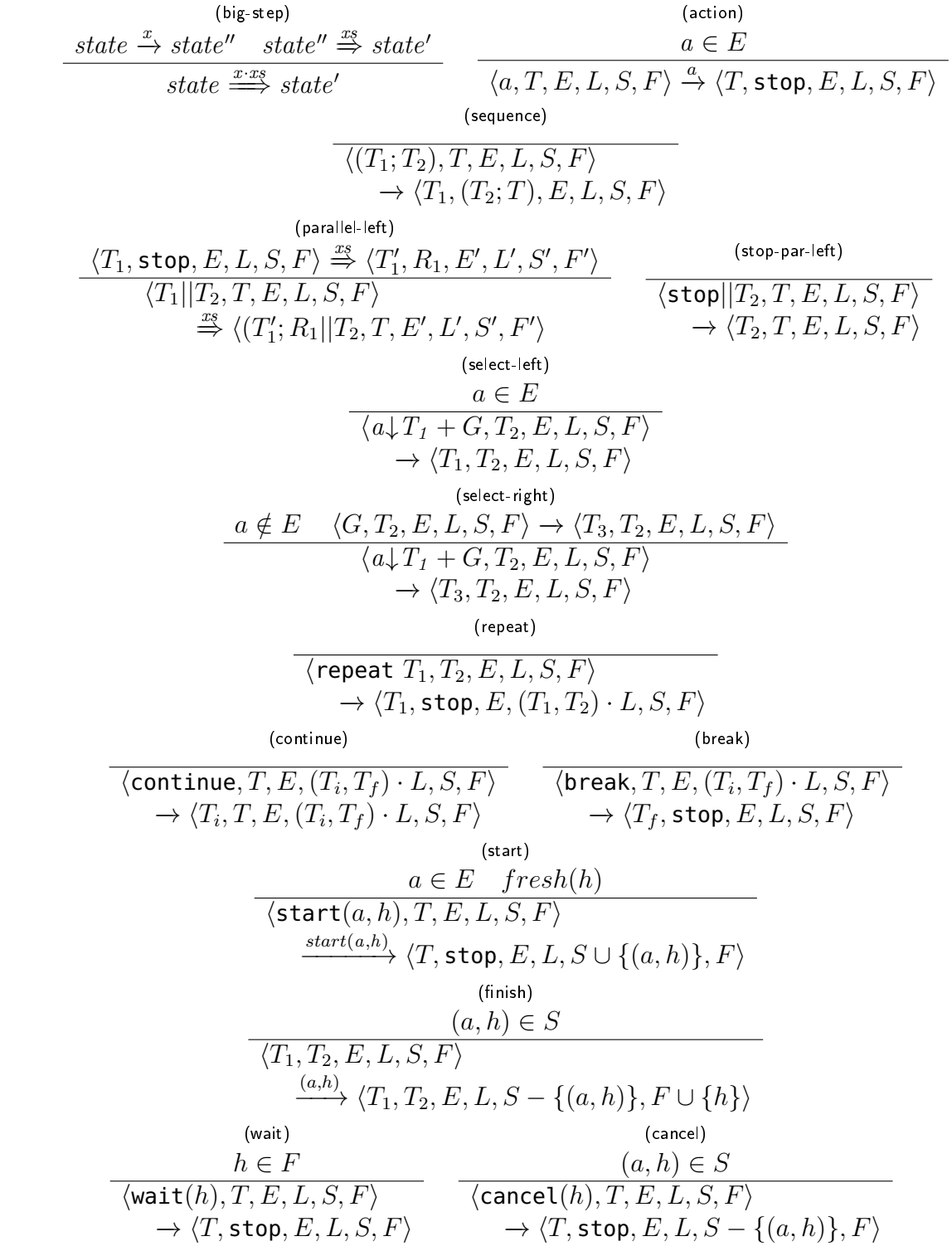


Figura 16: Regras da semântica operacional de SAFeSWL atualizadas

$$\frac{a \in E}{\langle a, T, E, L, S, F\rangle \xrightarrow{a} \langle T, \texttt{stop}, \mathit{update}(E, a), L, S, F\rangle}\ (\text{action})$$

$$\frac{(a, h) \in S}{\langle T_1, T_2, E, L, S, F\rangle \xrightarrow{(a,h)} \langle T_1, T_2, \mathit{update}(E, a), L, S - \{(a, h)\}, F \cup \{h\}\rangle}\ (\text{finish})$$

Fonte: Elaborado pelo autor.

$$\mathit{update}(E, a) = (E \cup \{a' \mid a \to a'\downarrow \in R\}) - (\{a\} \cup \{a' \mid a \to a'\not\downarrow \in R\})$$

4.4.2 Esquema Geral de Tradução

Apresenta-se aqui, sucintamente, o esquema geral de tradução feito pelo algoritmo S2m a partir das regras da semântica operacional do subconjunto de orquestração de SAFeSWL. Novamente, salienta-se que a descrição formal completa das regras de tradução pode ser consultada no Apêndice A.1.

A regra (`action`) afirma que cada ação SAFeSWL é uma ação mCRL2 observável. A regra (`sequence`) afirma que uma sequência de duas tarefas em SAFeSWL é traduzida pela composição sequencial das traduções correspondentes. As regras (`parallel-left`) e (`parallel-right`) indicam que a tradução de um conjunto de tarefas paralelas ocorre pela criação de processos mCRL2 em um paradigma *fork-join*. As regras (`select-left`) e (`select-right`) indicam a necessidade de criação de processos mCRL2 que controlam o estado (habilitado/desabilitado) das ações. As regras (`repeat`), (`continue`) e (`break`) indicam a necessidade de criação de um processo mCRL2 *gerenciador* de uma tarefa de iteração para detectar a necessidade de uma nova iteração, o retorno ao início da iteração, ou o fim da iteração. A regra (`start`) indica a necessidade de criação de um processo mCRL2 assíncrono que irá eventualmente executar a ação. Além disso, também é necessário criar um processo *gerenciador* que armazene o estado de todas as ações iniciadas de forma assíncrona (pendente ou concluída) e um processo *iterador* que, ao fim da computação, se comunique com esse gerenciador para esperar por ações assíncronas ainda pendentes. Finalmente, as regras (`wait`) e (`cancel`) indicam a necessidade da comunicação com esse gerenciador, para, dependendo do estado da ação assíncrona, bloquear o processo chamador ou cancelar o processo assíncrono criado para a ação.

4.5 Propriedades μ-Calculus Modal Padrão

As propriedades formais sobre *workflows* da HPC Shelf são especificadas através de fórmulas escritas na lógica modal μ-calculus modal. Neste trabalho, se está interessado em verificar propriedades de continuidade e de segurança, visando garantir, respectivamente, que computações orquestradas por *workflows* eventualmente alcancem certos estados desejáveis e que jamais alcancem certos estados indesejáveis.

Dado um código mCRL2 que foi traduzido pelo algoritmo S2m a partir de um código SAFeSWL, para que se verifique nele uma propriedade formal, primeiramente é necessário convertê-lo para uma especificação LPS. O passo seguinte consiste em converter a fórmula μ-calculus modal junto com a especificação LPS do programa para um sistema de equações PBES. Por fim, utiliza-se *model checking* para resolver o PBES. Se uma solução for encontrada, a propriedade foi satisfeita.

O exemplo mais comum de propriedade de segurança em um sistema concorrente é a *ausência de deadlock*, o qual é abordado adiante.

4.5.1 Ausência de *Deadlock*

Dado um LTS que descreve um sistema concorrente, diz-se que nele há *deadlock* quando existe um estado alcançável que não termina ou que não possui transições de saída. De outra maneira, pode-se dizer que em um LTS não existe *deadlock* quando todo estado é alcançável. Essa segunda definição permite derivar a seguinte fórmula μ-calculus modal, a partir da qual é possível verificar ausência de *deadlock* em um LTS:

$$\mathsf{AD}\text{: } [\mathsf{true}*]\langle\mathsf{true}\rangle\mathsf{true}$$

Todos os códigos apresentados no Apêndice A.1, bem como os códigos mCRL2 obtidos pelo algoritmo S2m para os *workflows* referentes aos estudos de casos relatados no Capítulo 6 foram verificados e não apresentaram *deadlock*.

4.5.2 Ausência de *Loops* Infinitos

Um programa SAFeSWL que possui uma tarefa de iteração (repeat) é passível de permanecer em *loop* infinito caso não exista uma tarefa break alcançável dentro do escopo da tarefa repeat. Uma maneira simples de verificar isso é verificar se todas as ações mCRL2 $break(i)$ podem ocorrer a partir de um determinado ponto em diante, em que i é o índice do repeat relacionado. A propriedade de ausência de loop infinito (ALI) é representada pela seguinte fórmula:

$$\mathsf{ALI}\text{: } \forall i : Nat.[\mathsf{true}*]\langle\mathsf{true} * .break(i)\rangle\mathsf{true}$$

4.5.3 Verificação da Consistência de Ativações das Ações da Porta do Ciclo de Vida dos Componentes com *Model Checking*

As verificações relativas às ativações de ações da porta *LifeCycle* podem ser expressas também através de fórmulas μ-calculus modal. Modelou-se aqui cada relação de precedência como uma fórmula distina.

A seguir, será apresentada cada fórmula e será comentado seu significado.

$$\text{VCV1: } \forall c : Nat.[!resolve(c) * .deploy(c)]\mathsf{false}$$
$$\&\& \langle \mathsf{true} * .resolve(c).!release(c) * .deploy(c)\rangle \mathsf{true}$$

Essa fórmula é iterada para todo componente de *id* c. A primeira parte da conjunção diz que nenhum **deploy** pode ocorrer sem ter havido um **resolve** antes. Note que ! significa complemento de conjuntos, portanto a expressão $[!a]$ false afirma que todas as evoluções feitas por uma ação diferente de a são proibidas. A segunda parte afirma que, dado que houve um **resolve**, e entre esse **resolve** e o **deploy** não houve nenhum **release**, o **deploy** pode ser realizado.

$$\text{VCV2: } \forall c : Nat.[!deploy(c) * .instantiate(c)]\mathsf{false}$$
$$\&\& \langle \mathsf{true} * .deploy(c).!release(c) * .instantiate(c)\rangle \mathsf{true}$$

Essa fórmula é semelhante à anterior, mas verificando a precedência entre **deploy** e **instantiate**.

$$\text{VCV3: } \forall c, a : Nat.[!instantiate(c) * .compute(c, a)]\mathsf{false}$$
$$\&\& \langle \mathsf{true} * .instantiate(c).!release(c) * .compute(c, a)\rangle \mathsf{true}$$

Essa fórmula também é similar, liberando ações computacionais dos componentes, aqui genericamente representadas por **compute**(*componente*, *ação*), somente após as instanciações dos componentes nas respectivas plataformas virtuais.

$$\text{VCV4: } \forall c : Nat.[!instantiate(c) * .release(c)]\mathsf{false}$$
$$\&\& \langle \mathsf{true} * .instantiate(c).!release(c) * .release(c)\rangle \mathsf{true}$$

Por fim, essa fórmula afirma que um **release** só pode ocorrer se tiver ocorrido um **instantiate** e que, dado que ocorreu um **instantiate** e um **release**, nenhum outro **release** para o mesmo componente ocorreu entre eles.

Uma observação importante é que, por questões de legibilidade, omitiu-se nessas fórmulas que o certificador restringe os valores de c para somente os valores de identificadores dos componentes presentes na orquestração, visando diminuir o espaço de busca no *model checking*.

Por fim, para que essa verificação seja habilitada no certificador o parâmetro de contexto ***life_cycle_verification*** com LifeCicleVerificationModelChecker.

4.6 Considerações Finais

Neste capítulo descreveu-se a arquitetura dos componentes certificadores de *workflows* científicos descritos usando o subconjunto de orquestração de SAFeSWL. Tais *workflows*

representam orquestrações de componentes geridas por componentes workflow de sistemas de computação paralela na HPC Shelf e alcançam o *status* de certificados através da verificação de propriedades de continuidade e de segurança sobre eles.

Com relação às contribuições deste capítulo, por um lado propôs-se uma modelagem formal dos *workflows* científicos da HPC Shelf. Nesse sentido, primeiro foi proposta uma caracterização geral dos *workflows* científicos, visando encontrar padrões verificáveis, e foi investigado como esses padrões ocorrem nos *workflows* da HPC Shelf. Em seguida, com base nesses padrões, foi estabelecida uma modelagem dos *workflows* da HPC Shelf para mCRL2. Essa modelagem, contudo, foi formalizada através da criação de uma semântica operacional para o subconjunto de orquestração de SAFeSWL, que, por sua vez, foi obtida através de uma nova gramática, dita gramática formal, para o subconjunto de orquestração de SAFeSWL. Os detalhes das regras de tradução geradas foram, por questão de organização, disponibilizados no Apêndice A.1. Por outro lado, outra contribuição importante recaiu sobre a definição (contratos contextuais) de um componente abstrato e um concreto para o certificador SWC2 e seu componente tático associado. Um conjunto inicial de propriedades padrão já verificáveis pelo certificador também foi proposto.

5 Componente Certificador de Componente de Computação (C4)

Neste capítulo, apresenta-se a arquitetura de um componente certificador de componentes de computação, doravante chamado de C4 (DANTAS; CARVALHO JUNIOR; BARBOSA, 2017a; DANTAS; CARVALHO JUNIOR; BARBOSA; PROENÇA, 2017). Essa arquitetura tem por finalidade principal permitir a verificação de propriedades funcionais e de segurança sobre programas escritos em linguagens de programação compatíveis com a HPC Shelf, através das quais componentes da espécie *computação* são desenvolvidos por usuários desenvolvedores e disponibilizados no catálogo de componentes do Core para serem utilizados em sistemas de computação paralela oferecidos por provedores de aplicações a seus usuários especialistas alvo.

A arquitetura dos certificadores de computação possui os seguintes objetivos específicos:

- Possibilitar a criação desses componentes certificadores para componentes da espécie computação por parte de usuários certificadores de componentes, utilizando contratos contextuais para exprimir características das ferramentas de verificação formal de *software* que esses componentes são capazes de aplicar na verificação de propriedades formais, as quais guiam o algoritmo de resolução de contratos contextuais e seleção de componentes implementado pelo *sistema de contratos contextuais*;
- Viabilizar a existência de diversos certificadores de computação, a partir do C4, os quais se diferenciam pelos conjuntos de ferramentas de verificação que empregam (componentes táticos). Os componentes táticos de um certificador determinam, dentre outras coisas, as linguagens de programação nas quais os programas dos componentes de computação a serem certificados pelo certificador devem ser escritos e o tipo de anotações (asserções) que tais programas devem carregar, caso devam ser anotados (*ferramentas assercionais*);
- Permitir que usuários desenvolvedores de componentes de computação possam criar componentes certificáveis, os quais podem, através da infraestrutura da nuvem, ser certificados, aumentando assim seus níveis de confiança perante usuários provedores de aplicações;
- Possibilitar ao usuário provedor de aplicações que, ao compor um sistema de computação paralela para resolver um problema específico do domínio do usuário especialista, o faça utilizando componentes de computação certificados ou certificáveis. No caso de um componente certificável, o provedor de aplicações pode escolher o momento durante a execução do *workflow* da aplicação em que a certificação ocorrerá (ação **certify**). Esse comportamento visa agregar mais qualidade à solução proposta e reduzir riscos de a aplicação alcançar um estado indesejável devido a uma falha

de programação;

- Permitir que o desenvolvedor de componentes possa criar propriedades funcionais e de segurança a serem verificadas nos programas que compõe para os componentes de computação que desenvolve. As propriedades podem ser especificadas na forma de asserções de especificação (*EsPre* e *EsPost*) diretamente nos programas (veja, por exemplo, a Seção 5.3), caso seja pretendido que a certificação seja feita por um certificador equipado com um componente tático assercional. Caso algum componente tático (assercional ou não) do componente certificador pretendido aceite propriedades externas (não anotadas), chamadas de *propriedades ad hoc*, essas propriedades podem ser especificadas pelo desenvolvedor do componente e incorporadas a ele no momento do seu registro no Core. Assim, no momento da execução do certificador, essas propriedades são recuperadas do componente de computação pelo SAFe e repassadas ao certificador, que as repassa ao componente tático responsável;
- Oferecer a possibilidade de que, a um mesmo componente de computação, possam ser ligados mais de um certificador, permitindo assim a execução de vários sistemas de certificação paralela diferentes para a sua certificação.

Na Seção 5.1, apresentam-se detalhes sobre os componentes táticos que podem ser orquestrados por componentes C4 com o objetivo de provar propriedades formais sobre os programas contidos em componentes de computação. O passo subsequente, na Seção 5.2, é a apresentação do contrato contextual do componente C4, o qual serve para guiar o sistema de contratos contextuais na escolha de um componente certificador adequado aos propósitos do desenvolvedor do componente. Finalmente, na Seção 5.3, apresenta-se um cenário hipotético que exemplifica o uso de certificadores de computação.

5.1 Componentes Táticos de Certificadores de Computação

Componentes táticos de certificadores de computação são categorizados de acordo com a técnica de verificação que utilizam, que pode ser verificação dedutiva de programas ou *model checking*. Antes de apresentá-los, contudo, serão feitos alguns comentários sobre as portas desses componentes.

A porta de serviços usuária, além de operações padrão para receber os programas e propriedades formais a serem verificadas sobre eles, possui operações para receber do componente certificador o mapeamento desse programas às unidades paralelas em que executarão no componente de computação. Alguns componentes táticos de certificadores de computação podem utilizar informações sobre as unidades em que os programas que verificam irão executar. A exemplo disso, os componentes táticos de *model checking* descritos mais adiante exigem saber a quantidade de unidades em que o programa SPMD que verificarão irá executar, a fim de reduzir o espaço de busca no *model checking*. Observe que no caso desses componentes táticos, os programas que verificam só são verificados

Tabela 1: Possibilidades de composição de Componentes Táticos de Verificação Dedutiva

Ling. Prog.	*Plugin*	*Frontend*	Ling. Interm.	Provador
C	Jessie, WP	Frama-C	Why3	Alt-Ergo, CVC3, CVC4, Z3, E, SPASS, Vampire, Gappa, Coq, Isabelle/Hol
Java		Krakatoa	Why3	Alt-Ergo, CVC3, CVC4, Z3, E, SPASS, Vampire, Gappa, Coq, Isabelle/Hol
Dafny		Dafny	Boogie	Z3
C, Java		VeriFast		Z3, Redux
C	ParTypes	VCC	Boogie	Z3

para o número específico de unidades a que são associados. Assim, se a propriedade for provada para um programa, no registro do componente de computação no Core, essa propriedade é considerada provada para esse programa somente para a quantidade de unidades a qual ele foi associado. Observe ainda que quando for solicitado o início da certificação do componente de computação, ou seja, quando a ação **certify** for ativada no *workflow*, é interessante que a plataforma de execução que executará o componente computação já esteja resolvida, uma vez que o mapeamento dos programas às unidades do componente computação só poderá ser de fato conhecido quando se souber quantas unidades do componente serão lançadas, número esse que é igual à quantidade de unidades de processamento da plataforma virtual escolhida. Essa porta possui também uma operação que permite que o certificador recupere um arquivo contendo VCs na forma de teoremas não provados que o ator que comandou a certificação terá que provar, caso o componente tático possua um provador interativo. Por fim, possui uma operação para receber do componente certificador o arquivo contendo VCs (teoremas) provadas através do provador interativo.

5.1.1 Componentes Táticos de Verificação Dedutiva

Um componente tático de verificação dedutiva é geralmente composto por uma composição adequada de um *frontend* de verificação, uma linguagem de verificação intermediária e um provador (automático ou interativo). Outros elementos podem aparecer, como no caso dos *plugins* de *frontends* de verificação, os quais visam adicionar aos últimos novas características de verificação. Outros componentes táticos, entretanto, não utilizam linguagem de verificação intermediária, como no exemplo dos que utilizam o *frontend* VeriFast, o qual gera VCs diretamente para provadores.

Na Tabela 1, são mostradas as possibilidades de composição de componentes táticos de verificação dedutiva da arquitetura de certificadores de computação. Na referida tabela, também é mostrada a linguagem de programação que o componente tático está apto a compreender. Vale lembrar que, devido ao fato de a HPC Shelf utilizar atualmente o padrão de interoperabilidade Mono, os componentes de computação podem ser escritos em diversas linguagens de programação aceitas por esse padrão e compiladas diretamente para a plataforma virtual alvo (compilação nativa). Outro fato importante é que no protótipo do arcabouço de certificação desenvolvido para a Tese, foram implementados somente os componentes táticos utilizados nos estudos de caso (Capítulo 6).

Quando há somente uma possibilidade de composição para o componente tático, seu nome é o nome da sua ferramenta de verificação mais especializada. Por exemplo, o componente tático ParTypes é na verdade composto por ParTypes/VCC/Boogie/Z3. Quando há mais de uma possibilidade de composição, o nome do tático é composto por excertos que vêm dos nomes das ferramentas que o compõem de uma forma a identificar o tático de forma única. Por exemplo, JFWCVC4 é composto por Jessie/Frama-C/Why3/CVC4.

Componentes táticos de verificação dedutiva são *assercionais*, ou seja, os programas que recebem devem ser decorados com asserções da lógica de Floyd-Hoare ou suas derivadas. Chama-se um componente tático de *assercional puro* quando as propriedades que verifica são na forma de asserções de especificação (*EsPre* e *EsPost*), ou seja, pré e pós-condições de métodos. Por sua vez, chama-se um componente tático de *assercional não puro* quando as propriedades que ele verifica são *ad hoc*, ou seja, propriedades criadas pelo desenvolvedor de componentes e gravadas no componente de computação. Atualmente, o único componente tático assercional não puro é o ParTypes, o qual verifica se um programa MPI anotado implementa um dado protocolo de alto nível que reflete suas operações de comunicação. O protocolo é fornecido ao componente tático como uma propriedade *ad hoc*.

Componentes táticos de verificação dedutiva são, na maioria das vezes, empregados para provar propriedades funcionais sobre programas, ou seja, provar asserções de especificação que refletem a corretude do sistema. Visando exemplificar o uso dessas asserções, apresenta-se, na Seção 5.3, um exemplo de componente de computação hipotético cujas asserções de especificação visam garantir que, ao final do programa, o vetor dado como entrada encontra-se ordenado.

5.1.2 Componentes Táticos de *Model Checking*

O conjunto inicial de componentes táticos de *model checking* para a arquitetura dos certificadores de componentes de computação é composto somente pelos componentes táticos ISP e CIVL. Ambos verificam conjuntos fixos, porém expressivos, de propriedades de segurança. O CIVL, entretanto, permite ainda verificar se um programa do componente computação é funcionalmente equivalente a outro programa fornecido. Esse segundo programa é recebido pelo componente tático como uma propriedade *ad hoc*.

5.2 Contratos Contextuais

Enquanto o contrato contextual do componente SWC2 reflete as propriedades de continuidade e de segurança que os certificadores são capazes de verificar, o contrato contextual do componente C4 visa expressar características dos conjuntos de ferramentas de verificação formal de *software* que os certificadores são capazes de orquestrar. A seguir, apresenta-se

a assinatura contextual do certificador C4, com um conjunto inicial de parâmetros de contexto:

C4 [***programming_ language*** $= P$: PLType,
separation_ logic $= S$: SLType,
floating_ point_ operations $= F$: FPOType,
existential_ quantifier $= E$: EQType,
interactive_ proof $= I$: IPType,
message_ passing_ interface_ verif $= M$: MPIVerifType,
ad_ hoc_ properties $= A$: AHType,
max_ verification_ time $= MV$: Integer]

A descrição dos parâmetros de contexto ainda não apresentados é apresentada a seguir:

- ***programming_ language*** denota a linguagem de programação na qual os programas dos componentes de computação alvo devem ser escritos;
- ***separation_ logic*** afirma se o certificador utiliza componentes táticos que verificam programas anotados com asserções da lógica de separação;
- ***floating_ point_ operations*** indica se o certificador é capaz de verificar operações de ponto flutuante (atualmente possível através do emprego de componentes táticos que utilizam o provador Gappa);
- ***existential_ quantifier*** afirma se nas asserções de programas dos componentes de computação a serem verificados pelo certificador existem quantificadores existenciais;

  Estudos comprovam que provadores SMT não são bons em encontrar valores para as variáveis quantificadas existencialmente de forma a tornar a fórmula verdadeira (*testemunhas*) (BOBOT; FILLIÂTRE; MARCHÉ; PASKEVICH, 2014). Nesses casos, é aconselhável que a orquestração do certificador aplique primeiro componentes táticos com provadores ATP para depois aplicar componentes táticos com provadores SMT.

- ***message_ passing_ interface_ verif*** afirma se o certificador é capaz de verificar programas MPI.

Note a diferença entre o parâmetro ***message_ passing_ interface_ verif*** e o parâmetro ***message_ passing_ interface*** de componentes táticos, o qual afirma que o componente tático foi implementado através da biblioteca MPI.

Abaixo, um exemplo hipotético de contrato contextual para essa assinatura:

C4Impl : C4 [***programming_language*** = C&Java,
separation_logic = SeparationLogic,
floating_point_operations = NoFloatingPointOperations,
existential_quantifier = NoExistentialQuantifier,
interactive_proof = NoInteractiveProof,
message_passing_interface_verif = MPIVerif,
ad_hoc_properties = NoAdHocProperties,
max_verification_time = 20]

Através desse contrato, podem-se fazer as seguintes afirmações sobre o certificador concreto C4Impl:

- certifica componentes de computação cujos programas podem ser escrito em C, Java ou ambos;
- é capaz de fazer verificações de asserções da lógica de separação;
- não verifica operações de ponto flutuante;
- não aplica nenhum tratamento especial quanto à presença de quantificadores existenciais nas asserções;
- não permite a utilização de provadores interativos;
- pode verificar propriedades sobre programas MPI;
- não recebe propriedades não anotadas nos programas (*ad hoc*);
- o tempo de verificação máximo inicial calculado experimentalmente pelo certificador de componentes foi 20 segundos.

Na Seção 5.3, será apresentado um cenário que ilustra a utilização do certificador C4Impl.

Um desenvolvedor de componentes de computação pode incluir em uma ligação de certificação a seguinte valoração:

C4 [***programming_language*** = C,
separation_logic = SeparationLogic,
ad_hoc_properties = NoAdHocProperties,
message_passing_interface_verif = MPIVerif]

Essa valoração afirma que esse ator deseja verificar automaticamente e sem ter que fornecer propriedades *ad_hoc* programas do componente de computação que desenvolve escritos na linguagem C, podendo alguns conterem asserções da lógica de separação e alguns serem escritos utilizando a biblioteca MPI. Certamente, o componente concreto C4Impl é um candidato retornado pelo Core para essa valoração.

Como dito anteriormente, as propriedades que um certificador de computação pode verificar podem ser anotadas nos programas (asserções de especificação) ou fornecidas pelo componente de computação ao certificador (propriedades *ad hoc*). Novamente, as propriedades podem ser divididas em *obrigatórias* e *optativas*. Para que uma asserção de especificação seja considerada obrigatória, ela deve ser precedida da anotação *@Man-*

Figura 17: Código de orquestração TCOL do componente certificador C4Impl

```
<parallel>
        <invoke action="verify_perform" id_port="verify-JFWA" /> <!--Jessie/Frama-C/Why3/Alt-Ergo-->
        <invoke action="verify_perform" id_port="verify-FJWZ" /> <!--Jessie/Frama-C/Why3/Z3-->
        <invoke action="verify_perform" id_port="verify-FJWCVC3" /> <!--Jessie/Frama-C/Why3/CVC3-->
        <invoke action="verify_perform" id_port="verify-FJWCVC4" /> <!--Jessie/Frama-C/Why3/CVC4-->
        <invoke action="verify_perform" id_port="verify-VeriFast" /> <!--VeriFast/Z3-->
        <invoke action="verify_perform" id_port="verify-ISP" /> <!--ISP-->
</parallel>
```

Fonte: Elaborado pelo autor.

datory. Caso contrário, ela será considerada opcional. Com relação às propriedades *ad hoc*, o desenvolvedor do componente de computação é quem faz essa distinção. Por fim, todas as propriedades fixas de componente tático, como no caso do ISP e do CIVL, são consideradas obrigatórias.

Para o componente C4Impl, suponha que foram associados no momento do desenvolvimento os componentes táticos abstratos JFWA, FJWZ, FJWCVC3, FJWCVC4, VeriFast e ISP, que somente estendem o componente tático abstrato Tactical, definido na Seção 4.3, sem adicionar nenhum parâmetro. Os quatro primeiros componentes táticos são assercionais puros e empregados na verificação de programas sequenciais decorados com asserções da lógica Hoare. Por sua vez, o quinto é assercional puro e verifica programas sequenciais ou paralelos com memória compartilhada decorados com asserções da lógica de separação. Finalmente, o sexto verifica um conjunto fixo de propriedades de segurança em programas MPI/OpenMP. Como componentes táticos concretos (hipotéticos) para esses componentes, citam-se, respectivamente, JFWAImpl, FJWZImpl, FJWCVC3Impl, FJWCVC4Impl, VeriFastImpl e ISPImpl, os quais foram implementados usando MPICH2 e requerem que cada unidade de processamento da plataforma virtual alvo possua no mínimo 4 núcleos de processamento. Como todos os contratos são similares, será apresentado somente o contrato de ISPImpl:

$$\text{ISPImpl} : \text{ISP}\ [\boldsymbol{message\_passing\_interface} = \text{MPICH2}, \boldsymbol{number\_cores} = 4]$$

Em C4Impl, a valoração às assinaturas dos componentes táticos abstratos associados foi semelhante. Por exemplo, no caso de ISP, tem se a seguinte valoração:

$$\text{ISP}\ [\boldsymbol{message\_passing\_interface} = \text{MPI}]$$

Através dessa valoração, pode-se dizer que o certificador concreto C4Impl tem interesse que para o tático abstrato ISP seja escolhida uma implementação paralela MPI. Claramente, ISPImpl é um candidato a essa valoração.

A orquestração feita pelo componente C4Impl é descrita na Figura 17. Nela, esse componente ativa em paralelo as ações **verify_perform** de todos os componentes táticos associados. Por questões de espaço, foram omitidas as ativações das ações do ciclo de vida dos componentes. Ao lado de cada operação, um comentário diz qual é a infraestrutura de verificação do componente tático.

5.3 Exemplo: Componente para Ordenação de Vetores

Nesta seção, será apresentado um exemplo que ilustrará o uso de componentes C4 para verificar propriedades funcionais e de segurança sobre um componente da espécie computação hipotético chamado SORTING, o qual visa fornecer implementações paralelas para algoritmos de ordenação de vetores de números inteiros, como *QuickSort*, *MergeSort*, *HeapSort*, etc. A apresentação da assinatura contextual do seu componente abstrato e dos contratos contextuais de seus componentes concretos será ignorada, uma vez que é dispensável para os propósitos desta seção.

O cenário natural para a utilização de SORTING consiste na existência de um componente abstrato e diversos componentes concretos seus, em que cada um desses últimos apresentaria exclusivamente uma versão paralela de cada algoritmo de ordenação. Todavia, visando melhor exemplificar a orquestração feita pelo certificador, supõe-se que existe um componente concreto de SORTING, chamado SortingImpl, que utiliza tantas unidades paralelas quantos forem os núcleos de processamento da plataforma virtual alvo (suponha $n > 5$) e possui três versões paralelas para o algoritmo *QuickSort* escritas em C, que se deseja verificar. São elas:

- `QuickSortS.c`: Consiste em um programa paralelo que dispara nas unidades 1 a 4 execuções da versão sequencial do *QuickSort* sobre porções distintas do vetor original e em seguida combina os resultados. Para esse caso, o desenvolvedor do componente tem interesse em verificar somente o programa sequencial (`QuickSortSeq.c`) e não o programa combinador. O programa `QuickSortSeq.c` é anotado com asserções da lógica de Hoare que verificam, dentre outras coisas, se, ao final da execução, o vetor está ordenado e é uma permutação do original;
- `QuickSortT.c`: Consiste em um programa paralelo com memória compartilhada que é capaz de lançar tantas *threads* quantos forem os núcleos disponíveis na unidade (isto é controlado pelo parâmetro *depth* do programa). Nesse exemplo, esse programa é lançado somente na unidade 5 do componente, cuja quantidade de núcleos será a quantidade de núcleos da unidade de processamento que executará a unidade 5 na plataforma virtual a ser escolhida pelo Core. É anotado com asserções da lógica de separação que visam garantir somente que o vetor de saída é um vetor de inteiros. Implicitamente, garantem também que o programa nunca faz acessos a posições de memórias não alocadas;
- `QuickSortM.c`: Consiste em um programa paralelo com memória distribuída que utiliza a biblioteca MPI e executa paralelamente nas n unidades do componente. Nesse programa, somente a unidade que recebeu *rank* 0 pelo ambiente MPI recebe o vetor a ser ordenado. Ela então o divide entre as demais, que ordenam suas respectivas partes. Em seguida, todas as unidades executam rodadas de intercalação de suas partes com as dos demais. Ao final da execução, a unidade de *rank* 0 possui

o vetor ordenado.

As figuras 18, 19 e 20 mostram os códigos desses programas. Observe anotações *@Mandatory* precedendo algumas asserções de especificação.

É importante lembrar que o paralelismo oferecido pelo arcabouço de certificação pode se dar em três níveis:

- um componente certificador repassa um conjunto de programas a um conjunto de componentes táticos, disparados em paralelo (tarefas start e parallel de TCOL);
- cada componente tático divide os programas de sua responsabilidade entre seu conjunto de unidades paralelas;
- cada unidade paralela de um componente tático pode dividir o conjunto de programas de sua responsabilidade entre um conjunto de *threads*.

Contudo, note que outros níveis mais finos de paralelismo poderiam ser alcançados pela própria ferramenta de verificação dedutiva, a nível de asserções ou condições de verificação. No segundo caso, por exemplo, o provador automático Z3 possuía uma funcionalidade que permitia que ele pudesse disparar *threads* e dividir entre elas o conjunto de VCs a ser provado. Porém, essa funcionalidade encontra-se desabilitada atualmente. Com relação ao primeiro caso, considere as asserções das linhas 61 e 62 da Figura 18 (`QuickSortSeq.c`). Claramente, as VCs com relação às duas podem ser demonstradas separadamente e poderiam ser demonstradas paralelamente no Frama-C, caso essa ferramenta possuísse essa funcionalidade.

A Figura 21 mostra a arquitetura de certificação do componente SortingImpl. Nela, o desenvolvedor de SortingImpl, em tempo de desenvolvimento do componente, aninhou um componente certificador abstrato C4 (ligação de certificação), forneceu a valoração descrita na Seção 5.2, fazendo com que o certificador C4Impl seja um candidato, e informou qual componente tático de C4Impl verificará cada programa (`QuickSortSeq.c` em JFWA, JFWZ, JFWCVC3 e JFWCVC4; `QuickSortT.c` em VeriFast; e `QuickSortM.c` em ISP). Observe que quando C4 for resolvido, no momento da execução do processo de certificação, pode ser escolhido um certificador concreto diferente de C4Impl e que não é associado aos mesmos componentes táticos. Dessa forma, as propriedades que seriam verificadas sobre os programas atribuídos a um componente tático que não esteja associado ao certificador concreto escolhido serão automaticamente consideradas inconclusivas.

Suponha que quando o processo de certificação for iniciado no SAFe, através da ativação da ação **certify** do componente durante a execução do *workflow*, C4Impl foi escolhido e foram escolhidos para ele os componentes táticos concretos apresentados na Seção 5.2, para os quais foram escolhidas plataformas virtuais contendo somente uma unidade de processamento.

Ao final do processo de certificação, o vetor associativo **formal_properties**, mantido pelo certificador, possui o resultado da verificação das propriedades formais. Todas as propriedades obrigatórias foram provadas, fazendo com que Core registre no

Figura 18: Programa `QuickSortSeq.c`, que utiliza asserções da Lógica de Hoare

```
#pragma JessieIntegerModel(math)
/*@ predicate Swap{L1,L2}(int *a, integer i, integer j) =
  @   \at(a[i],L1) == \at(a[j],L2) &&
  @   \at(a[j],L1) == \at(a[i],L2) &&
  @   \forall integer k; k != i && k != j
  @       ==> \at(a[k],L1) == \at(a[k],L2);
  @*/
/*@ inductive Permut{L1,L2}(int *a, integer l, integer h) {
  @  case Permut_refl{L}:
  @   \forall int *a, integer l, h; Permut{L,L}(a, l, h) ;
  @  case Permut_sym{L1,L2}:
  @    \forall int *a, integer l, h;
  @      Permut{L1,L2}(a, l, h) ==> Permut{L2,L1}(a, l, h) ;
  @  case Permut_trans{L1,L2,L3}:
  @    \forall int *a, integer l, h;
  @      Permut{L1,L2}(a, l, h) && Permut{L2,L3}(a, l, h) ==>
  @        Permut{L1,L3}(a, l, h) ;
  @  case Permut_swap{L1,L2}:
  @    \forall int *a, integer l, h, i, j;
  @       l <= i <= h && l <= j <= h && Swap{L1,L2}(a, i, j) ==>
  @     Permut{L1,L2}(a, l, h) ;
  @ }
  @*/

/*@ requires \valid(A+i) && \valid(A+j);
  @ assigns A[i],A[j];
  @ behavior swap: ensures Swap{Old,Here}(A,i,j);
  @*/
void swap(int A[], int i, int j) {
    int tmp = A[i];
    A[i] = A[j];
    A[j] = tmp;
}

/*@ requires \valid(A+(le..ri));
  @ assigns A[le..ri];
  @ behavior pivot1: ensures \forall integer k; le <= k <= \result ==> A[k] <= A[\result];
  @ behavior pivot2: ensures \forall integer k; \result < k <= ri==> A[\result] < A[k];
  @ behavior permut_partition: ensures Permut{Old,Here}(A,le,ri);
  @*/
int partition(int A[], int le, int ri) {
    int center,piv,i;
    L0:center = A[le];
    piv = le;
    /*@ loop invariant \forall integer j; le < j <= piv ==> A[j] < center;
      @ loop invariant \forall integer j; piv < j <  i ==> A[j] >= center;
      @ loop invariant A[le] == center && le <= piv < i <= ri+1;
      @ loop invariant Permut{L0,Here}(A,le,ri);
      @ loop variant ri-i;
      @*/
    for (i = le + 1; i <= ri; i++) {
        if (A[i] <= center) {
            piv=piv+1;
            L:swap(A,i,piv);
            //@ assert Permut{L,Here}(A,le,ri);
        }
    }
    //@ assert le <= piv <= ri;
    L1:swap(A,le,piv);
    //@ assert Permut{L1,Here}(A,le,ri);
    //@ assert \forall integer k; le <= k <= piv ==> A[k] <= A[piv];
    //@ assert \forall integer k; piv < k <= ri==> A[piv] < A[k];
    return piv;
}

/*@ predicate sorted(int *a, integer l, integer h) =
  @   \forall integer i,j; l <= i <= j < h ==> a[i] <= a[j] ;
  @*/

/*@ requires \valid(A+(le..ri));
  @ decreases ri-le;
  @Mandatory
  @ behavior sorted: ensures sorted(A,le,ri+1);
  @Mandatory
  @ behavior permut: ensures Permut{Old,Here}(A,le,ri);
  @*/
void QuickSortSeq(int A[], int le, int ri) {
    int piv;
    if (ri>=le) {
        L:piv = partition(A,le,ri);
        //@ assert \forall integer k; le <= k <= piv ==> A[k] <= A[piv];
        //@ assert \forall integer k; piv < k <= ri==> A[piv] < A[k];
        //@ assert Permut{L,Here}(A,le,ri);
        L1:QuickSortSeq(A,le,piv-1);
        //@ assert Permut{L1,Here}(A,le,ri);
        //@ assert sorted(A,le,piv-1);
        L2:QuickSortSeq(A,piv+1,ri);
        //@ assert Permut{L2,Here}(A,le,ri);
        //@ assert sorted(A,piv+1,ri);
        //@ assert sorted(A,le,ri+1);
    }
}
```

Fonte: Elaborado pelo autor.

Figura 19: Programa `QuickSortT.c`, que utiliza asserções da Lógica de Separação

```
#include <pthread.h>
/*@
    predicate int_array(int *A, int n)
        requires 0 <= n &*& n == 0 ? emp : integer(A, _) &*& int_array(A + 1, n - 1);
    lemma void split(int *A, int offset);
        requires int_array(A, ?n) &*& 0 <= offset &*& offset <= n;
        ensures int_array(A, offset) &*& int_array(A + offset, n - offset);
    lemma void merge(int *A);
        requires int_array(A, ?n) &*& int_array(A + n, ?m);
        ensures int_array(A, n + m);
@*/
int partition(int n, int *A)
//@ requires int_array(A, n) &*& 0<=n &*& 2<=n;
//@ ensures int_array(A, n) &*& 0<=n &*& 2<=n &*& 0<=result &*& result<=n; {
    //@ open int_array(A, n);
    int pivot = *A;
    //@ close int_array(A, n);
    int l = 0;
    int r = n;
    while (l < r)
        /*@ invariant 0 <= l &*& l <= r &*& r <= n &*&
                int_array(A, n);
                  @*/ {
        //@ split(A, r - 1);
        //@ open int_array(A + r - 1, n - (r - 1));
        int b = *(A + r - 1);
        //@ close int_array(A + r - 1, n - (r - 1));
        //@ merge(A);
        if (b < pivot) {
            //@ split(A, l);
            //@ open int_array(A + l, n - l);
            int a = *(A + l);
            //@ close int_array(A + l, n - l);
            //@ merge(A);
            //@ split(A, r - 1);
            //@ open int_array(A + r - 1, n - (r - 1));
            *(A + r - 1) = a;
            //@ close int_array(A + r - 1, n - (r - 1));
            //@ merge(A);
            //@ split(A, l);
            //@ open int_array(A + l, n - l);
            *(A + l) = b;
            //@ close int_array(A + l, n - l);
            //@ merge(A);
            l = l + 1;
        } else {
            r = r - 1;
        }
    }
    return l;
}
struct my_data {
    int *A;
    int n;
    int depth;
};

/*@
predicate_family_instance pthread_run_pre(QuickSort_Thread)(struct my_data *data, any info) =
    [1/2]data->n |-> ?n &*& [1/2]data->A |-> ?A &*& [1/2]data->depth |-> ?depth &*&
    0 <= n &*& 0 < depth &*& int_array(A, n);
predicate_family_instance pthread_run_post(QuickSort_Thread)(struct my_data *data, any info) =
    [1/2]data->n |-> ?n &*& [1/2]data->A |-> ?A &*& [1/2]data->depth |-> ?depth &*&
    0 <= n &*& 0 < depth &*& int_array(A, n);
@*/

void* QuickSort_Thread(struct my_data *data) //@ : pthread_run_joinable
//@ requires pthread_run_pre(QuickSort_Thread)(data, ?info);
//@ ensures pthread_run_post(QuickSort_Thread)(data, info) &*& result == 0; {
    //@ open pthread_run_pre(QuickSort_Thread)(_, _);
    QuickSortT(data->A,data->n,data->depth);
    //@ close pthread_run_post(QuickSort_Thread)(data, info);
    return 0;
}
void QuickSortT(int *A, int n, int depth)
//@ requires int_array(A, n) &*& 0 <= n &*& 1 <= depth;
//@Mandatory
//@ ensures int_array(A, n); {
    if(2<=n) {
        if(depth > 0) {
            int l = partition(n,A);
            //@ split(A,l);
            struct my_data data1;
            pthread_t pthr1;
            data1.A=A;
            //@ close my_data_A(&data1,A);
            data1.n=l;
            //@ close my_data_n(&data1,l);
            data1.depth=depth;
            //@ close my_data_depth(&data1,depth);
            //@ close pthread_run_pre(QuickSort_Thread)(&data1, unit);
            pthread_create(&pthr1, (void *) 0, &QuickSort_Thread, &data1);
            QuickSortT(A+l,n-l,depth);
            pthread_join(pthr1,(void *) 0);
            //@ open pthread_run_post(QuickSort_Thread)(&data1, _);
            //@merge(A);
        }
    }
}
```

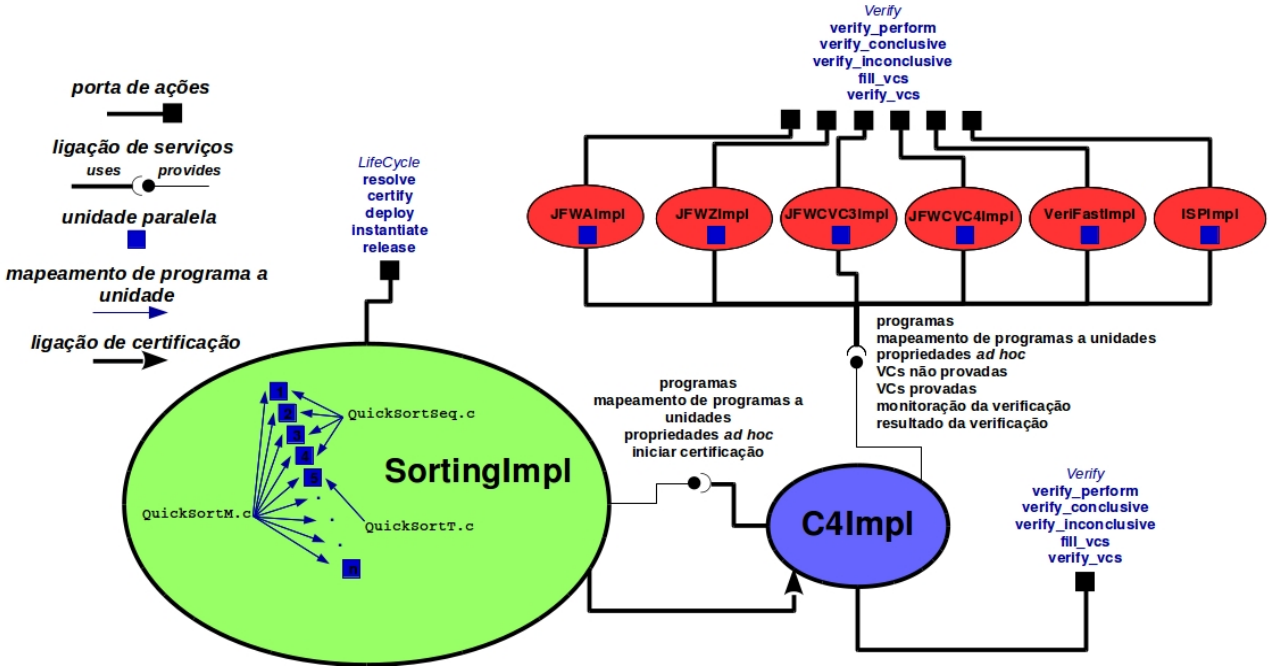
Fonte: Elaborado pelo autor.

Figura 20: Programa `QuickSortM.c`

```c
#include <mpi.h>
#include <stdlib.h>
int * merge(int *A1, int n1, int *A2, int n2) {
    int i=0,j=0,k=0,*result;
    result = (int *)malloc((n1+n2)*sizeof(int));
    while(i<n1 && j<n2) {
        if(A1[i]<A2[j]) {
            result[k] = A1[i];
            i++;
            k++;
        } else {
            result[k] = A2[j];
            j++;
            k++;
        }
    }
    if(i==n1) {
        while(j<n2) {
            result[k] = A2[j];
            j++;
            k++;
        }
    } else {
        while(i<n1) {
            result[k] = A1[i];
            i++;
            k++;
        }
    }
    return result;
}
void swap(int *A, int i, int j) {
    int t = A[i];
    A[i] = A[j];
    A[j] = t;
}
void QuickSort(int *A, int i, int j) {
    if(i>=j) {
        return;
    }
    int k,last=i;
    swap(A,i,(i+j)/2);
    for (k=i+1;k<=j;k++) {
        if(A[k]<A[i]) {
            swap(A,++last,k);
        }
    }
    swap(A,i,last);
    QuickSort(A,i,last-1);
    QuickSort(A,last+1,j);
}
void QuickSortM(int *A, int n, int rank, int number_procs, int comm) {
    int *chunk,*other,m,s,i,step=1;
    MPI_Status status;
    if(rank==0) {
        s = n/number_procs;
        MPI_Bcast(&s,1,MPI_INT,0,comm);
        chunk = (int *)malloc(s*sizeof(int));
        MPI_Scatter(A,s,MPI_INT,chunk,s,MPI_INT,0,comm);
        QuickSort(chunk,0,s-1);
    } else {
        MPI_Bcast(&s,1,MPI_INT,0,comm);
        chunk = (int *)malloc(s*sizeof(int));
        MPI_Scatter(A,s,MPI_INT,chunk,s,MPI_INT,0,comm);
        QuickSort(chunk,0,s-1);
    }
    while(step<number_procs) {
        if(rank%(2*step)==0) {
            if(rank+step<number_procs) {
                MPI_Recv(&m,1,MPI_INT,rank+step,0,comm,&status);
                other = (int *)malloc(m*sizeof(int));
                MPI_Recv(other,m,MPI_INT,rank+step,0,comm,&status);
                chunk = merge(chunk,s,other,m);
                s = s+m;
            }
        } else {
            int near = rank-step;
            MPI_Send(&s,1,MPI_INT,near,0,comm);
            MPI_Send(chunk,s,MPI_INT,near,0,comm);
            break;
        }
        step = step*2;
    }
    if(rank==0) {
        for (i=0;i<n;i++) {
            A[i] = chunk[i];
        }
    }
    free(other);
    free(chunk);
}
```

Fonte: Elaborado pelo autor.

Figura 21: Arquitetura de certificação de SortingImpl



Fonte: Elaborado pelo autor.

componente computação que ele é certificado, informando também quais propriedades foram provadas sobre cada programa. Dentre as diversas entradas desse vetor, as mais importantes são:

- **formal_properties**[“`QuickSortSeq.c`” , “sorted(A,le,ri+1)”] = true
- **formal_properties**[“`QuickSortSeq.c`” , “Permut{Old,Here}(A,le,ri)”] = true
- **formal_properties**[“`QuickSortT.c`” , “int_array(A,n)”] = true
- **formal_properties**[“`QuickSortT.c`” , “no access to unallocated memory”] = true
- **formal_properties**[“`QuickSortM.c`” , “no deadlock”] = true
- **formal_properties**[“`QuickSortM.c`” , “no MPI object leaks”] = true
- **formal_properties**[“`QuickSortM.c`” , “no communication races”] = true
- **formal_properties**[“`QuickSortM.c`” , “no irrelevant barriers”] = true

5.4 Considerações Finais

Neste capítulo descreveu-se a arquitetura dos componentes certificadores de componentes de computação (C4). Tais certificadores orquestram componentes táticos visando certificar componentes de sistemas de computação paralela da HPC Shelf e a certificação é alcançada mediante a verificação de propriedades funcionais e de segurança sobre os componentes.

Como contribuições desse capítulo, é possível elencar, por um lado, um estudo exploratório sobre as diferentes ferramentas e técnicas de verificação formal de *software* que pudessem ser adequadas a verificar propriedades nos programas contidos nos componentes de computação da HPC Shelf. Viu-se que existem dois grandes grupos de téc-

nicas/ferramentas: as de verificação dedutiva, que verificam propriedades em programas utilizando um conjunto de regras de inferência, e as de *model checking*, que visam verificar propriedades enumerando apenas os estados relevantes do programa. É importante ressaltar que o estudo exploratório deste trabalho revelou um crescimento considerável, na última década, no número de ferramentas desse tipo voltadas à certificação de aplicações reais. Embora a maior parte dessas ferramentas se concentre ainda em verificações de programas sequenciais, observa-se uma tendência de incorporar a essas ferramentas mecanismos para verificação de programas concorrentes. Ao passo que isso se desenvolva, as novas ferramentas de verificação poderão ser incorporadas à arquitetura descrita nesta Tese, sob a abstração de componentes táticos, sem demandar esforço significativo. Por outro lado, outra contribuição importante consistiu na definição (contratos contextuais) de um componente abstrato e um concreto para o certificador C4, os quais visam expressar características dos conjuntos de componentes táticos associados aos certificadores.

6 Estudos de Caso

Este capítulo tem por finalidade apresentar dois estudos de caso que visam evidenciar a eficácia da arquitetura dos componentes certificadores proposta no tocante à sua utilização em aplicações reais. Os estudos de caso dizem respeito, respectivamente, à certificação de um *workflow* clássico da arquitetura de processamento MapReduce, e à certificação dos componentes da ferramenta Montage que utilizam paralelismo. Nas seções que se seguem, serão detalhados tais estudos de caso e também os passos efetuados para a certificação dos componentes.

6.1 Estudo de Caso 1: MapReduce

O MapReduce consiste em um modelo de programação e uma implementação associada voltados ao processamento e geração de grandes conjuntos de dados. Sua arquitetura inicial, proposta pela *Google Inc.* em (DEAN; GHEMAWAT, 2008), teve raízes na programação funcional e consiste em o usuário especificar uma função de mapeamento, comumente chamada *map*, que processa um conjunto de pares *chave/valor* para gerar um conjunto intermediário de pares *chave/valor*, e uma função de redução, comumente chamada *reduce* ou *fold*, que combina os valores intermediários associados a uma mesma *chave* intermediária.

A implementação da *Google*, feita em C++ e RPC, preza ainda que, para um dado conjunto de programas escritos segundo esse estilo de programação, podem ser automaticamente lançados em paralelo, deixando a cargo do motor de execução, os detalhes do particionamento dos dados de entrada, o escalonamento da execução de tais programas nos nós de processamento disponíveis (balanceamento de carga), o tratamento de falhas e o gerenciamento da intercomunicação entre os nós. Portanto, o principal intuito dessa implementação é permitir a utilização de um sistema paralelo em larga escala por usuários que não necessariamente possuam experiência em programação paralela.

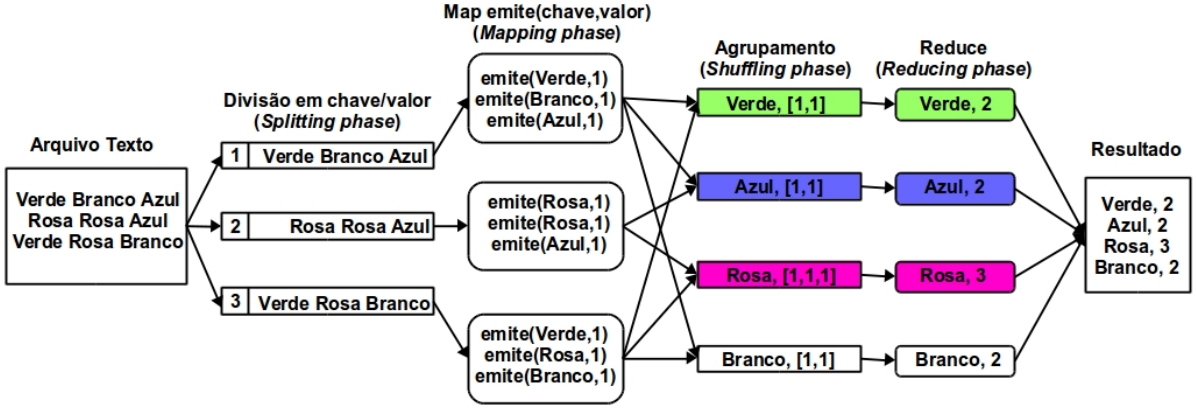
Diversas outras implementações foram propostas como variações do modelo original da *Google*, dentre as quais destacam-se o Hadoop[43], da *Apache Software Foundation*, e o MR-MPI (PLIMPTON; DEVINE, 2011).

A implementação *open-source* do Hadoop se tornou bastante popular na última década devido a sua utilização na análise de conjuntos de dados em larga escala em empresas voltadas ao armazenamento de dados, como a *Yahoo! Inc.*, e também em grupos de pesquisa em universidades, devido à sua disponibilização gratuita. Sua implementação foi realizada utilizando a linguagem Java. Entretanto, as funções de mapeamento e de redução podem ser escritas em outras linguagens.

O MR-MPI, diferentemente das duas anteriores, implementa apenas a funci-

[43] <http://hadoop.apache.org>

Figura 22: Exemplo clássico de contagem de palavras com MapReduce



Fonte: Elaborado pelo autor.

onalidade básica do MapReduce e é voltado exclusivamente à computação científica. É escrito em C++ e MPI, mas também permite a escrita de funções de mapeamento e de redução em C ou Python. Um ponto forte do MR-MPI é sua *flexibilidade* em permitir que chamadas MPI possam ser feitas diretamente e utilizando informações sobre a execução MapReduce armazenadas em estruturas de dados gerenciadas pelo próprio MR-MPI.

O modelo MapReduce é objeto de estudo no grupo de pesquisa no qual o autor desta Tese faz parte. Na Tese de Doutorado defendida por Cenez Araújo Rezende (REZENDE, 2017), uma das contribuições é um arcabouço orientado a componentes para construção de sistemas de computação paralela baseados em MapReduce, para a plataforma HPC Shelf. Por sua vez, na Tese de Doutorado de Jefferson Carvalho Silva (SILVA, 2016), utilizou-se uma versão prévia desse arcabouço como estudo de caso para avaliação do SAFe, notadamente referente à sua expressividade para descrição e execução de *workflows* científicos. Nesta Tese, utiliza-se o mesmo estudo de caso dessa última Tese para avaliar o arcabouço de certificação de componentes da HPC Shelf.

6.1.1 Um *Workflow* de Processamento MapReduce: Contador de Palavras

O exemplo *Contador de Palavras* com MapReduce é clássico na literatura para ilustrar a aplicação de tal modelo para contar ocorrências de palavras em um texto. Na Figura 22, apresenta-se um exemplo simples para este cenário. Nele, um arquivo de texto contém frases (linhas) formadas pela combinação das palavras *verde*, *branco*, *azul* e *rosa*. Ao final do processamento, espera-se obter quantas palavras existem para cada cor.

Na fase inicial (divisão, ou *splitting*), agrupam-se as frases em tuplas de *chave/valor*, em que a chave é um número inteiro que representa o número da linha e o valor é a frase em si. Dependendo da chave, as frases são distribuídas a um conjunto de agentes de mapeamento paralelos, os quais aplicam a função de mapeamento a cada par (fase de mapeamento). Para cada frase, a função de mapeamento lê sequencialmente cada palavra e gera, para essa palavra, um par $\langle cor, 1\rangle$, onde $cor \in \{verde, branco, azul, rosa\}$. Em outras

palavras, a função de mapeamento gera um conjunto de pares, chamados intermediários, para cada par de entrada.

Ao fim da fase de mapeamento, cada agente paralelo possui um conjunto de pares intermediários. Para que a fase de redução possa ser realizada, os pares referentes à mesma chave (cor) devem ser agrupados, formando pares $\langle cor, [1, 1, \ldots, 1] \rangle$. Essa fase intermediária, que é computacionalmente custosa, é chamada de embaralhamento (*shuffling*). Os pares agrupados então são distribuídos conforme suas chaves (função de particionamento) entre um conjunto de agentes de redução paralelos, os quais aplicam a função de redução a cada par que recebem. A função de redução, nesse exemplo, recebe cada um dos pares agrupados e soma os 1's correspondentes à chave, gerando um par $\langle cor, n \rangle$, em que n é o número de ocorrências de cada cor no texto. O resultado final é então a composição dos pares calculados pelos agentes de redução.

Uma fase adicional de *combinação* pode ser inserida nesse cenário. Nessa fase, cada agente de combinação é associado a um agente de mapeamento. O papel de cada agente de combinação é aplicar a mesma função de redução aplicada pelos agentes de redução sobre os dados do agente de mapeamento associado a ele. Dessa maneira, caso o agente de mapeamento emita k pares $<c, 1>$, para uma determinada cor c, o agente de combinação associado emitirá um par $<c, k>$. Como se pode ver, a vantagem da introdução dessa fase extra é reduzir a quantidade de comunicação na fase de embaralhamento, pois, em vez de se enviar k pares $<c, 1>$, idênticos, para o agente de redução associado à cor c, envia-se somente o par $<c, k>$.

Os seguintes quesitos devem ser levados em consideração para um problema de processamento em particular com MapReduce:

- definição do mapeamento da estrutura de dados original, que contém o dado a ser processado, para um conjunto de pares *chave/valor* de entrada;
- definição das funções que distribuem os pares entres os agentes de mapeamento e de redução, visando balancear a carga de computação entre os processos paralelos;
- definição das funções de mapeamento e redução, que gerarão os pares intermediários e finais, respectivamente.

6.1.2 Componentes HPC Shelf para o MapReduce

Para este estudo de caso, utiliza-se o sistema de computação paralela MapReduce de natureza iterativa originalmente proposto como estudo de caso para avaliação da eficácia do arcabouço SAFe (SILVA, 2016). O arcabouço MapReduce utilizado para esse estudo de caso propõe os seguintes componentes como blocos de construção:

- DataSource, da espécie *fonte de dados*, que representa o repositório da estrutura de dados de entrada para o processamento;
- DataSink, da espécie *fonte de dados*, que representa o repositório onde será armazenada a estrutura de dados resultante do processamento;

- MAPPER, da espécie *computação*, que implementa um agente de mapeamento;
- REDUCER, da espécie *computação*, que implementa um agente de redução;
- SPLITTER, da espécie *conector*, responsável por recuperar a lista de pares de entrada na fonte de dados DATASOURCE (primeira iteração), bem como recuperar os pares produzidos pelos agentes de redução (resultado na iteração anterior) e distribuí-los entre os agentes de mapeamento (próxima iteração) ou redirecioná-los para o depósito de dados de saída (fim do processamento);
- SHUFFLER, da espécie *conector*, responsável por agrupar as chaves intermediárias produzidas pelos agentes de mapeamento e distribuí-las entre os agentes de redução, agrupando valores associados à mesma chave em pares chave/multivalor.

Para que uma aplicação possa utilizar esses componentes em seu *workflow*, as portas de serviços e de ações desses componentes devem ser definidas e associadas aos componentes aplicação e workflow. A aplicação utilizará as portas de serviços para enviar parâmetros de configuração e acompanhar o progresso da execução. O *workflow*, por sua vez, necessita das portas de ações para orquestrar os componentes, visando guiá-los em direção à solução do problema que o sistema de computação paralela se propõe a resolver.

Por fim, é importante lembrar a necessidade da especificação das associações de portas de serviços (ligações de serviços) entre componentes de computação (agentes MAPPER e REDUCER) e conectores (acopladores SPLITTER e SHUFFLER) envolvidos no processamento. Tais portas de serviço implementam iteradores de pares chave/valor ou chave/multivalor, capazes de agrupar pares em pacotes, chamados *chunks*, a fim de otimizar o desempenho da comunicação dos conectores, controlando a sua granularidade. A Figura 23 apresenta o arranjo entre esses componentes que será usado nesse estudo de caso. Uma melhor explicação será apresentada adiante.

6.1.3 Assinaturas Contextuais para os Componentes MapReduce

As assinaturas contextuais utilizadas para a escolha das implementações apropriadas dos componentes MapReduce, feitas pelo mecanismo de resolução implementado pelo sistema de contratos contextuais, são apresentadas a seguir.

Neste estudo de caso, são relacionados somente parâmetros de contexto relativos a propriedades da aplicação em si, não versando sobre parâmetros relacionados às características das plataformas, bem como parâmetros de contexto relacionados a qualidade e custo, pois não dizem respeito a aspectos avaliados neste estudo de caso.

As assinaturas contextuais dos componentes MAPPER e MAPFUNCTION são, respectivamente:

MAPPER [***input_key_type*** = IK : DATA,
input_value_type = IV : DATA,
map_function = MF : MAPFUNCTION [***input_key_type*** = IK,
input_value_type = IV,
intermediary_key_type = TK,
intermediary_key_value = TV]
intermediary_key_type = TK : DATA,
intermediary_key_value = TV : DATA]

MAPFUNCTION [***input_key_type*** = IK : DATA,
input_value_type = IV : DATA,
intermediary_key_type = TK : DATA,
intermediary_key_value = TV : DATA]

Através dos parâmetros de contexto do componente MAPPER, a aplicação pode configurar o tipo dos pares *chave/valor* de entrada e intermediários, bem como o tipo de função de mapeamento, um de seus componentes aninhados. O tipo da função de mapeamento é determinado por um contrato contextual para o componente abstrato MAPFUNCTION, no qual as variáveis de contexto recebem o mesmo valor das variáveis de contexto de MAPPER de mesmo nome.

Por sua vez, as assinaturas contextuais dos componentes REDUCER e REDUCEFUNCTION são, respectivamente:

REDUCER [***intermediary_key_type*** = TK : DATA,
intermediary_value_type = TV : DATA,
reduce_function = RF : REDUCEFUNCTION [***intermediary_key_type*** = TK,
intermediary_value_type = TV,
output_key_type = OK,
output_key_value = OV]
output_key_type = OK : DATA,
output_key_value = OV : DATA]

REDUCEFUNCTION [***intermediary_key_type*** = TK : DATA,
intermediary_value_type = TV : DATA,
output_key_type = OK : DATA,
output_key_value = OV : DATA]

A partir dos parâmetros de contexto de REDUCER, a aplicação pode configurar o tipo de pares *chave/valor* intermediários e de saída, bem como o tipo da função de redução, um de seus componentes aninhados. O tipo da função de redução é determinado por um contrato contextual para o componente abstrato REDUCEFUNCTION, no qual as variáveis de contexto têm o mesmo valor das variáveis de contexto de REDUCER de mesmo nome.

As assinaturas contextuais dos componentes SPLITTER e PARTITIONFUNCTION são, respectivamente:

SPLITTER [***input_key_type*** = TK : DATA,
input_value_type = TV : DATA,
bin_function = BF : PARTITIONFUNCTION [***input_key*** = TK]]

PARTITIONFUNCTION [***input_key*** = K : DATA]

Através dos parâmetros de contexto do componente SPLITTER, a aplicação pode configurar o tipo dos pares *chave/valor* de entrada, bem como o tipo da função de distribuição das chaves de entrada entre os agentes de mapeamento (supondo um arranjo MapReduce típico), chamada de função de partição, um dos componentes aninhados de SPLITTER. O tipo da função de partição é determinado por um contrato contextual para o componente abstrato PARTITIONFUNCTION, no qual o valor da variável de contexto K recebe o mesmo valor de TK, do componente hospedeiro SPLITTER.

Finalmente, a assinatura contextual do componente SHUFFLER é:

SHUFFLER [***intermediary_key_type*** = TK : DATA,
intermediary_value_type = TV : DATA,
partition_function = PF : PARTITIONFUNCTION [***input_key*** = TK]]

Através dos parâmetros de contexto do componente SHUFFLER, a aplicação pode configurar o tipo dos pares *chave/valor* intermediários, bem como o tipo da função de partição para distribuir chaves intermediárias entre os agentes de redução. O tipo dessa função é determinado também por um contrato contextual para PARTITIONFUNCTION, par a qual o valor da variável K recebe o mesmo valor de TK, do componente SHUFFLER.

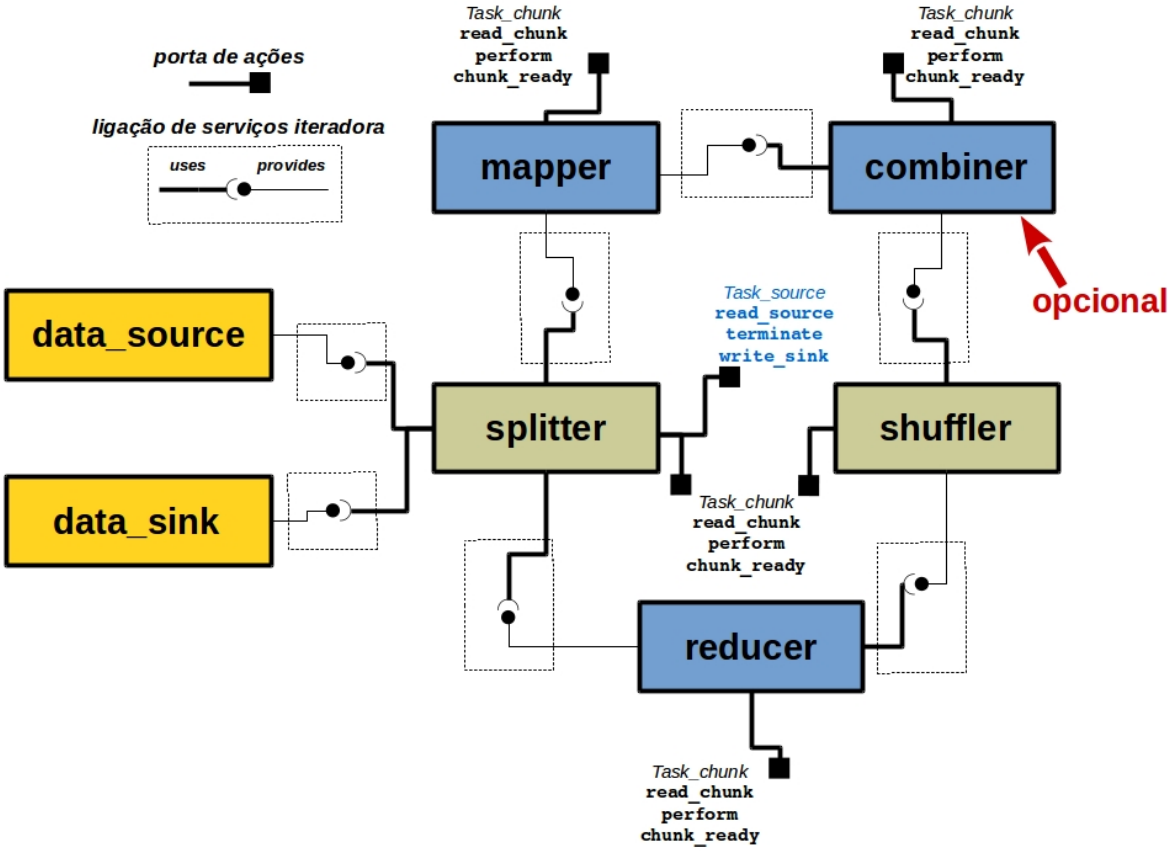
As fontes de dados DATASOURCE e DATASINK não possuem parâmetros de contexto da aplicação.

6.1.4 Arquitetura de um *Workflow* de Processamento MapReduce Iterativo

A Figura 23 mostra a arquitetura do sistema de computação paralela MapReduce interativo típico, composto de instâncias dos componentes elencados anteriormente, utilizado neste estudo de caso. Observa-se que nela existem duas instâncias do componente REDUCER: reducer e combiner.

A comunicação entre os componentes em um *workflow* de processamento MapReduce é feita através de ligações de serviços do tipo ITERATORPORT (retângulos pontilhados), através das quais um componente pode disponibilizar a outro um *stream* de agrupamentos de pares *chave/valor*, doravante chamados de *chunks*. Uma vez que as facetas dos componentes conectores (shuffler e splitter) residem nas mesmas plataformas onde residem seus componentes parceiros via portas de serviços, as ligações ITERATORPORT são diretas. Mais ainda, as implementações dessas ligações são de responsabilidade do desenvolvedor de componentes. Portanto, devem ser escolhidas em tempo de execução de acordo com o mecanismo de resolução de contratos contextuais, dentre implementa-

Figura 23: Arquitetura MapReduce com as portas de ações



Fonte: Elaborado pelo autor.

ções otimizadas de acordo com as características das plataformas virtuais onde estarão instanciadas.

Ainda na Figura 23, é possível notar a porta *Task_chunk*, exposta pelos componentes splitter, reducer, mapper e shuffler, que exporta as seguintes ações:

- **read_chunk**: lê um *chunk*, de tamanho determinado pela granularidade configurada pela aplicação, através da porta usuária apropriada;
- **perform**: efetua a computação sobre o próximo *chunk* da fila de entrada;
- **chunk_ready**: sinaliza que há *chunks* disponíveis na fila de saída, os quais podem ser lidos pelo componente interessado por meio da porta provedora apropriada.

Pode-se ver também, na Figura 23, a porta *Task_source_chunk*, exposta pelo componente splitter, cuja finalidade é definir ações para acessar os repositórios data_source e data_sink, bem como controlar a terminação do algoritmo iterativo. Para suprir tais necessidades, essa porta exporta as seguintes ações:

- **read_source**: lê pares de entrada do repositório data_source, através de uma porta usuária apropriada;
- **terminate**: indica que a iteração atual é a iteração final do processamento (condição de terminação do algoritmo);
- **write_sink**: após o fim do algoritmo, escreve os pares de saída no repositório data_sink, através da porta usuária apropriada.

Para a arquitetura MapReduce iterativa da Figura 23, a ativação das ações e invocações dos serviços são realizadas de acordo com as seguintes etapas:

- O conector splitter inicia a computação com a ativação da ação **read_source**, que faz com que, em sua faceta coletora (*collector*), aquela que executa no mesmo espaço de endereçamento do componente data_source, pares *chave/valor* de entrada sejam lidos através da ligação de serviços entre esses dois componentes, em que splitter é usuário, e agrupados em *chunks* associados a cada agente de mapeamento (mapper). Ao ter a ação **perform** ativada, splitter distribui os *chunks* entre as unidades de sua faceta alimentadora (*feeder*), associadas ao mapper através de uma porta provedora através da qual lê os pares. A ação **chunk_ready**, associada à faceta alimentadora, é então ativada cada vez que um *chunk* está pronto para ser lido por um agente de mapeamento, sinalizando-o a respeito disso;
- O componente mapper lê um *chunk* de splitter quando sua ação **read_chunk** é ativada. Ao ativar sua ação **perform**, mapper executa a função de mapeamento sobre cada par de entrada do *chunk* lido. Pares intermediários calculados são acumulados em *chunks* à medida que são gerados. A ativação da ação **chunk_ready** denota que um novo *chunk* de saída foi gerado. Caso isso ocorra, um *chunk* já pode ser lido pelo próximo componente, que pode ser combiner, implementando, opcionalmente, a fase de combinação, ou shuffler;
- O componente combiner lê um *chunk* de mapper quando sua ação **read_chunk** é

ativada. Quando sua ação **perform** é ativada, a função de redução (combinação) é efetuada sobre os pares *chave/valor* de mesma chave. Sua ação **chunk_ready**, quando ativada, sinaliza que um *chunk* calculado já pode ser lido por shuffler;

- O conector shuffler lê um *chunk* provido pelo componente combiner (ou mapper, caso não haja fase de combinação) ao ter sua ação **read_chunk** ativada. Os dados recebidos representam pares intermediários. Ao ativar sua ação **perform**, shuffler combina os pares que têm a mesma chave, produzindo novos *chunks*, que serão redistribuídos entre as facetas associadas a cada agente de redução. Sua ação **chunk_ready** aponta que um determinado *chunk* está pronto para ser consumido pelo próximo componente, ou seja, reducer;
- O componente reducer lê um *chunk* de shuffler quando sua ação **read_chunk** é ativada. Quando sua ação **perform** é ativada, a função de redução é aplicada sobre os pares *chave/valor* lidos. A ativação de sua ação **chunk_ready** aponta que um *chunk* calculado já pode ser lido pelo próximo componente no fluxo, ou seja, splitter,
- Ao receber dados de reducer, o componente splitter é o responsavel por determinar se o algoritmo deve terminar, quando a ação **terminate** é ativada, ou se uma nova iteração deve acontecer, redistribuindo entre os agentes de mapeamento os pares recebidos do reducer, oriundos da iteração anterior;
- Quando **terminate** é ativada por splitter, sinalizando que o processamento MapReduce acabou, a ação **write_sink** é ativada, fazendo com que os pares de saída, oriundos dos agentes de redução da última iteração, sejam repassados à porta usuária conectada ao repositório data_sink.

6.1.5 Contagem de Palavras em um Repositório de Arquivos Texto

Tendo como base a arquitetura e o fluxo de orquestração descritos anteriormente, o processamento MapReduce pode ser particularizado para executar um processamento de contagem de ocorrências de palavras em arquivos de texto armazenados em um repositório. Assim, a partir de agora, supõe-se que os componentes de computação e conectores concretos elencados anteriormente possuem contratos contextuais representados pelas valorações aos parâmetros de contexto dos respectivos componentes abstratos mostradas na Tabela **??**. Nessa tabela, são descritos o nome do parâmetro de contexto, sua variável associada, a qual componentes está associado, e qual é a sua valoração.

O componente data_source implementa o acesso a uma pasta de um sistema de arquivos, o qual contém arquivos de texto. Através da ligação que o liga a splitter, são fornecidos os pares contendo um valor inteiro e um conteúdo de texto, formado pela concatenação de uma determinada quantidade de linhas de texto lidas dos arquivos que cabem em um *buffer* (a origem da linha é indiferente para esse processamento).

Observe que, na referida tabela, o parâmetro ***bin_function***, do componente abstrato SPLITTER, não recebe valoração, fazendo com que seja escolhida a implementação

Tabela 2: Valoração dos Parâmetros de Contexto dos componentes abstratos do Processamento MapReduce de Contagem de Palavras

Nome do Parâmetro	Variável	Componentes	Valor
input_key_type	IK	mapper splitter	INTEGER
input_value_type	IV	mapper splitter	STRING
map_function	MF	mapper	WORDCOUNTER
intermediate_key_type	TK	mapper mplitter combiner shuffler reducer	STRING
intermediate_value_type	TV	mapper splitter combiner shuffler reducer	INTEGER
reduce_function	RF	combiner reducer	REDUCESUM
output_key_type	OK	reducer	STRING
output_value_type	OV	reducer	INTEGER

default de PARTITIONFUNCTION, a qual terá seu parâmetro ***input_key*** associado ao componente INTEGER. Essa implementação utiliza a função módulo para distribuir as chaves inteiras entre os agentes de mapeamento.

Como se pode ver ainda na tabela, a função de mapeamento é determinada por um componente do tipo WORDCOUNTER. Assim, para cada par de entrada, cada agente de mapeamento emite pares intermediários $\langle w, 1\rangle$, para cada ocorrência da palavra w encontrada no texto, utilizando tal função de mapeamento. Após isso ocorrer, o agente de combinação associado utiliza a função de redução, que é um componente do tipo REDUCESUM, para somar os valores associados a uma mesma chave, emitindo pares intermediários da forma $\langle w, k\rangle$, considerando que k pares intermediários da forma $\langle w, 1\rangle$ foram emitidos pelo agente de mapeamento.

No passo que segue a esse processamento, o componente shuffler distribui os pares intermediários obtidos dos agentes de combinação entre os agentes de redução. Cada agente de redução, por sua vez, soma os valores associados à mesma chave, utilizando a função de redução. Como não é fornecido valor ao parâmetro ***partition_function***, de SHUFFLER, a implementação *default* de PARTITIONFUNCTION também é escolhida, utilizando a função módulo para distribuição de chaves inteiras. O componente splitter encerra a computação ao final da primeira e única iteração após receber os pares de saída $\langle w, n\rangle$. Ao encerrar a computação, ele repassa os dados ao repositório de saída.

6.1.6 Orquestração do Processamento MapReduce

Considerando-se que, no SAFe, o provedor de aplicações especificou o código arquitetural descrito no Apêndice B.2, o qual associa devidamente as portas dos componentes, esse usuário deve então fornecer um código de orquestração em SAFeSWL que realizará o processamento MapReduce. Note que omitimos no código arquitetural, por questões de organização, qualquer menção a componentes certificadores ou táticos, uma vez esses elementos e suas ligações serão devidamente abordados mais adiante. No mesmo apêndice, é mostrado também um exemplo de orquestração referente à arquitetura de processamento

Tabela 3: Relação de identificadores dos componentes MapReduce orquestrados e de suas ações computacionais

Componente	Identificador	Ação	Identificador
splitter	3	read_source	341
		read_chunk	351
		perform	352
		chunk_ready	353
		terminate	342
		write_sink	343
shuffler	4	read_chunk	451
		perform	452
		chunk_ready	453
mapper	5	read_chunk	551
		perform	552
		chunk_ready	553
reducer	6	read_chunk	651
		perform	652
		chunk_ready	653
combiner	10	read_chunk	1051
		perform	1052
		chunk_ready	1053

básico MapReduce descrita na seção 6.1.4. Nesse código de orquestração, introduzido na Tese de Doutorado de Jefferson Carvalho Silva, números inteiros são usados para se referir aos componentes MapReduce e suas ações computacionais. A fim de facilitar a compreensão, a Tabela 3 associa os componentes MapReduce presentes nessa orquestração, bem como suas ações computacionais, aos identificadores inteiros usados no resto deste capítulo, incluindo os códigos SAFeSWL do Apêndice B.2.

6.1.7 *Workflows* Internos dos Componentes MapReduce

De acordo com o protocolo apresentado para a arquitetura MapReduce, é possível notar uma ordem de dependência entre as ações de um mesmo componente, as quais são impostas dentro da lógica de programação de cada componente. Por exemplo, a ação de guarda **read_chunk**, de splitter, só deverá se tornar habilitada para ser ativada após o término da ação **perform** desse mesmo componente. Dessa forma, cada componente MapReduce possui registrado no seu catálogo um *workflow* de habilitação/desabilitação de suas ações, que pode ser composto com o *workflow* de processamento MapReduce para fins de verificação. A seguinte versão simplificada dos *workflows* dos componentes MapReduce é suficiente para que todas as propriedades formais verificadas nesse estudo de caso sejam provadas:

$$W_{shuffler} = W_{mapper} = W_{reducer} = W_{combiner} = \{\top \rightarrow resolve\downarrow, \top \rightarrow deploy\downarrow, \top \rightarrow instantiate\downarrow, \top \rightarrow release\downarrow, \top \rightarrow read\_chunk\downarrow, \top \rightarrow perform\downarrow,\ perform \rightarrow chunk\_ready\downarrow\}$$

$$W_{splitter} = W_{shuffler} \cup \{\top \rightarrow read\_source\downarrow, \top \rightarrow write\_sink\downarrow, perform \rightarrow terminate\downarrow\}$$

6.1.8 Certificação do *Workflow* de Processamento MapReduce

Nesta seção, apresentará-se a certificação do *workflow* da arquitetura básica de computação MapReduce iterativa descrita na Seção 6.1.4. Considere o cenário arquitetural descrito na Figura 24. Nele, primeiro o provedor de aplicações criou, no SAFe, uma ligação de certificação, contendo o *workflow* de processamento MapReduce, o certificador SWC2 e a valoração descrita na Seção 4.3, a qual visa escolher componente concreto SWC2Impl, descrito nessa mesma seção. Em seguida, ele conectou as portas de ações e de serviços entre o *workflow* e o componente certificador abstrato. No passo seguinte, ele solicitou ao SAFe a resolução dos componentes de computação (mapper, combiner e reducer), e para eles foram calculadas plataformas virtuais contendo, respectivamente, 2, 1 e 1 nós de processamento. Note que, com essa resolução, o componente mapper se torna um componente *cluster* com duas unidades (componentes de fina granularidade) e a tradução feita pelo certificador (apresentada mais adiante) levará isso em consideração. Para tanto, antes de realizar a tradução, o certificador altera o *workflow* MapReduce para substituir cada ação a do componente mapper por duas ações a_1 e a_2, cada uma com relação a uma unidade, que devem ser ativadas paralelamente. Mais especificamente, para cada ativação da ação a, caso ela seja síncrona (invoke a), ela será substituída por parallel invoke a_1 invoke a_2 e, caso ela seja assíncrona (start h a), ela será substituída por parallel start h_1 a_1 start h_2 a_2, em que h_1 e h_2 são manipuladores (*handles*) livres.

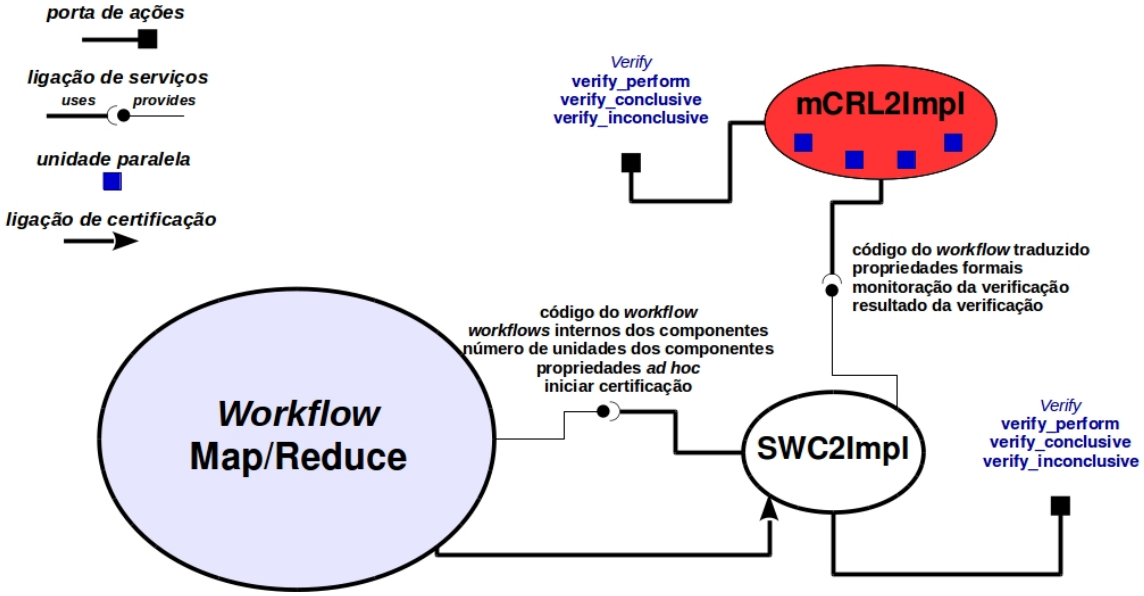
Por fim, o provedor de aplicações solicita ao SAFe que dispare o processo de certificação referente à ligação de certificação, que então instancia e executa automaticamente um sistema de certificação paralela. Durante o processo de certificação, o componente certificador concreto escolhido foi SWC2Impl e como seu componente tático foi escolhido mCRL2Impl, também descrito na Seção 4.3, instanciado com 4 unidades de processamento e 4 núcleos de processamento por unidade de processamento.

De acordo com o contrato contextual de SWC2Impl, as propriedades formais (μ-calculus modal) padrão verificadas por ele sobre o *workflow* de processamento MapReduce são as seguintes:

- **formal_properties**[“MapReduce.swl” , “AD”]: denota ausência de *deadlock*;
- **formal_properties**[“MapReduce.swl” , “ALI”]: significa ausência *loops* infinitos;
- **formal_properties**[“MapReduce.swl” , “VCV1”, “VCV2”, “VCV3”, “VCV4”]: significam propriedades para verificação da consistência das ativações das ações das portas do ciclo de vida dos componentes, o que, no caso desse certificador, é feita utilizando *model checking*.

É importante lembrar que esse componente certificador aceita propriedades *ad hoc*, ou seja, propriedades criadas pelo provedor de aplicações, as quais são repassadas pelo *workflow* ao certificador através da porta de serviços apropriada. Mais adiante, serão definidas tais propriedades para a arquitetura básica de processamento MapReduce

Figura 24: Arquitetura de certificação do *workflow* de processamento MapReduce



Fonte: Elaborado pelo autor.

interativo deste estudo de caso.

6.1.9 Tradução do *Workflow* de Processamento MapReduce

A tradução do código do *workflow*, de SAFeSWL para mCRL2, é comandada pelo componente certificador, que obtém o código do *workflow* MapReduce (Figura 59) e os *workflows* internos dos componentes (Seção 6.1.7), modifica o primeiro para incorporar o comportamento relativo ao componente *cluster* (mapper), os repassa ao algoritmo S2m e deposita o resultado da tradução na porta apropriada do componente tático.

De acordo com o esquema de tradução do Capítulo 4, o código mCRL2 resultante é descrito na Figura 25.

6.1.10 Propriedades Formais *Ad Hoc* para a Arquitetura MapReduce

As propriedades formais *ad hoc* para a arquitetura de processamento MapReduce são todas obrigatórias e se dividem em um grupo de segurança e um grupo de continuidade. O primeiro diz respeito a relações de precedência entre dois componentes distintos ou duas ações deles. Elas são:

- Deve haver uma ativação de splitter.**perform** antes de qualquer ativação de mapper.**read_chunk** (SAM): $[!compute(3,352) * .compute(5,551)]$false
- Entre dois mapper.**read_chunk**, um splitter.**perform** deve ocorrer (MSM): $[\text{true} * .compute(5,551).!compute(3,352) * .compute(5,551)]$false
- Deve haver uma ativação de mapper.**perform** antes de qualquer ativação de combiner.**read_chunk** (MAC): $[!compute(5,552) * .compute(10,1051)]$false
- Entre dois combiner.**read_chunk**, um mapper.**perform** deve ocorrer (CMC): [true $*$

Figura 25: Código mCRL2 traduzido pelo algoritmo S2m para a orquestração da Figura 59

```
act resolve, resolve1_mapper, resolve2_mapper, deploy, deploy1_mapper, deploy2_mapper, instantiate, instantiate1_mapper, instantiate2_mapper,
release, release1_mapper, release2_mapper: Nat; act compute, compute1_mapper, compute2_mapper, w_guard, c_guard, guard, guard1_mapper,
guard2_mapper, w_guard_mapper: Nat # Nat; act pass, obtain, transfer: Nat; act m_repeat, r_repeat, repeat: Nat; act p_break, m_break,
break:Nat; act pass_enable, obtain_enable, transfer_enable: Nat # Nat;

proc Workflow =
resolve(3).deploy(3).instantiate(3).
resolve(4).deploy(4).instantiate(4).
(sum n:Nat.(n == 1 || n == 2) -> pass(n).sum n:Nat.(n == 1 || n == 2) -> pass(n).sum m:Nat.(m == 3 || m == 4) -> obtain(m).
sum m:Nat.(m == 3 || m == 4) ->obtain(m)).
(sum n:Nat.(n == 5 || n == 6) -> pass(n).sum n:Nat.(n == 5 || n == 6) -> pass(n).sum m:Nat.(m == 7 || m == 8) -> obtain(m).
sum m:Nat.(m == 7 || m == 8) ->obtain(m)).
(sum n:Nat.(n == 9 || n == 10) -> pass(n).sum n:Nat.(n == 9 || n == 10) -> pass(n).sum m:Nat.(m == 11 || m == 12) -> obtain(m).
sum m:Nat.(m == 11 || m == 12) ->obtain(m)).
resolve(6).deploy(6).instantiate(6).
resolve(10).deploy(10).instantiate(10).
compute(3,341).compute(3,352).(pass_enable(3,353)+pass_enable(3,342)).
m_repeat(1).obtain(26).
release(3).release(4).
(sum n:Nat.(n == 27 || n == 28) -> pass(n).sum n:Nat.(n == 27 || n == 28) -> pass(n).sum m:Nat.(m == 29 || m == 30) -> obtain(m).
sum m:Nat.(m == 29 || m == 30) ->obtain(m)).
release(6).release(10).Workflow;

proc ManagerRepeat1 = (obtain(25).m_repeat(1) + m_break(1).pass(26)).ManagerRepeat1;

proc Repeat1 = r_repeat(1).(w_guard(3,353).
(sum n:Nat.(n == 13 || n == 14) -> pass(n).sum n:Nat.(n == 13 || n == 14) -> pass(n).sum m:Nat.(m == 15 || m == 16) -> obtain(m).
sum m:Nat.(m == 15 || m == 16) ->obtain(m)).
(sum n:Nat.(n == 17 || n == 18) -> pass(n).sum n:Nat.(n == 17 || n == 18) -> pass(n).sum m:Nat.(m == 19 || m == 20) -> obtain(m).
sum m:Nat.(m == 19 || m == 20) ->obtain(m)).
pass_enable(5,553) +
w_guard_mapper(5,553).compute(10,1051).compute(10,1052).pass_enable(10,1053) +
w_guard(10,1053).compute(4,451).compute(4,452).pass_enable(4,453) +
w_guard(4,453).compute(6,651).compute(6,652).pass_enable(6,653) +
w_guard(6,653).compute(3,351).compute(3,352).(pass_enable(3,353)+pass_enable(3,342)) +
w_guard(3,342).compute(3,343).p_break(1).Repeat1).pass(25).Repeat1;

proc Psplitter_chunk_ready = obtain_enable(3,353).c_guard(3,353).Psplitter_chunk_ready;

proc Psplitter_terminate = obtain_enable(3,342).c_guard(3,342).Psplitter_terminate;

proc Pshuffler_chunk_ready = obtain_enable(4,453).c_guard(4,453).Pshuffler_chunk_ready;

proc Pmapper_chunk_ready = obtain_enable(5,553).
 (sum n:Nat.(n == 21 || n == 22) -> pass(n).sum n:Nat.(n == 21 || n == 22) -> pass(n).sum m:Nat.(m == 23 || m == 24) -> obtain(m).
 sum m:Nat.(m == 23 || m == 24) ->obtain(m)).Pmapper_chunk_ready;

proc Preducer_chunk_ready = obtain_enable(6,653).c_guard(6,653).Preducer_chunk_ready;

proc Pcombiner_chunk_ready = obtain_enable(10,1053).c_guard(10,1053).Pcombiner_chunk_ready;

proc Parallel1mapper_resolve = obtain(1).resolve1_mapper(5).pass(3).Parallel1mapper_resolve;

proc Parallel2mapper_resolve = obtain(2).resolve2_mapper(5).pass(4).Parallel2mapper_resolve;

proc Parallel1mapper_deploy = obtain(5).deploy1_mapper(5).pass(7).Parallel1mapper_deploy;

proc Parallel2mapper_deploy = obtain(6).deploy2_mapper(5).pass(8).Parallel2mapper_deploy;

proc Parallel1mapper_instantiate = obtain(9).instantiate1_mapper(5).pass(11).Parallel1mapper_instantiate;

proc Parallel2mapper_instantiate = obtain(10).instantiate2_mapper(5).pass(12).Parallel2mapper_instantiate;

proc Parallel1mapper_release = obtain(27).release1_mapper(5).pass(29).Parallel1mapper_release;

proc Parallel2mapper_release = obtain(28).release2_mapper(5).pass(30).Parallel2mapper_release;

proc Parallel1mapper_read_chunk = obtain(13).compute1_mapper(5,551).pass(15).Parallel1mapper_read_chunk;

proc Parallel2mapper_read_chunk = obtain(14).compute2_mapper(5,551).pass(16).Parallel2mapper_read_chunk;

proc Parallel1mapper_perform = obtain(17).compute1_mapper(5,552).pass(19).Parallel1mapper_perform;

proc Parallel2mapper_perform = obtain(18).compute2_mapper(5,552).pass(20).Parallel2mapper_perform;

proc Parallel1mapper_chunk_ready = obtain(21).guard1_mapper(5,553).pass(23).Parallel1mapper_chunk_ready;

proc Parallel2mapper_chunk_ready = obtain(22).guard2_mapper(5,553).pass(24).Parallel2mapper_chunk_ready;

proc S = allow({resolve, deploy, instantiate, release, compute, guard, transfer, repeat, break, transfer_enable},
         comm ({resolve1_mapper | resolve2_mapper -> resolve,
                deploy1_mapper | deploy2_mapper -> deploy,
                instantiate1_mapper | instantiate2_mapper -> instantiate,
                release1_mapper | release2_mapper -> release,
                compute1_mapper | compute2_mapper -> compute,
                w_guard | c_guard -> guard,
                guard1_mapper | guard2_mapper | w_guard_mapper-> guard,
                pass | obtain -> transfer,
                m_repeat | r_repeat -> repeat,
                p_break | m_break -> break,
                pass_enable | obtain_enable -> transfer_enable},
         Workflow||ManagerRepeat1||Repeat1||Psplitter_chunk_ready||Psplitter_terminate||Pshuffler_chunk_ready||Pmapper_chunk_ready||
         Preducer_chunk_ready||Pcombiner_chunk_ready||Parallel1mapper_resolve||Parallel2mapper_resolve||Parallel1mapper_deploy||
         Parallel2mapper_deploy||Parallel1mapper_instantiate||Parallel2mapper_instantiate||Parallel1mapper_read_chunk||
         Parallel2mapper_read_chunk||Parallel1mapper_perform||Parallel2mapper_perform||Parallel1mapper_chunk_ready||
         Parallel2mapper_chunk_ready||Parallel1mapper_release||Parallel2mapper_release));
init S;
```

Fonte: Elaborado pelo autor.

$.compute(10, 1051).!compute(5, 552) * .compute(10, 1051)]$false

- Deve haver uma ativação de combiner.**perform** antes de qualquer ativação de shuffler.**read_chunk** (CAS): $[!compute(10, 1052) * .compute(4, 451)]$false
- Entre dois shuffler.**read_chunk**, um combiner.**perform** deve ocorrer (SCS): [true $*$ $.compute(4, 451).!compute(10, 1052) * .compute(4, 451)]$false
- Deve haver uma ativação de shuffler.**perform** antes de qualquer ativação de reducer.**read_chunk** (SAR): $[!compute(4, 452) * .compute(6, 651)]$false
- Entre dois reducer.**read_chunk**, um shuffler.**perform** deve ocorrer (RSR): [true $*$ $.compute(6, 651).!compute(4, 452) * .compute(6, 651)]$false
- Deve haver uma ativação de reducer.**perform** antes de qualquer ativação de splitter.**read_chunk** (RAS): $[!compute(6, 652) * .compute(3, 351)]$false
- Entre dois splitter.**read_chunk**, um reducer.**perform** deve ocorrer (SRS): [true $*$ $.compute(3, 351).!compute(6, 652) * .compute(3, 351)]$false
- Deve haver uma ativação de splitter.**perform** antes de qualquer ativação de splitter.**write_sink** (SAS): $[!compute(3, 352) * .compute(3, 343)]$false

O grupo de continuidade inclui uma variedade mais ampla de propriedades. Elas são:

- LIV1: $\langle$true $* .guard(3, 342)\rangle$true;
- LIV2: $\forall c, a : Nat.\nu X.\mu Y.[compute(c, a)]Y \;\&\&\; [!compute(c, a)]X$;
- LIV3: $[$true$*](\forall c : Nat, a : Nat => \mu Y.([!compute(c, a)]Y \;\&\&\; <$ true $>$ true$))$.

A propriedade LIV1 garante a existência de uma ação particular (splitter.**terminate**) ao longo de qualquer caminho de execução (*trace*) do *workflow*. LIV2 expressa o fato de que uma ação só pode executar durante períodos não consecutivos. Por fim, LIV3 é uma condição de não-inanição (*nonstarvation*), ou seja, para todo estado alcançável é eventualmente possível executar **compute** c a, para qualquer valor possível de c e a.

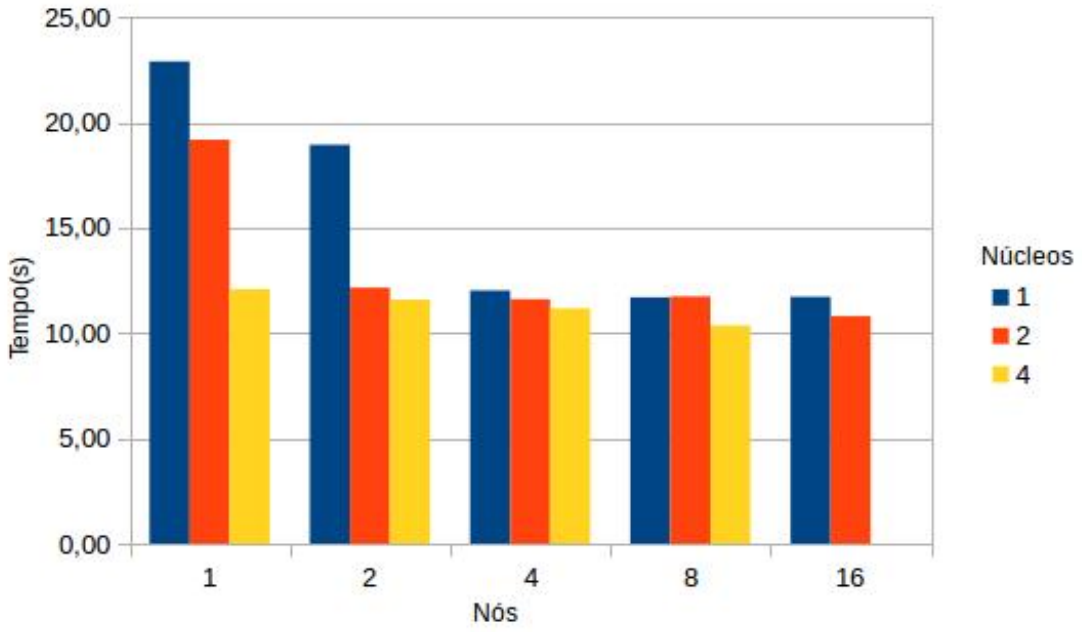
Ao final da certificação, o certificador armazenará o resultado da prova de todas essas propriedades no vetor **formal_properties**, indexado pelo nome das propriedades (SAM, MSM, ..., LIV3).

6.1.11 Resultado da Certificação do *Workflow* de Processamento MapReduce

Quando o processo de certificação acaba, o vetor **formal_properties** possui o resultado da verificação das 20 propriedades formais sobre o código traduzido. Para o estudo de caso em questão, todas as propriedades presentes nesse vetor foram provadas, sendo que cada uma das 4 unidades paralelas do componente tático provou 5, dividindo-as entre 4 *threads*.

A prova de tais propriedades garante, portanto, que o *workflow* não entra em *deadlock* e sempre termina (ausência de *loop* infinito). Garante também que o ciclo de vida dos componentes é bem gerenciado, ou seja, todos primeiro são resolvidos, em seguida implantados, depois instanciados, se tornando prontos para computarem, e, por

Figura 26: Tempos de certificação do *workflow* MapReduce



Fonte: Elaborado pelo autor.

fim, liberados. No tocante às propriedades específicas da aplicação, suas provas implicam que o comportamento desejado pelo provedor de aplicações se verificou no *workflow*.

Implementou-se um protótipo do certificador concreto SWC2Impl e de seu componente tático mCRL2 em C#/MPI tendo em vista validar o arcabouço de certificação, mediante a certificação do *workflow* da arquitetura MapReduce na HPC Shelf. Salienta-se de antemão que todas as implementações oriundas deste trabalho, tanto relativas ao arcabouço de certificação quanto aos componentes certificadores e táticos descritos nos estudos de caso estão disponíveis através de <http://github.com/UFC-MDCC-HPC/HPC-Shelf-Certification>. A Figura 26 retrata os tempos médios experimentais calculados para o cenário arquitetural de certificação apresentado na Seção 6.1.8, através da variação do número de nós de processamento e do número de núcleos de processamento por nó (dois níveis de paralelismo) engajados na execução do componente tático. O tempo sequencial (1 nó e 1 núcleo) foi calculado fora da HPC Shelf, através de um programa *Shell Script* que apenas iterou sequencialmente chamadas de verificação à ferramenta mCRL2 para cada propriedade. Como pode ser visto, o paralelismo foi justificado em todos os casos, levando na maioria dos casos a um tempo de certificação menor que a metade do tempo sequencial. Note que isso pode configurar uma alternativa interessante ao provedor de aplicações, uma vez que a certificação de um *workflow* certificável é pré-requisito para executar a aplicação. Acredita-se, inclusive, que o ganho com a paralelização pode se tornar ainda mais rentável quando um conjunto maior de propriedades é verificado ou quando especificações dos componentes (*workflows* internos, componentes *cluster*, etc.)

mais reais são introduzidas na modelagem.

6.2 Estudo de Caso 2: Montage

O *software* Montage[44] (BERRIMAN et al., 2004) consiste em um conjunto de ferramentas utilizadas em astrofotografia (estudo de imagens de corpos celestiais) para compor mosaicos (conjuntos de imagens organizadas de uma determinada área) a partir de imagens de natureza astronômica. Os mosaicos do Montage preservam fielmente a calibração e a posição das imagens de entrada originais e são capazes de representar áreas do céu que são grandes demais para serem reproduzidas por câmeras astronômicas.

O Montage é bem difundido na comunidade científica, sendo utilizado como estudo de caso em vários projetos de pesquisa relacionados a *workflows* científicos. Entretanto, para os propósitos deste estudo de caso, não se está interessado em provar propriedades sobre *workflows* do Montage, mas em provar propriedades sobre componentes a ele pertencentes que exploram a computação paralela, os quais foram encapsulados em componentes do modelo Hash.

Há três etapas para construir um mosaico com o Montage:

- Reprojetar as imagens de entrada para uma projeção, sistema de coordenadas e escala espacial comum;
- Modelar a radiação de *background* das imagens para as corrigir para uma escala de fluxo e nível de *background* comum, minimizando, assim, as diferenças entre elas;
- Co-adicionar as imagens, reprojetadas e com *background* retificado, em um mosaico final.

O conjunto de ferramenta consiste de um conjunto de mais de 50 componentes independentes e auto-executáveis, os quais podem ser orquestrados tendo em vista obter o resultado final (mosaico). Na Tabela 4, apresentam-se os principais componentes do Montage. Podem-se citar as seguintes características desses componentes:

- Todos são escritos em C e possuem somente uma ação computacional, a qual é implementada no método *main*, porém invocando métodos de bibliotecas contidas no próprio pacote do Montage;
- Como entrada, os componentes podem receber parâmetros de configuração (opcionais) e arquivos. Os arquivos podem representar informações sobre imagens ou até mesmo imagens em formatos específicos;
- Como saída, os componentes podem gerar um único arquivo de informações sobre imagens ou uma série de arquivos de imagens em um formato específico, os quais podem ser utilizados como entrada para outros componentes. Os arquivos de saída gerados pelos componentes são depositados em diretórios do sistema operacional para então serem recuperados por outros;

[44] <http://montage.ipac.caltech.edu/>

Tabela 4: Principais componentes do Montage

Componente	Descrição
Componentes do motor de mosaicos	
mImgtbl	Extrai informações de geometria de um conjunto de cabeçalhos FITS (*Flexible Image Transport System*) [45] e cria uma tabela de metadados a partir delas.
mProject	Reprojeta uma imagem FITS.
mProjExec	Executa mProject para cada imagem da tabela de metadados.
mAdd	Co-adiciona as imagens corrigidas em uma imagem final (mosaico).
Componentes de retificação de background	
mOverlaps	Analisa uma tabela de metadados de imagens para determinar as imagens que se sobrepõem no céu.
mDiff	Executa uma diferença de imagem simples entre um par de imagens sobrepostas. Deve ser aplicado em imagens reprojetadas cujos *pixels* já se alinham exatamente.
mDiffExec	Executa mDiff em todos os pares de sobreposição identificados por mOverlaps.
mFitplane	Ajusta um plano (excluindo *pixels* ultrapassados) a uma imagem. Deve ser usado nas imagens de diferença geradas por mDiff.
mFitExec	Executa mFitplane em todos os pares de imagens que se sobrepõem (detectadas por mOverlaps). Cria uma tabela de parâmetros de diferenças entre cada par de imagens que se sobrepõem.
mConcatFit	Mescla vários arquivos de parâmetros de ajuste de plano (gerados por mFitplane) em um único arquivo.
mBgModel	Componente de modelagem/adequação que usa a tabela de parâmetros de diferenças de imagens para determinar de forma interativa um conjunto de correções que devem ser aplicadas a cada imagem para alcançar uma "melhor" forma global.
mBackground	Remove um plano de fundo de uma única imagem.
mBgExec	Executa mBackground em todas as imagens da tabela de metadados.
Componentes auxiliares	
mArchiveList	Dada uma posição do céu e o nome de um diretório, ele recupera do repositório DSS2 (*Digital Sky Survey* - <http://archive.eso.org/dss/dss>) do servidor IRSA (*Infrared Science Archive* - <http://irsa.ipac.caltech.edu/ibe/>) um arquivo com uma lista de imagens que sobrepõem a posição dada e o guarda no diretório.
mArchiveExec	Recupera do servidor IRSA imagens no formato FITS, relativas a cada imagem listada em um arquivo fornecido e as armazena no diretório corrente.
mJpeg	Gera um arquivo JPEG para o mosaico.

- As ações computacionais de alguns componentes consistem em simular a execução paralela de instâncias da ação computacional de um outro componente associado sobre entradas distintas. Por exemplo, para o componente mAdd, o qual co-adiciona (sobrepõe) imagens reprojetadas contidas em uma lista de entrada para formar uma área do mosaico, existe o componente mAddExec, que simula a execução paralela de instâncias de mAdd, em que cada uma co-adiciona imagens contidas em uma lista de entrada diferente, gerando assim uma área distinta do mosaico. Os componentes paralelos utilizam a biblioteca MPI e é neles o interesse de certificação nesse estudo de caso.

Existe uma relação de dependência entre os componentes do estudo de caso anterior (MapReduce), uma vez que existe uma relação de dependência entre as fases (*splitting*, *mapping*, *shuffling* e *reducing*) em que cada um executa. Mais ainda, as fronteiras (entradas e saídas) dos componentes MapReduce não são exatas, uma vez que elas dependem das escolhas das funções de mapeamento, redução e particionamento. Diferentes desses, os componentes do Montage são independentes, podendo ser invocados há qualquer momento desde que se forneçam as entradas adequadas, e possuem fronteiras bem definidas, uma vez que trabalham sempre com os mesmos tipos de arquivos e parâmetros de configuração. Esse fato permitiu que os componentes do Montage fossem portados diretamente para a HPC Shelf (sem reescrita), sendo apenas encapsulados como componentes HPC Shelf e criadas as devidas portas de serviços e de ações. Essa implementação foi usada como estudo de caso para o SAFe (SILVA; CARVALHO JUNIOR, 2016).

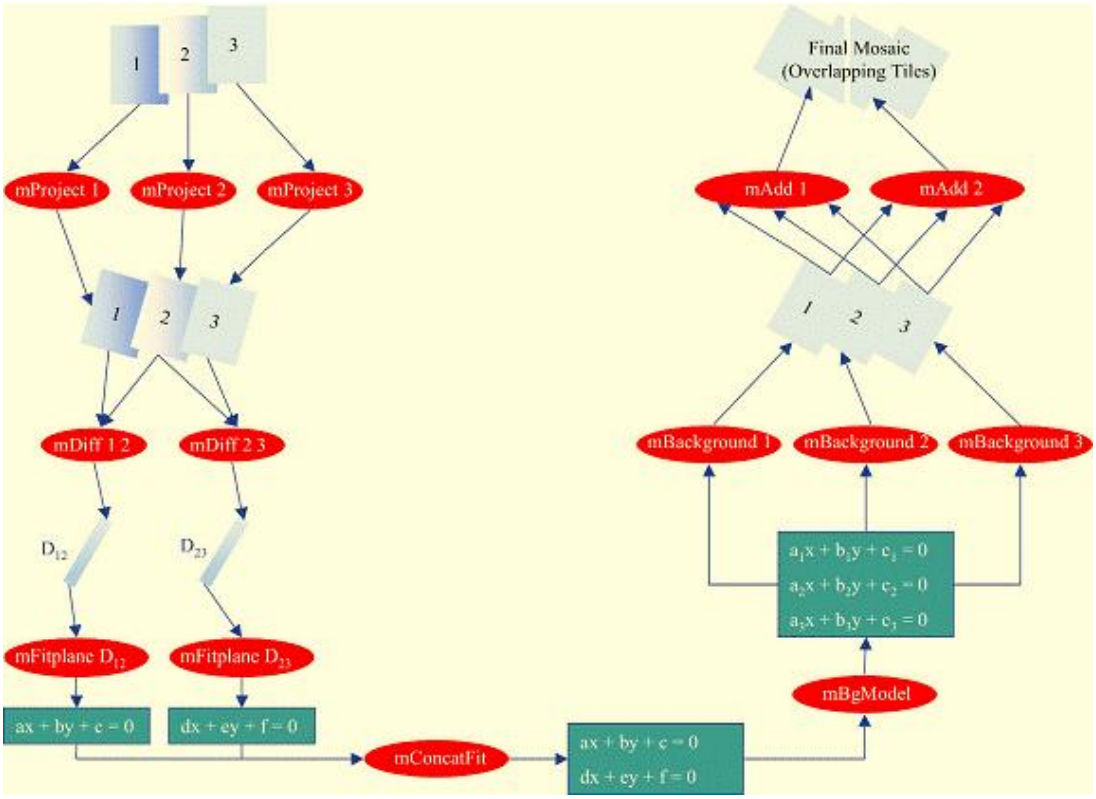
6.2.1 *Workflows* do Montage

Orquestrações de componentes Montage são formadas por composições adequadas dos componentes disponibilizados na ferramenta com a finalidade de obter mosaicos com os efeitos pretendidos. As orquestrações que regem as ativações dos componentes, entretanto, não possuem uma linguagem específica de domínio, como a SAFeSWL. Assim, para se aplicar um fluxo de execução pretendido (sequencial, condicional, paralelo, repetido, etc.) tem-se que utilizar um *script* externo, que pode ser um simples programa *Shell Script* que é interpretado pelo *bash* do sistema operacional ou um *script* complexo gerado a partir de um sistema de gerenciamento de *workflow* como o Pegasus[46] ou o próprio SAFe.

A Figura 27 mostra um exemplo de *workflow* que visa ilustrar a ativação paralela dos componentes do Montage. Nela, aparecem os componentes mProject, mDiff, mFitPlane, mConcatFit, mBgModel, mBackground e mAdd (elipses). O *workflow* recebe como entrada imagens denotadas por 1, 2 e 3 e tem como saída o mosaico final. O único passo em que os componentes não podem executar paralelamente é na execução de mBgModel, o qual implementa a computação do modelo de *background* do mosaico. Dentre os

[46] <http://confluence.pegasus.isi.edu/display/pegasus/Montage>

Figura 27: Exemplo de *workflow* utilizando os componentes do Montage, ressaltando a paralelização dos componentes. Fonte:
<http://montage.ipac.caltech.edu/docs/grid.html>



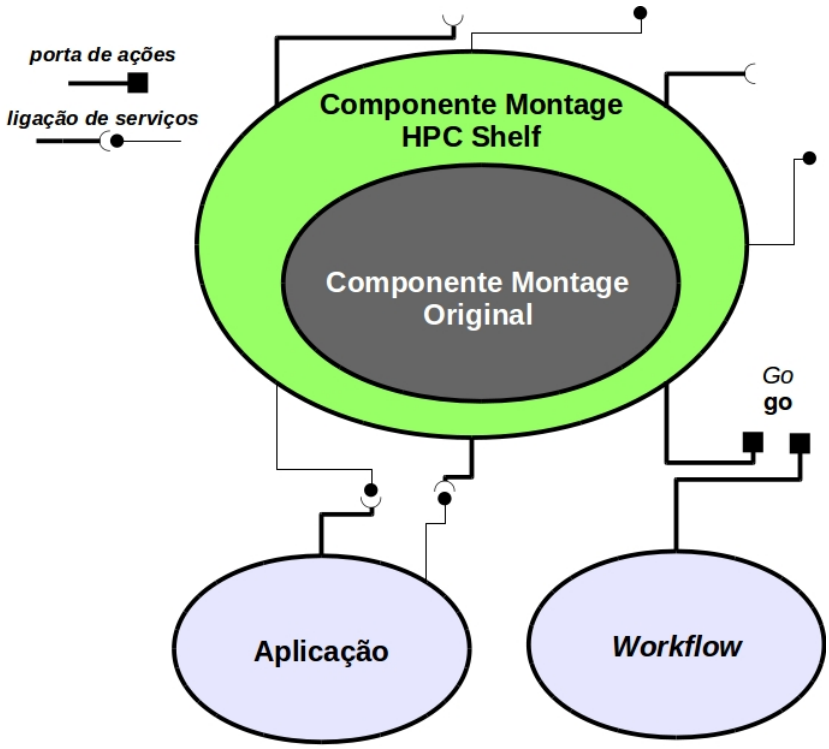
Fonte: Elaborado pelo autor.

componentes desse *workflow*, os únicos componentes que possuem versões paralelas (MPI) são mProject (mProjExec), mAdd (mAddExec) e mDiff (mDiffExec). Dessa forma, em vez de ativarem-se diversas instâncias desses componentes paralelamente no *workflow*, poderia-se disparar somente uma instância do componente paralelo associado, e ele simularia a ativação paralela de instâncias da ação computacional do componente original, gerando o mesmo efeito da paralelização original. Contudo, essa segunda alternativa pode revelar um desempenho superior, uma vez que toda a paralelização seria feita dentro do ambiente de execução do MPI (através de processos MPI), não necessitando criar um processo de sistema operacional para cada instância do componente original e também pelo fato de que o componente paralelo realiza somente uma vez cálculos comuns que seriam feitos em cada ativação paralela de instâncias do componente original. Por fim, os retângulos na figura representam os cálculos realizados internamente pelos componentes.

6.2.2 Componentes HPC Shelf para o Montage

A Figura 28 apresenta a configuração das portas de um componente genérico do Montage, encapsulado através de um componente HPC Shelf. Por simplicidade, doravante chamará-se um componente HPC Shelf que encapsula um componente do Montage somente de

Figura 28: Portas de serviços e de ações dos componentes HPC Shelf que encapsulam componentes do Montage



Fonte: Elaborado pelo autor.

"componente Montage". Cada componente Montage se comunica com a aplicação através portas de serviços para receber parâmetros, fornecer estatísticas de execução e o mosaico final. Para se comunicar com outros componentes Montage, utilizam portas de serviços para receber e fornecer arquivos. Por fim, para serem orquestrados pelo *workflow*, utilizam uma porta de ações chamada *Go*, que exporta a ação **go**, a qual representa sua ação computacional.

A implementação feita na HPC Shelf, entretanto, criou um componente da espécie fonte de dados para representar cada repositório em que os componentes depositam as imagens de saída. Assim, quando um componente de computação produz arquivos, ele os repassa a um componente fonte de dados, que então podem ser recuperados por outros.

Uma observação que deve ser feita é que os componentes Montage possuem somente parâmetros de contexto de plataforma. Novamente, dessa forma, cada componente possui somente uma implementação possível.

6.2.3 O *Workflow* Plêiades

Para ilustrar a orquestração dos componentes do Montage na HPC Shelf, apresenta-se o *workflow* Plêiades, que gera um mosaico para a constelação *Plêiades*[47], o qual foi implementado no SAFe e serviu como estudo de caso para ele. Salienta-se que esse exemplo serve

[47] <http://montage.ipac.caltech.edu/docs/pleiades_tutorial.html>

somente para contextualizar o leitor sobre a utilização dos componentes do Montage, através de uma orquestração, e que, como será visto adiante, o interesse nesse estudo de caso consistirá em certificar somente os componentes paralelos do Montage, e essa certificação independe do *workflow* em que os componentes são empregados. O *workflow* Plêiades é formado pela orquestração dos componentes mArchiveList, mArchiveExec, mImgTbl, mProjExec, mAdd e mJpeg. Observa-se que o único componente paralelo nessa orquestração é o mProjExec.

Para cada imagem que sobrepõe o ponto central da constelação Plêiades (56.5 23.75), o *workflow* mantém três versões suas em diferentes faixas de cor (vermelho, infravermelho e azul), obtidas a partir do repositório DSS2. As imagens de uma mesma faixa de cor são armazenadas, respectivamente, nos componentes fontes de dados dss2rDir, dss2irDir e dss2bDir. Cada faixa de cor é processada sequencialmente por um sub-*workflow* idêntico, disparado em paralelo. A Figura 29 ilustra a lógica de orquestração dos componentes de computação do *workflow* Plêiades. O código de orquestração SAFeSWL do *workflow* Plêiades, entretanto, é dispensável para os propósitos dessa seção. As seguintes observações sobre esse *workflow* são importantes. Primeiro, todas as trocas de informações entre os componentes são realizadas através de portas de serviços, a menos no caso em que um componente acessa um repositório remoto mantido por um componente fonte de dados. Nesse caso, a comunicação se dá através do uso do protocolo SSH. Segundo, todos os componentes são ativados sincronamente através da tarefa invoke. Terceiro, em cada sub-*workflow*, o componente mImgtbl é ativado duas vezes, uma vez para trabalhar sobre imagens cruas e outra vez para trabalhar sobre imagens reprojetadas. Ele, contudo, obtém da aplicação a informação de qual grupo de imagens trabalhar em cada vez. Quarto, o usuário especialista, no momento da execução da aplicação, deve fornecer um arquivo de configuração no formato HDR[48], que contém informações determinadas por ele e que influenciam as transformações aplicadas às imagens por certos componentes. Exemplos dessas informações são qual rotação deve ser aplicada às imagens, para qual sistema de coordenadas celestiais elas devem ser migradas e parâmetros de reprojeção. Quinto, ao fim dos sub-*workflows* paralelos, o componente mAdd possui três versões do mosaico, cada uma em uma faixa de cor. O componente mJpeg então as recupera dele, gera uma imagem única e a repassa à aplicação.

Operacionalmente, cada sub-*workflow* realiza em sequência os seguintes passos:

1. Ativa o componente mArchiveList, que recupera do repositório DSS2 o arquivo que lista as imagens da respectiva faixa de cor que sobrepõem o ponto central da constelação Plêiades;
2. Ativa o componente mArchiveExec, que recupera do componente mArchiveList o arquivo que lista as imagens, recupera do servidor as imagens no formato FITS contidas nesse arquivo (chamadas de imagens cruas), recupera da fonte de dados o endereço

[48] <http://montage.ipac.caltech.edu/docs/headers.html>

do repositório remoto de onde deve depositar as imagens cruas e as deposita no endereço fornecido;

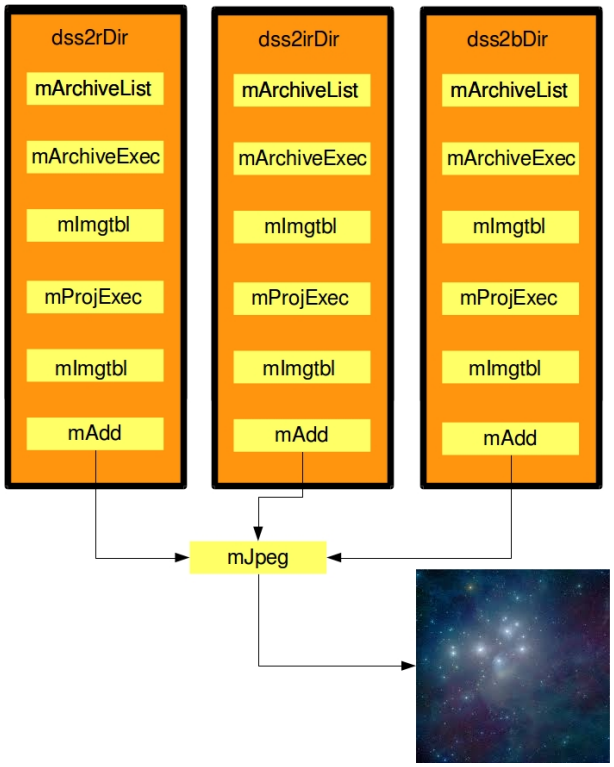
3. Ativa mImgtbl, que recupera da aplicação a informação de que deve recuperar do componente fonte de dados o endereço remoto das imagens cruas mantidas por ele. mImgtbl acessa esse repositório e gera a tabela de metadados para essas imagens;
4. Ativa mProjExec, que recupera de mImgtbl a tabela de metadados das imagens cruas, recupera da aplicação o arquivo de configuração para obter detalhes de como as imagens FITS cruas devem ser reprojetadas e recupera da fonte de dados o endereço remoto das imagens cruas mantidas por ele e um endereço de um novo repositório onde depositará as imagens reprojetadas. De posse dessas informações, mProjExec acessa o repositório das imagens cruas e reprojeta cada uma paralelamente, de acordo com as informações de reprojeção do arquivo de configuração, e as deposita no novo repositório. Por fim, mProjExec repassa à aplicação um arquivo com estatísticas da reprojeção das imagens, com informações como o tempo de reprojeção e possíveis erros durante o processamento;
5. Ativa novamente mImgtbl, que obtém da aplicação a informação de que deve recuperar do componente fonte de dados agora o endereço remoto das imagens reprojetadas. mImgtbl então acessa esse repositório e gera a tabela de metadados relativa a essas imagens;
6. Ativa mAdd, que recupera de mImgtbl a tabela de metadados das imagens reprojetadas, recupera da fonte de dados o endereço remoto do repositório das imagens reprojetadas e recupera da aplicação o arquivo de configuração para obter informações de como as imagens devem ser co-adicionadas. De posse desses dados, acessa o referido repositório e co-adiciona as imagens formando um mosaico na faixa de cor sendo tratada pelo sub-*workflow*.

6.2.4 Certificação de Componentes de Computação do Montage

A maior carga de processamento do Montage recai sobre os componentes paralelos. Isso se deve tanto pelo fato do paralelismo em si, quanto pelo fato dos componentes sequenciais associados a eles efetuarem as ações computacionais mais críticas. Dessa forma, os esforços deste trabalho relativos a este estudo de caso se concentraram na certificação desses componentes. Os únicos componentes paralelos disponíveis são mAddExec, mBgExec, mDiffExec, mFitExec e mProjExec. Como todo componente computação da HPC Shelf, cada componente paralelo do Montage utiliza tantas unidades paralelas quantas forem as unidades de processamento da plataforma alvo. Assim, cada componente paralelo do Montage dispara nas suas unidades processos MPI referentes ao seu único programa (SPMD), que tem o mesmo nome do componente.

Para certificar esses e quaisquer outros componentes implementados em C/MPI, foi criado por um certificador de componentes o componente certificador concreto C4CMPIImpl.

Figura 29: Os três sub-*workflows* do *workflow* Plêiades. Ao final, as imagens de mesma faixa de cor são combinadas pelo componente mJpeg



Fonte: (SILVA, 2016).

C4CMPIImpl orquestra dois componentes táticos, um de *model checking* (ISP, descrito na Seção 5.2) e um de verificação dedutiva (ParTypes, descrito mais adiante). O componente certificador concreto possui o seguinte contrato contextual:

C4CMPIImpl : C4 [***programming_language*** = C,
separation_logic = NoSeparationLogic,
floating_point_operations = NoFloatingPointOperations,
existential_quantifier = NoExistentialQuantifier,
interactive_proof = NoInteractiveProof,
message_passing_interface_verif = MPIVerif,
ad_hoc_properties = AdHocProperties,
max_verification_time = 30]

Através desse contrato, pode-se afirmar que C4CMPIImpl verifica programas C/MPI, aceita propriedades *ad hoc* e seu tempo de verificação máximo inicial (experimental) é 30 segundos.

O componente tático abstrato ParTypes simplesmente estende o componente tático abstrato Tactical, descrito na Seção 4.3, sem adicionar parâmetros. Como componente tático concreto para ele, cita-se ParTypesImpl, descrito a seguir:

ParTypesImpl : ParTypes [***message_passing_interface*** = MPICH2, ***number_cores*** = 4]

As valorações fornecidas por C4CMPIImpl para ISP e ParTypes são, respec-

Figura 30: Código de orquestração TCOL do componente certificador C4CMPIImpl

```
<sequence>
        <parallel>
                <sequence>
                        <invoke action="resolve" id_port="life-cycle-ISP" />
                        <invoke action="deploy" id_port="life-cycle-ISP" />
                        <invoke action="instantiate" id_port="life-cycle-ISP" />
                </sequence>
                <sequence>
                        <invoke action="resolve" id_port="life-cycle-ParTypes" />
                        <invoke action="deploy" id_port="life-cycle-ParTypes" />
                        <invoke action="instantiate" id_port="life-cycle-ParTypes" />
                </sequence>
        </parallel>
        <parallel>
                <invoke action="verify_perform" id_port="verify-ISP" />
                <invoke action="verify_perform" id_port="verify-ParTypes" />
        </parallel>
        <invoke action="release" id_port="life-cycle-ISP" />
        <invoke action="release" id_port="life-cycle-ParTypes" />
</sequence>
```

Fonte: Elaborado pelo autor.

tivamente, as seguintes:

$$\text{ISP } [\textbf{\textit{message\_passing\_interface}} = \text{MPI}]$$

$$\text{ParTypes } [\textbf{\textit{message\_passing\_interface}} = \text{MPI}]$$

Claramente, ISPImpl, da Seção 5.2, e ParTypesImpl são, respectivamente, candidatos a essas valorações.

A orquestração executada por C4CMPIImpl é regida pelo fragmento TCOL da Figura 30. Como se pode ver nessa orquestração, ela primeiro gerencia paralelamente o ciclo de vida dos componentes táticos através das ações correspondentes. Em seguida, ativa suas ações de verificação em paralelo. Isso fará, paralelamente, com que o tático ISP verifique um conjunto fixo de propriedades sobre os programas atribuídos a ele e que o tático ParTypes confronte cada programa a ele atribuído com um protocolo de alto nível que reflete as operações de comunicação do MPI executadas dentro do programa. Observa-se novamente que o componente ParTypes é assercional não puro. Ou seja, um programa que ele verifica deve ser anotado com asserções de Hoare na sintaxe do seu *frontend* (VCC) e guiarão o componente tático na verificação do protocolo correspondente, que é externo ao programa (não anotado). Os protocolos ParTypes são recebidos pelo componente certificador como propriedades *ad hoc* e são considerados obrigatórios por ele. Por fim, essa orquestração libera esses componentes das respectivas plataformas virtuais.

Para tornar os componentes paralelos do Montage certificáveis, o desenvolvedor desses componentes criou no Core entre cada um deles e o componente certificador C4 uma ligação de certificação, para a qual foi fornecida a seguinte valoração:

$$\begin{aligned}\text{C4 } [&\textbf{\textit{programming\_language}} = \text{C},\\ &\textbf{\textit{ad\_hoc\_properties}} = \text{AdHocProperties},\\ &\textbf{\textit{message\_passing\_interface\_verif}} = \text{MPIVerif}]\end{aligned}$$

Essa valoração afirma que ele quer os componentes do Montage sejam certificados através de um certificador que verifique programas C/MPI e que aceite propriedades

Figura 31: Código SAFeSWL de orquestração da aplicação de certificação dos componentes paralelos do Montage

```
<parallel>
        <sequence>
                <invoke action="resolve" id_port="life-cycle-mAddExec" />
                <invoke action="certify" id_port="life-cycle-mAddExec" />
        </sequence>
        <sequence>
                <invoke action="resolve" id_port="life-cycle-mBgExec" />
                <invoke action="certify" id_port="life-cycle-mBgExec" />
        </sequence>
        <sequence>
                <invoke action="resolve" id_port="life-cycle-mDiffExec" />
                <invoke action="certify" id_port="life-cycle-mDiffExec" />
        </sequence>
        <sequence>
                <invoke action="resolve" id_port="life-cycle-mFitExec" />
                <invoke action="certify" id_port="life-cycle-mFitExec" />
        </sequence>
        <sequence>
                <invoke action="resolve" id_port="life-cycle-mProjExec" />
                <invoke action="certify" id_port="life-cycle-mProjExec" />
        </sequence>
</parallel>
```

Fonte: Elaborado pelo autor.

externas (*ad hoc*). Certamente, C4CMPIImpl é um candidato a essa valoração. Por fim, ele configurou o único programa de cada componente paralelo do Montage para ser verificado pelos dois componentes táticos de C4CMPIImpl, ISP e ParTypes, e gravou em cada componente, como uma propriedade *ad hoc*, um protocolo ParTypes a ser confrontado com a implementação do seu programa.

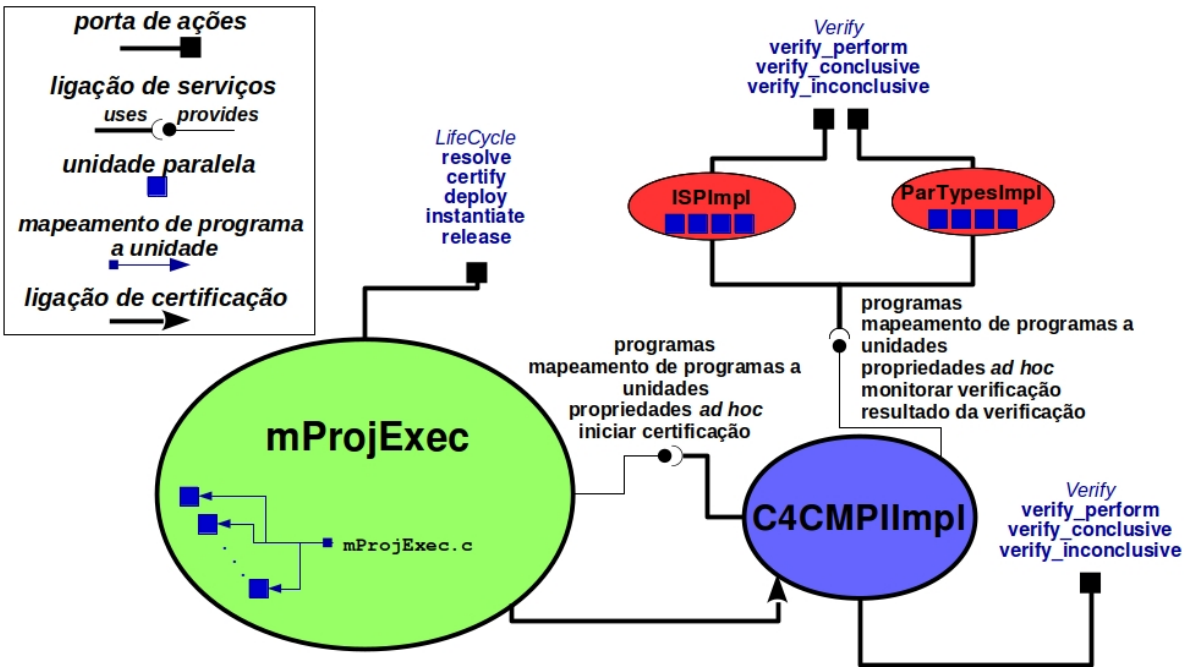
Visando certificar em lote os componentes paralelos do Montage que criou, o desenvolvedor acessou uma *aplicação de certificação* existente somente com propósito de auxiliar o desenvolvedor em tarefas de certificação prévia dos componentes que desenvolve. Através dessa aplicação, ele criou um sistema de computação paralela fictício contendo todos os componentes a serem certificados, incluindo o *workflow* associado, o qual é definido pelo fragmento SAFeSWL contido na Figura 31, bem como ligações de serviços entre os componentes e C4 para troca de informação. Uma vez que os componentes Montage possuem somente parâmetros de contexto de plataforma, a resolução dos componentes sempre trará a implementação registrada pelo desenvolvedor.

Como se pode ver na orquestração da Figura 31, ela dispara em paralelo sequências de ativações síncronas de ações. Cada sequência é associada a um componente e primeiro ativa sua ação **resolve** para depois ativar **certify**.

O desenvolvedor então solicita a execução da aplicação de certificação no SAFe, o qual então escolhe o certificador C4CMPIImpl como componente concreto para C4 e cria/instancia automaticamente um sistema de certificação paralela ele. No momento da ativação da ação **certify** de cada componente, esse componente se junta ao sistema de certificação paralela de C4CMPIImpl para ser certificado.

O cenário arquitetural de certificação com relação a cada componente paralelo do Montage é similar e particularizou-se, por propósitos didáticos, para o componente mProjExec, na Figura 32. Para todos os componentes, plataformas virtuais contendo 20 nós de processamento foram escolhidas. Para os componentes táticos de C4CMPIImpl

Figura 32: Arquitetura de certificação do componente mProjExec do Montage



Fonte: Elaborado pelo autor.

Figura 33: Protocolo ParTypes a ser confrontado com o programa `mProjExec.c`, que é anotado com asserções na sintaxe do VCC

```
0  protocol mProjExec {
1
2      allreduce sum integer[1]
3      allreduce sum integer[1]
4      allreduce sum integer[1]
5      allreduce sum integer[1]
6      allreduce sum integer[1]
7      allreduce sum integer[1]
8      allreduce sum integer[1]
9      allreduce sum integer[1]
10
11 }
```

Fonte: Elaborado pelo autor.

foram escolhidos ISPImpl e ParTypesImpl, respectivamente, ambos com 4 nós de processamento. Como protocolo ParTypes a ser confrontado com o programa de mProjExec (`mProjExec.c`), considere o protocolo da Figura 33, que foi fornecido no próprio componente. Esse protocolo afirma que `mProjExec.c` executa uma sequência de oito operações **MPI_Allreduce**, em que cada uma aplica uma redução (no caso, soma) de números inteiros oriundos de todos os processos e distribui o resultado de volta para todos os processos.

6.2.5 Resultado da Certificação dos Componentes Paralelos do Montage

Quando a orquestração do sistema de certificação paralela de C4CMPIImpl com relação a um determinado componente termina, o vetor **formal_properties** mantido pelo componente certificador possui o resultado da verificação de todas as propriedades sobre esse

componente, dizendo se ele foi certificado, no caso em que no mínimo todas as propriedades obrigatórias do certificador foram provadas. Todos os componentes exceto mBgExec foram certificados. A não certificação de mBgExec se deu pelo fato de que, durante sua certificação, o tático ISPImpl detectou um possível entrelaçamento que pode levar à recepção de um *buffer* vazio em uma operação **MPI_Allreduce**.

Como exemplo, observe a configuração do vetor **formal_properties** com relação ao componente mProjExec:

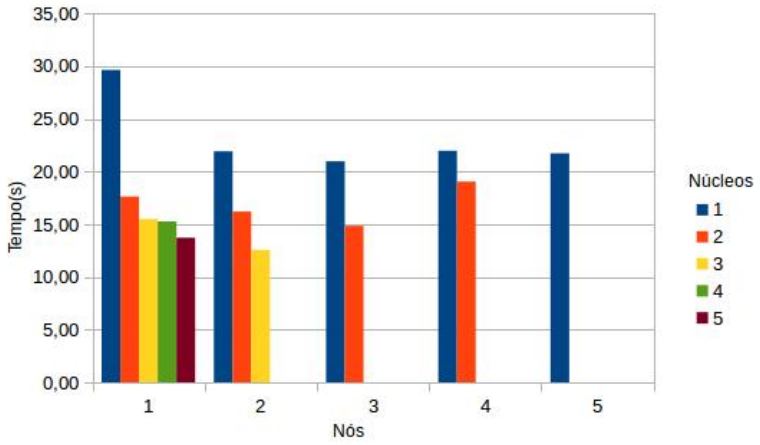
- **formal_properties**[“mProjExec.c” , “no deadlock”] = true
- **formal_properties**[“mProjExec.c” , “no MPI object leaks”] = true
- **formal_properties**[“mProjExec.c” , “no communication races”] = true
- **formal_properties**[“mProjExec.c” , “no irrelevant barriers”] = true
- **formal_properties**[“mProjExec.c” , “protocol mProjExec”] = true

Como se pode ver, todas as propriedades foram provadas. Dessa forma, o componente mProjExec foi considerado certificado, uma vez que todas essas propriedades são obrigatórias. A prova dessas propriedades garante que mProjExec.c não entra em *deadlock*, não tem falhas de acesso a objetos MPI, não apresenta condições de corrida em comunicações, não utiliza barreiras MPI de forma irrelevante e que as operações de comunicação que realiza condizem com o protocolo dado.

Desenvolveu-se um protótipo do certificador C4CMPIImpl e de seus componentes táticos (ISPImpl e ParTypesImpl), a fim de avaliar o arcabouço de certificação mediante a certificação de componentes de computação (componentes paralelos do *software* Montage)[49]. A Figura 34 mostra os tempos experimentais médios calculados para a execução da aplicação de certificação, através da variação do número de nós de processamento e do número de núcleos de processamento por nó de processamento engajados na execução dos componentes táticos. Mais uma vez, o tempo sequencial (1 nó e 1 núcleo) foi calculado chamando diretamente as ferramentas de verificação para verificar os programas (fora da HPC Shelf), sequencialmente, através de um programa *Shell Script*. Como se pode notar, novamente o paralelismo foi justificado em todos os casos, levando em alguns casos a um tempo de certificação menor que a metade do tempo sequencial. Outra vez, avaliou-se que esse ganho pode ser uma alternativa viável ao provedor de aplicações ou ao desenvolvedor de componentes, já que a certificação de um componente certificável é pré-requisito para sua execução na aplicação. Acredita-se que esse ganho pode ser ainda mais evidente quando os componentes possuem mais programas a verificar ou quando o certificador pode orquestrar mais componentes táticos.

[49] <http://github.com/UFC-MDCC-HPC/HPC-Shelf-Certification>

Figura 34: Tempos de execução da aplicação de certificação dos componentes paralelos do Montage



Fonte: Elaborado pelo autor.

6.3 Conclusões Finais

Nesse capítulo, apresentaram-se dois estudos de caso para a arquitetura dos componentes certificadores. O primeiro estudo de caso é relativo à certificação de um *workflow* típico da arquitetura de processamento MapReduce. Por sua vez, o segundo estudo de caso trata da certificação dos componentes de computação paralelos de um arcabouço Montage sobre a plataforma HPC Shelf.

Podem-se elencar três contribuições principais deste capítulo, comentadas sucintamente adiante:

1. a capacidade da modelagem do componente SWC2 em representar aspectos relevantes de um *workflow* real;
2. a capacidade do componente C4 orquestrar componentes táticos com naturezas distintas; e
3. o desempenho do arcabouço de certificação.

Em relação ao primeiro, pôde-se perceber que a modelagem foi suficientemente expressiva para capturar diversos aspectos relevantes do *workflow* MapReduce além da orquestração em si, como, por exemplo, os *workflows* internos dos componentes e os componentes *cluster*. Mais ainda, ela foi expressiva também para permitir a prova de todas as 20 propriedades elencadas neste trabalho, divididas igualmente entre 4 unidades do componente tático disparadas paralelamente.

Com relação ao segundo, a orquestração de componentes táticos que encapsulam ferramentas de verificação com características distintas permitiu verificar em lote um conjunto pequeno, porém relevante, de propriedades sobre cada componente paralelo do Montage. Seguindo o que preza a filosofia da Verificação como Serviço (VaaS), pôde-se atestar a viabilidade de um mecanismo de certificação voltado à Computação de Alto Desempenho que claramente diminui a complexidade do serviço de verificação.

Por fim, com relação ao terceiro, observa-se inicialmente que tarefas de verificação disparadas por ferramentas de verificação são geralmente custosas e que os componentes táticos são componentes de grossa granularidade. Note que devido a esses fatos, quão maior o conjunto de propriedades a ser verificado e quão maior o conjunto de componentes táticos orquestrados, mais se tenderia a amenizar a percepção dos impactos (sobrecarga) do gerenciamento da certificação pelo arcabouço de certificação no tempo total. Assim sendo, intencionalmente adotaram-se conjuntos relativamente pequenos de propriedades formais a serem verificadas e componentes táticos orquestrados, crendo ser a alternativa mais viável para medir empiricamente o desempenho do arcabouço de certificação. Como se pôde ver, os testes mostraram ganhos razoáveis de performance através da certificação paralela.

7 Conclusões e Trabalhos Futuros

Neste capítulo, são delineadas as contribuições deste trabalho, as conclusões acerca delas e as perspectivas no que tange a trabalhos futuros.

7.1 Contribuições e Conclusões Relacionadas

A HPC Shelf, a qual constitui uma plataforma de nuvens computacionais voltada à construção de aplicações de cunho científico e que demandam pelo uso de computação paralela em larga escala, é um projeto em andamento do Grupo de Computação de Alto Desempenho do Programa de Pós-Graduação em Ciência da Computação da Universidade Federal do Ceará. Inserida nesse contexto, a Tese de Doutorado proposta traz, como sua principal contribuição, um arcabouço de verificação como serviço em nuvem (VaaS) para a certificação de componentes paralelos na HPC Shelf, os quais alcançam o *status* de certificados ao terem propriedades formais de interesse das aplicações que os necessitam provadas sobre suas implementações.

7.1.1 Arcabouço de Certificação

O arcabouço de certificação proposto, descrito no Capítulo 2, surge como um mecanismo para atestar a confiabilidade da implementação de componentes, tendo um papel importante na nuvem, por questões de segurança, tanto com relação a um comportamento malicioso que pode ser assumido por códigos desenvolvidos por terceiros (desenvolvedores de componentes), quanto no que tange a um comportamento anômalo ou errôneo, motivado por erros de programação. Tais erros se tornam potencialmente mais comuns no contexto das aplicações da HPC Shelf, por serem compostas por componentes paralelos. Eles se tornam ainda mais prejudiciais, uma vez que podem causar a invalidação dos estudos científicos, devido à produção de resultados errôneos ou cuja confiabilidade não pode ser comprovada, além de desperdiçar tempo precioso, uma vez que se tratam comumente de aplicações de longa duração.

Uma observação inicial que se faz é que o arcabouço de certificação proposto nesta Tese não é um ambiente de verificação, mas sim um *ambiente de certificação*. Desta maneira, é suposto (ou altamente recomendável) que o desenvolvedor de um componente certificável (seja ele de computação, *workflow*, ou de qualquer outra espécie que por ventura venha a ter componente certificador na HPC Shelf no futuro) já tenha verificado, fora da HPC Shelf, através das respectivas ferramentas de verificação empregadas pelo componente certificador pretendido, nos programas do componente, propriedades formais que serão verificadas pelo referido componente certificador (todas ou uma parte significante delas). Com isso, o desenvolvedor do componente não é obrigado a abandonar as ferramentas de verificação que comumente utiliza, e se refere ao mecanismo de certificação

da nuvem apenas para atestar a confiabilidade do componente que desenvolveu, através de abstrações adequadas (componentes certificadores). É nesse ponto que a figura do certificador de componentes (ator) tem um papel fundamental: criar abstrações (componentes certificadores, táticos e orquestrações dos últimos) que visam potencializar a aplicabilidade da verificação formal como um meio para a certificação de componentes, sem, contudo, limitar de forma significativa a forma de uso das ferramentas de verificação já usadas pelos desenvolvedores dos componentes. Note ainda que o mecanismo de certificação proposto utiliza exclusivamente a verificação formal como ferramenta, mas nada impede que outros meios de certificar componentes possam ser investigados e empregados no futuro.

Como se pôde notar ao longo do texto, o tipo de especificação formal pretendido neste trabalho é uma especificação parcial dos componentes, a qual é alcançada através de uma especificação parcial dos seus programas. A especificação parcial dos programas, por sua vez, é alcançada através de um tipo de verificação pontual deles, ou seja, apenas propriedades formais acerca de pontos ou comportamentos específicos dos programas são verificadas. Especificações completas dos programas não são o foco deste trabalho e são geralmente criadas a partir de ferramentas de verificação assistida (provadores interativos), através da colaboração entre usuário e ferramenta. O tipo de verificação adotado (pontual) foi apresentado no texto de duas formas: a verificação semi-automática, no caso em que os programas necessitam ser anotados com asserções que guiarão os provadores nas provas das propriedades, e a verificação automática, no caso em que os programas por si só já são suficientes aos provadores para provar as propriedades. A verificação assistida por computador até foi abordada neste trabalho, mas em um contexto limitado: o provador interativo apenas se destina a verificar se um arquivo contendo teoremas (condições de verificação) completados pelo desenvolvedor do componente, os quais foram criados por uma linguagem de verificação intermediária a partir de um programa escrito em uma linguagem de alto nível (C ou Java, por exemplo) e das propriedades a serem provadas, realmente contém as provas dos teoremas. Contudo, não se considera que essa limitação seja demasiado relevante para o arcabouço de certificação, uma vez que, além de se estar interessado somente em uma verificação pontual dos programas, o foco é também em verificar programas de aplicações reais, escritos em linguagens de alto nível e propósito geral, para os quais as pesquisas realizadas neste trabalho apontaram somente essa forma particular de uso para os provadores interativos. Mais ainda, nas pesquisas realizadas percebeu-se uma evolução significativa, tanto em pesquisas quanto no número de usuários interessados, com relação às ferramentas de verificação semi-automática, como, por exemplo, Frama-C, o que faz acreditar que essa forma de verificação será a mais disseminada no futuro.

Outra observação que se faz é que o objetivo principal do arcabouço de certificação é aumentar os níveis de confiança sobre os componentes certificados, e não somente

encontrar erros de programação e de projeto neles. Assim, as propriedades que se está interessado principalmente são aquelas que permitem enriquecer a especificação formal dos componentes, muito embora possam também ser verificadas propriedades que representem a detecção de erros de programação ou projeto.

Tendo em vista tornar o processo de certificação de componentes menos disruptivo possível, utilizou-se como pano de fundo para ele um sistema de certificação paralela. Essa nova abstração visa reusar o fluxo de trabalho já utilizado pelo provedor de aplicações para construir um sistema de computação paralela, utilizando também componentes como blocos de construção. De fato, as orquestrações de componentes nas duas perspectivas (sistema de computação paralela e sistema de certificação paralela), embora com objetivos distintos, são operacionalmente parecidas, e podem ser de alguma maneira sobrepostas sem interferência durante a execução da aplicação.

Os componentes que podem compor os sistemas de certificação paralela são, além dos próprios componentes a serem certificados, os componentes certificadores e componentes táticos. No tocante aos componentes certificadores, acredita-se que a inclusão, no momento da criação de uma aplicação, de componentes que, de uma forma reflexiva, são capazes de inspecionar e certificar os componentes da aplicação vale à pena ser explorada por duas razões. Por um lado, a complexidade inerente de aplicações baseadas em nuvens, dada a heterogeneidade dos recursos e o controle aberto e descentralizado que possuem, implica na necessidade de se aumentar a escala das ferramentas de modelagem e verificação formal. Por outro lado, a nuvem é por si própria um ecossistema em que componentes que orquestram motores de verificação podem coexistir e serem invocados para certificar suas próprias computações e as maneiras em que elas interagem, como foi sugerido nesse trabalho.

No que tange aos componentes táticos, acredita-se que a invocação de componentes que encapsulam infraestruturas de verificação formal de *software* em vez da invocação direta das infraestruturas de verificação pelas aplicações é um passo para aumentar a escala da aplicabilidade da verificação formal, como preza a idéia de oferecer *verificação como um serviço* (VaaS). Outro ponto forte dos componentes táticos é que não são meramente componentes que encapsulam infraestruturas de verificação formal, uma vez que existe dentro de cada componente uma lógica de computação paralela, a qual permite extrair o maior desempenho da plataforma alvo, através da aplicação de diversos níveis de paralelismo nas unidades do componente. Até onde se sabe, a arquitetura proposta nesta Tese é a primeira arquitetura VaaS que emprega exclusivamente técnicas de Computação de Alto Desempenho para esse fim.

Os *workflows* da HPC Shelf são compostos em geral por componentes de grossa granularidade, os quais geralmente possuem um número limitado de ações computacionais, porém computacionalmente intensivas. A linguagem SAFeSWL é assumidamente expressiva para representar *workflows* que orquestram esses componentes. Dessa forma,

considerando que os componentes táticos são componentes com essas características (*process* e *data intensive*, mas não *coordination intensive*), acredita-se que uma linguagem para composição de *workflows* de verificação com poucos construtores de orquestração, na mesma filosofia da linguagem SAFeSWL, é necessária e suficiente para expressar todos os padrões de ativação das ações de verificação desse componentes. TCOL, em sendo uma extensão de SAFeSWL, é um passo nesse sentido.

7.1.2 Componentes Certificadores Particulares

Tendo em vista certificar componentes das aplicações dos estudos de caso, propuseram-se como *contribuições suplementares* dois componentes certificadores particulares, o Componente Certificador de *Workflows* Científicos (SWC2) e o Componente Certificador de Componente de Computação (C4). O primeiro é voltado à orquestração de componentes táticos visando provar propriedades de interesse da aplicação sobre o *workflow* associado. Já o segundo é voltado à orquestração de componentes táticos com o intuito de provar propriedades formais sobre um determinado componente de computação que possam aumentar os níveis de confiança por parte de seus utilizadores para compor aplicações. A criação desses componentes, por si só, já foi um objeto de pesquisa que demandou muito esforço de investigação. Para compor seus componentes táticos, foi necessário buscar na literatura técnicas e ferramentas de verificação formal de *software* capazes de expressar características tanto dos *workflows* de componentes com requisitos de Computação de Alto Desempenho quanto dos programas paralelos contidos nesses componentes.

7.1.2.1 Componente Certificador de *Workflow* Científico (SWC2)

No caso da certificação de *workflows* (componente SWC2), a modelagem dos *workflows* de SAFeSWL para uma ferramenta de raciocínio sobre processos concorrentes foi uma alternativa natural, considerando que, do ponto de vista da implementação no SAFe, eles são coordenadores de processos concorrentes, estabelecendo suas formas de ativação e pontos de sincronização. A modelagem foi realizada através da ferramenta mCRL2, a qual emprega o μ-calculus modal como lógica modal para especificação formal de propriedades a serem verificadas.

Tendo em vista realizar modelagens e verificações de propriedades sobre *workflows* científicos mais realistas, pôs-se a investigar os principais padrões encontrados nesses *workflows*. De um total de 8 padrões mais comuns, encontrados em *workflows* confeccionados nos principais SGWfC (Sistemas de Gerenciamento de *Workflows* Científicos) disponíveis, puderam ser estabelecidas características especiais a serem capturadas pela modelagem e propriedades que, por padrão, devem estar disponíveis para serem verificadas. O alcance desse estudo levou à constatação de que a abordagem descrita nesta Tese foi a primeira iniciativa de modelagem e verificação de *workflows* de cunho científico e seus padrões mais comuns.

Dentre as características especiais encontradas, citam-se a composição dos *workflows* da aplicação (coordenação exógena) e *workflows* internos dos componentes (coordenação endógena) e a verificação dos *timeouts*. No caso particular da composição de *workflows*, viu-se que os *workflows* internos dos componentes da HPC Shelf, os quais habilitam/desabilitam as ações dos componentes, puderam tornar a modelagem muito mais realista, e permitiram a verificação de propriedades que não puderam ser verificadas somente a partir do *workflow* da aplicação, como foi mostrado no estudo de caso 1, do Capítulo 6. Como ponto fraco da modelagem proposta, contudo, cita-se a impossibilidade de verificar os *timeouts* dos *workflows*, uma vez que o mCRL2 não proveu construtores adequados para desenvolver essa verificação. No futuro, contudo, pretende-se desenvolver essa verificação através de outro componente tático, que trabalhará em conjunto com o mCRL2.

Tendo em vista garantir a corretude da modelagem proposta, criou-se uma semântica operacional para SAFeSWL. As regras de execução definidas para essa semântica permitiram derivar com precisão e segurança as regras de tradução a serem executadas pelo algoritmo S2m, as quais são descritas formalmente no Apêndice A.1.

Por fim, é importante salientar que o modelo de certificação de *workflows* científicos proposto neste trabalho não é um modelo geral, que pode ser aplicado de forma direta a outros SGWfC, uma vez que cada SGWfC possui suas próprias características de implementação. O que é esperado com este trabalho, portanto, é que ele sirva como base conceitual para outros trabalhos. Para tanto, acredita-se que os padrões verificáveis que foram encontrados nos *workflows* e os conceitos de componente certificador, componente tático e sistema de certificação paralela possam ser aplicados com sucesso na certificação de *workflows* em SGWfC bem difundidos, tais como Pegasus ou Taverna.

7.1.2.2 Componente Certificador de Componente de Computação (C4)

No caso da certificação de componentes de computação, foi necessário um estudo aprofundado na literatura sobre lógicas para especificação, técnicas e ferramentas de verificação formal de *software* que permitissem verificar programas paralelos, sobretudo os voltados à Computação de Alto Desempenho, que são, na sua maioria, implementados através da biblioteca MPI (Capítulo 3).

No tocante às lógicas de especificação voltadas à verificação de componentes de computação, a única alternativa encontrada foi a Lógica de Floyd-Hoare e suas variações. De fato, ela é a lógica de especificação formal mais empregada na verificação de programas de qualquer natureza e possui diversas extensões propostas, cada uma voltada a lidar com uma classe diferente de programas. Por exemplo, possui a lógica de separação (REYNOLDS, 2002), que permite tratar, dentre outras coisas, de segurança sobre programas paralelos com memória compartilhada, a extensão de Owicky e Gries (OWICKI; GRIES, 1976), para especificar funcionalmente programas paralelos com memória compartilhada,

e a extensão de Apt (APT, 1986), para especificar funcionalmente programas paralelos com memória distribuída.

Como técnica de verificação que permite a automatização da abordagem axiomática da lógica de Floyd-Hoare, tem-se a *verificação dedutiva de programas*. Sua grande vantagem é se basear completamente no raciocínio lógico, não necessitando de qualquer informação do ambiente de execução do programa, como por exemplo a quantidade de processos paralelos (unidades do componente) relativos ao programa do componente que serão de fato lançados, tendo, portanto, tempo de verificação constante. Como desvantagem, aponta-se a necessidade de o usuário ter que compreender em detalhes o funcionamento do programa, a fim de guiar a ferramenta na verificação do programa, seja através de táticas pré-disponibilizadas ou através de asserções em pontos estratégicos do programa. Existem poucas ferramentas de verificação dedutiva construídas visando a especificação funcional de programas paralelos até o momento. Quando se trata de verificação de programas MPI, resume-se ao ParTypes, que permite apenas que seja confrontado contra o programa um protocolo de alto nível que reflete suas operações de comunicação.

Existe ainda a técnica *model-checking*, muito difundida na academia e na indústria, mas que permite verificar um programa explorando de uma forma conveniente os estados relevantes à execução dele. Em geral, *model checkers* de programas paralelos são automáticos, ou seja, os programas e alguns parâmetros de execução já são suficientes à execução da verificação das propriedades formais. A quantidade de processos paralelos (unidades do componente) que serão lançados quando o programa do componente for executado é um parâmetro sempre solicitado por essas ferramentas, a fim de reduzir o espaço de busca e, por consequência, o tempo de verificação, que é em geral proporcional ao número de processos engajados na computação. Como principal desvantagem dessa técnica, cita-se a explosão espacial dos estados, que tende a consumir de forma acelerada os recursos de memória da máquina utilizada. Os *model checkers* aptos a verificar programas MPI investigados foram o ISP e o CIVL.

7.1.2.3 Estudos de Caso

Para demonstrar e validar as características do arcabouço de certificação e dos componentes certificadores SWC2 e C4, selecionaram-se duas aplicações bem difundidas no meio científico: a computação paralela em larga-escala através do modelo MapReduce e a computação de mosaicos de astrofotografia com o uso da ferramenta Montage. Construíram-se implementações de protótipos do arcabouço de certificação e dos componentes certificadores SWC2 e C4 e, a partir do que foi avaliado nos estudos de caso, pôde-se tomar algumas conclusões a respeito deles, as quais serão descritas nos parágrafos que se seguem.

Com relação ao arcabouço de certificação, tem-se:

- Quanto ao **reuso**, ele é herdado diretamente do modelo de programação orientado a componentes Hash. Como quaisquer componentes, certificadores e táticos devem ser

implementados de forma independente, expondo *interfaces*, tanto funcionais (*portas de ações*) quanto não-funcionais (*portas de serviços*), que facilitam a programação *com* e *para* reuso. Dessa forma, um certificador de componentes pode reusar os mesmos componentes táticos em diversos componentes certificadores, permitindo a geração de *workflows* de verificação distintos. Mais ainda, um provedor de aplicações ou desenvolvedor de componentes pode reusar componentes certificadores para certificar componentes de diversas aplicações;

- Quanto à **complexidade** na criação de um sistema de certificação paralela, pode-se dizer que há um nível de dificuldade semelhante ao de criar um sistema de computação paralela no SAFe. Ou seja, o provedor de aplicações ou o desenvolvedor de componentes deve ter experiência em programação Java ou C# e em manipulação de arquivos XML, que são tecnologias bastante difundidas, além de conhecer a API do SAFe. Quanto à **complexidade** na criação de componentes certificadores e táticos, pode-se dizer que a dificuldade se assemelha à criação de componentes de computação Hash utilizando o HPE;
- Quanto à **extensibilidade**, crê-se que ela é o ponto mais forte da arquitetura proposta. Ao passo que surjam novas espécies de componentes, novas classes de propriedades formais ou novas ferramentas de verificação formal, novos componentes certificadores e táticos podem ser adicionados. Mais ainda, esses componentes podem expor novas portas de serviços e de ações, diferentes das já por existentes por padrão, para lidar com requisitos específicos de comunicação de uma arquitetura de verificação específica (e.g. as portas de ações **fill_vcs** e **verify_vcs** e a porta de serviços para transmitir e receber o arquivo com as condições de verificação do componente C4, quando este lida com provadores interativos). Extensões são ainda possíveis em TCOL, as quais são facilitadas pelo uso do padrão *Visitor* em sua implementação. Novamente, todo o código do arcabouço de certificação e dos componentes SWC2 e C4 está disponível publicamente em <http://github.com/UFC-MDCC-HPC/HPC-Shelf-Certification>;
- Quanto à **eficiência**, os tempos de certificação apontados nos estudos de caso revelaram que a certificação paralela realizada pelo arcabouço de certificação reduziu consideravelmente o tempo final de certificação. No caso da certificação do *workflow* de processamento MapReduce, note que intencionalmente uma especificação simplificada para os *workflows* internos dos componentes foi fornecida e que o conjunto de 20 propriedades que foram verificadas é relativamente pequeno, uma vez que quão menores fossem as tarefas de verificação, mais evidente seria a sobrecarga do arcabouço de certificação. Os tempos de certificação obtidos levam a crer que, no caso em que uma especificação mais completa seja fornecida e o conjunto de propriedades seja consideravelmente maior, o paralelismo tenderá a estabelecer um ganho próximo do linear. No caso da certificação dos componentes paralelos do

Montage, devido ao fato de cada componente possuir um único programa e serem empregados apenas dois componentes táticos, a sobrecarga do arcabouço de certificação foi claramente perceptível e maior que no estudo de caso anterior, porém não ultrapassando em nenhum cenário o tempo sequencial. Note, contudo, que é razoável pensar que a maioria dos componentes possua em torno de uma dezena de programas, dentre os quais alguns poderiam ser verificados paralelamente por um número maior de componentes táticos. Para esses casos, os testes levam a crer que essa maior paralelização levará a ganhos cada vez mais significativos quando comparados com os tempos sequenciais de verificação. Por fim, note ainda que os tempos sequenciais tomados por base nos experimentos são tempos livres, inclusive, da sobrecarga da própria HPC Shelf, uma vez que foram calculados pela invocação direta das ferramentas de verificação, o que corrobora a eficiência argumentada.

No tocante ao componente certificador SWC2, sua arquitetura e a modelagem de *workflows* para mCRL2 que realiza foram expressivas o suficiente para representar todas as propriedades relevantes elencadas neste trabalho para o *workflow* de processamento MapReduce. Um total de 20 propriedades formais, entre propriedades padrão e *ad hoc*, divididas em um grupo de segurança e um grupo de continuidade, foram provadas. A implementação do componente tático mCRL2, a qual divide o conjunto de propriedades de responsabilidade do componente tático entre as unidades paralelas do componente e divide o conjunto de propriedades de responsabilidade de uma unidade paralela em um conjunto de *threads*, revelou-se, apesar do cenário de modelagem simplificado e do conjunto pequeno de propriedades, uma alternativa real para diminuir significantemente o tempo final de certificação. Cabe aqui uma observação acerca do problema da explosão do espaço de estados na técnica de *model checking*. Observou-se nos testes iniciais que, mesmo em se tratando de um cenário de modelagem simplificado, a verificação do *workflow* MapReduce em um computador convencional demonstrou-se intratável, levando ao consumo total da memória (4GB) durante a verificação de algumas propriedades. Esse problema, contudo, foi resolvido quando se portou a verificação para uma máquina virtual da nuvem, a qual possuía 16GB de memória, o que é razoável para máquinas virtuais de cunho científico. Assim, considera-se que o problema da explosão espacial possa ser contornado na maioria dos casos quando se realiza a verificação em nuvens computacionais, as quais, na sua maioria, dispõem de infraestruturas computacionais extremamente poderosas, sem falar na capacidade adaptativa que possuem, podendo facilmente serem escaladas para agregar mais recursos computacionais quando os atuais não suprem as necessidades.

No que concerne ao componente certificador C4, sua arquitetura permitiu a certificação dos componentes paralelos da ferramenta Montage. Devido à escassez das ferramentas de verificação formais hábeis a verificar código MPI, o conjunto de propriedades formais que puderam ser verificadas sobre cada componente se resumiu ao conjunto de propriedades formais padrão do componente tático ISP mais a verificação de protocolo

feita pelo componente tático ParTypes. Claramente, essa limitação não é do arcabouço de certificação nem do componente C4, mas fez com que a sobrecarga acarretada pela orquestração dos dois componentes táticos fosse mais acentuada, mas não superou o tempo de verificação sequencial experimental calculado para os componentes paralelos do Montage. Por fim, observa-se que se possuía interesse em utilizar também o *model checker* CIVL para verificar a equivalência funcional dos programas paralelos do Montage com os programas sequenciais relacionados, mas a ferramenta demonstrou bastante instabilidade e, mesmo tendo recorrido aos autores da ferramenta para a correção de *bugs*, não foi possível utilizá-la efetivamente.

7.2 Trabalhos Futuros

Esta Tese de Doutorado é o ponto de partida para outros projetos de pesquisa que tenham por finalidade estender o arcabouço de certificação proposto para atender a outras necessidades de certificação que se tornem úteis para certificar componentes na plataforma HPC Shelf, sejam esses componentes relacionados diretamente a *workflows* científicos, a programas paralelos em geral suportados pela plataforma ou inclusive a novas espécies de componentes que passem a ser certificadas.

Os parágrafos que se seguem buscam delinear questões de pesquisa a serem tratadas em trabalhos futuros que estenderão os resultados desta Tese de Doutorado.

Investigar características verificáveis dos conectores da HPC Shelf

Dado que nesta Tese investigamos a certificação de componentes *workflow* e de computação, é natural pensar também em como certificar componentes da espécie conector. Conectores têm sido objeto de pesquisa de diversas iniciativas científicas, dentre as quais se destaca o modelo de conectores Reo (ARBAB, 2004). Nosso trabalho, contudo, consistirá em observar possíveis características inerentes a componentes conectores de aplicações com requisitos de computação de alto desempenho, notadamente os da HPC Shelf, e modelar tais características em um modelo formal de conectores, para sua posterior verificação.

Verificação dos *timeouts* dos *workflows*

Como dito no Capítulo 4, o único padrão de *workflow* científico verificável que não pôde ser modelado pelo componente certificador SWC2 foi os *timeouts* dos *workflows*. A ideia consistia em estabelecer um modelo temporal que contabilizasse o início da ativação de uma ação no *workflow* da aplicação e o início, de fato, da ativação da ação no componente. A diferença entre esses dois tempos seria então verificada com relação a um valor de *timeout* configurado pela aplicação. O mCRL2 até oferece construtores que permitem registrar os tempos em que as ações devem ocorrer, mas o *model checking* de propriedades sobre modelagens temporizadas não está disponível atualmente. Como alternativa ao mCRL2, já se começou a investigar a tradução dos *workflows* para a ferramenta

de modelagem de tempo real Uppaal (BEHRMANN et al., 2006), que tem como elemento formal os autômatos temporizados. Outra alternativa a ser investigada no futuro seriam as cadeias de Markov interativas (HERMANNS, 2002), através de alguma das ferramentas disponíveis a seu respeito, para estabelecer um comportamento probabilístico a ser verificado sobre as violações de *timeouts*.

Verificação de faltas de projeto sobre *workflows* científicos

De acordo com a classificação de Avizienis *et al.* (AVIZIENIS; LAPRIE; RANDELL; LANDWEHR, 2004), uma falha (*failure*) é um evento que ocorre quando um serviço fornecido por um sistema apresenta um comportamento diferente da sua especificação funcional, podendo ser percebida por usuários do sistema. Um erro (*error*), por sua vez, é uma parte do estado do sistema que pode levá-lo a uma falha posterior. Já uma falta (*fault*) é a causa física ou algorítmica de um erro. Faltas de projeto (*design faults*) são faltas cometidas por projetistas durante o projeto do *software*, diferentemente das faltas de *software* (*software faults*), que são cometidas por desenvolvedores no momento da codificação do *software*. Observe que, como as especificações de projetos de *software* são feitas geralmente em linguagens de alto nível e específicas do domínio das organizações, caso se queira verificar faltas de projeto nessas especificações, um verificador *ad hoc* (geralmente um *model checker*) deve ser construído ou uma modelagem da especificação e das faltas a se verificar deve ser feita para uma ferramenta de verificação. Note ainda que faltas de projeto não dizem respeito diretamente a detalhes específicos de uma aplicação projetada, mais sim a comportamentos faltosos globais que podem ser observados em diversas aplicações, os quais são geralmente motivados por "vícios" que os projetistas podem empregar no desenvolvimento dos projetos dessas aplicações. Um exemplo comum de falta de projeto é o não-determinismo, que pode ser quando a computação é guiada por computações disparadas por regras ativas no modelo e em um determinado estado pode haver mais de uma regra ativa. Existem diversos tipos de faltas de projeto já estudadas em sistemas adaptativos sensíveis ao contexto (SAMA et al., 2010; CUBO; SAMA; RAIMONDI; ROSENBLUM, 2009), e se tem a ideia de adaptar essas regras para serem verificadas no projeto de *workflows* científicos construídos na HPC Shelf. Note que já são verificadas algumas faltas de projeto através do certificador SWC2, como por exemplo ausência de *deadlock* e terminação, entretanto, caso seja mais produtivo, pode-se construir um verificador dedicado (*model checker*) para uma linguagem de projeto dos *workflows*, seja ela a própria SAFeSWL ou outra linguagem de mais alto nível, e encapsulá-lo em um componente tático. Vislumbram-se ainda componentes táticos que atuassem como repositórios de faltas de projeto verificáveis e que pudessem ser acoplados a componentes táticos verificadores por componentes certificadores, de acordo com a demanda da aplicação.

Implementação do fluxo de transferência de arquivos de teoremas na verificação com provadores interativos

Um aspecto que ficou preterido tanto na implementação do protótipo do arca-

bouço de certificação construído nesta Tese quanto nos estudos de caso foi o fluxo em que se tem como provador final um provador interativo, no qual o arcabouço envia e recebe preenchido do desenvolvedor do componente um arquivo contendo teoremas (condições de verificação) provados. Pretende-se, em próximas versões do arcabouço, já ter essa funcionalidade disponível.

Verificação paralela interna de uma ferramenta de verificação

Como dito na Seção 5.3, uma granularidade mais fina de paralelismo poderia ocorrer diretamente a partir da ferramenta de verificação. Nesse caso, caso a ferramenta (sequencial) verifique um conjunto de propriedades ou condições de verificação em lote, ela mesma poderia disparar tarefas paralelas para verificar essas propriedades ou condições de verificação, através de *threads* ou processos distribuídos. Para se implementar esse comportamento, a implementação da ferramenta teria que ser *open-source* para ser alterada ou ela teria que permitir incrementos através de *plugins*. Ainda se precisa investigar melhor qual ferramenta investir, mas o Frama-C, em sendo uma ferramenta colaborativa e que permite o desenvolvimento de *plugins*, parece uma alternativa interessante.

Investigar verificação de programas MPI

A biblioteca MPI ainda é um padrão dominante em aplicações de Computação de Alto Desempenho. Apesar de se ter investigado com certa profundidade as ferramentas já disponíveis para verificar programas paralelos escritos através dessa biblioteca, crê-se que, com algumas modificações ou não, outras ferramentas de verificação que ainda não foram empregadas na verificação de código MPI possam trazer novos resultados promissores. A exemplo disso, chegou-se a experimentar o VeriFast, que implementa a lógica de separação, com customizações feitas pelo autor deste trabalho e também pelos próprios autores da ferramenta, a pedido do primeiro, visando verificar programas MPI.

Referências

AALST, W. M. P. V. d. Verification of Workflow Nets. In: *Proceedings of the 18th International Conference on Application and Theory of Petri Nets.* : Springer-Verlag, 1997. p. 407–426.

AALST, W. M. P. Van der. The Application of Petri Nets to Workflow Management. *Journal of circuits, systems, and computers*, World Scientific, v. 8, n. 01, p. 21–66, 1998.

ADAM, N. R.; ATLURI, V.; HUANG, W.-K. Modeling and analysis of workflows using Petri nets. *Journal of Intelligent Information Systems*, Springer, v. 10, n. 2, p. 131–158, 1998.

AFFELDT, R.; NOWAK, D.; YAMADA, K. Certifying Assembly with Formal Security Proofs: The case of BBS. *Science of Computer Programming*, Elsevier, v. 77, n. 10, p. 1058–1074, 2012.

ALENCAR, J. M. U. d. *Reconfiguração Elástica de Componentes Paralelos sobre Nuvens de Serviços de Computação de Alto Desempenho.* 125 f. Tese (Doutorado em Ciência da Computação) — Programa de Pós-Graduação em Ciência da Computação (MDCC), Universidade Federal do Ceará, Fortaleza, 2017.

ALLAN, B. A.; ARMSTRONG, R. C.; WOLFE, A. P.; RAY, J.; BERNHOLDT, D. E.; KOHL, J. A. The CCA Core Specification in a Distributed Memory SPMD Framework. *Concurrency and Computation: Practice and Experience*, Wiley Online Library, v. 14, n. 5, p. 323–345, 2002.

ALMEIDA José Bacelar; BARBOSA Manuel; FILLIÂTRE Jean-Christophe; PINTO Jorge Sousa; VIEIRA Bárbara. CAOVerif: An Open-source Deductive Verification Platform for Cryptographic Software Implementations. *Science of Computer Programming*, v. 91, Part B, n. 0, p. 216 – 233, 2014. ISSN 0167-6423. Special Issue on Selected Contributions from the Open Source Software Certification (OpenCert) Workshops. Disponível em: <http://www.sciencedirect.com/science/article/pii/S016764231200189X>.

APT, K. R. Correctness Proofs of Distributed Termination Algorithms. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 8, n. 3, p. 388–405, jun. 1986. ISSN 0164-0925. Disponível em: <http://doi.acm.org/10.1145/5956.6000>.

APT, K. R.; BOER, F. de; OLDEROG, E. R. *Verification of Sequential and Concurrent Programs.* 3rd. ed. : Springer Publishing Company, Incorporated, 2009. ISBN 184882744X, 9781848827448.

ARBAB, F. Reo: A Channel-based Coordination Model for Component Composition. *Mathematical. Structures in Comp. Sci.*, Cambridge University Press, New York, NY, USA, v. 14, n. 3, p. 329–366, jun. 2004. ISSN 0960-1295. Disponível em: <http://dx.doi.org/10.1017/S0960129504004153>.

ARMSTRONG, R.; KUMFERT, G.; MCINNES, L. C.; PARKER, S.; ALLAN, B.; SOTTILE, M.; EPPERLY, T.; TAMARA, D. The CCA Component Model For High-Performance Scientific Computing. *Concurrency and Computation: Practice and Experience*, Wiley, v. 18, n. 2, p. 215–229, 2006.

ATTIE, P.; SINGH, M.; SHETH, A. P.; RUSINKIEWICZ, M. Specifying and Enforcing Intertask Dependencies. In: *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings.* 1993. p. 134–145.

AVIZIENIS, A.; LAPRIE, J. C.; RANDELL, B.; LANDWEHR, C. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE transactions on dependable and secure computing*, IEEE, v. 1, n. 1, p. 11–33, 2004.

AYAD, A.; MARCHÉ, C. Multi-Prover Verification of Floating-Point Programs. In: *IJCAR*. : Springer, 2010. (Lecture Notes in Computer Science, v. 6173), p. 127–141. ISBN 978-3-642-14202-4.

BAETEN, J. C. M.; BASTEN, T.; RENIERS, M. A. *Process Algebra: Equational theories of communicating processes*. : Cambridge University Press, 2010. (Cambridge Tracts in Theoretical Computer Science (50)).

BAIER, C.; KATOEN, J.-P. *Principles of Model Checking*. : The MIT Press, 2008. ISBN 026202649X, 9780262026499.

BARNETT, M.; CHANG, B.-Y. E.; DELINE, R.; JACOBS, B.; LEINO, K. R. M. Boogie: A Modular Reusable Verifier for Object-oriented Programs. In: *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*. : Springer-Verlag, 2006. (FMCO), p. 364–387. ISBN 3-540-36749-7, 978-3-540-36749-9.

BARTHE, G.; GRÉGOIRE, B.; KUNZ, C.; REZK, T. Certificate Translation for Optimizing Compilers. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 31, n. 5, p. 18:1–18:45, jul. 2009. ISSN 0164-0925.

BASILI Victor R.; CARVER Jeffrey C.; CRUZES Daniela; HOCHSTEIN Lorin M.; HOLLINGSWORTH Jeffrey K.; SHULL Forrest; ZELKOWITZ Marvin V. Understanding the High-Performance-Computing Community: A Software Engineer's Perspective. *IEEE Software*, IEEE Computer Society, Los Alamitos, CA, USA, v. 25, n. 4, p. 29–36, 2008. ISSN 0740-7459.

BAUDE, F.; CAROMEL, D.; DALMASSO, C.; DANELUTTO, M.; GETOV, W.; HENRIO, L.; PREZ, C. GCM: A Grid Extension to Fractal for Autonomous Distributed Components. *Annals of Telecommunications*, v. 64, n. 1, p. 5–24, 2009.

BEHRMANN Gerd; DAVID Alexandre; LARSEN Kim; HÅKANSSON John; PETTERSSON Paul; YI Wang; HENDRIKS Martijn. UPPAAL 4.0. In: *QEST, 3rd Int. Conf. Quantitative Evaluation of Systems*. : IEEE Computer Society, 2006. p. 125–126.

BELLETTINI, C.; CAMILLI, M.; CAPRA, L.; MONGA, M. Distributed CTL Model Checking using MapReduce: Theory and Practice. *Concurrency and Computation: Practice and Experience*, Wiley Online Library, 2015.

BERGSTRA, J. A.; KLOP, J. W. Process Algebra for Synchronous Communication. *Information and control*, Elsevier, v. 60, n. 1, p. 109–137, 1984.

BERNHOLDT, D. E.; NIEPLOCHA, J.; SADAYAPPAN, P. Raising Level of Programming Abstraction in Scalable Programming Models. In: Madrid, Spain. *IEEE International Conference on High Performance Computer Architecture (HPCA), Workshop on Productivity and Performance in High-End Computing (P-PHEC)*. : IEEE Computer Society, 2004. p. 76–84.

BERRIMAN, G. B.; DEELMAN, E.; GOOD, J. C.; JACOB, J. C.; KATZ, D. S.; KESSELMAN, C.; LAITY, A. C.; PRINCE, T. A.; SINGH, G.; SU, M.-H. Montage: a Grid-enabled Engine for Delivering Custom Science-grade Mosaics on Demand. In: INTERNATIONAL SOCIETY FOR OPTICS AND PHOTONICS. *SPIE Astronomical Telescopes+ Instrumentation*. 2004. p. 221–232.

BLACKBURN, P.; BENTHEM, J. F. A. K. van; WOLTER, F. *Handbook of Modal Logic*. : Elsevier, 2006. v. 3.

BLECH; GRÉGOIRE. Certifying Compilers using Higher-order Theorem Provers as Certificate Checkers. *Formal Methods in System Design*, v. 38, n. 1, p. 33–61, 2011.

BOBOT, F.; FILLIÂTRE, J.-C.; MARCHÉ, C.; PASKEVICH, A. Why3: Shepherd Your Herd of Provers. In: *Boogie: First International Workshop on Intermediate Verification Languages*. Wroclaw, Poland: , 2011. p. 53–64. Disponível em:

<https://hal.inria.fr/hal-00790310>.
BOBOT François; FILLIÂTRE Jean-Christophe; MARCHÉ Claude; PASKEVICH Andrei. Let's Verify This with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, Springer Berlin / Heidelberg, p. 1–19, 2014.
BROWN, P. F.; HAMILTON, R. M. B. A. Reference Model for Service Oriented Architecture 1.0. Citeseer, 2006.
BRUNETON, E.; COUPAYE, T.; LECLERCQ, M.; QUMA, V.; STEFANI, J.-B. The Fractal Component Model and Its Support In Java. *Software - Practice and Experience*, Wiley, v. 36, p. 1257–1284, 2006.
CARVALHO JUNIOR, F. H. de; CORREA, R. C. The Design of a CCA Framework with Distribution, Parallelism, and Recursive Composition. In: *Workshop on Component-Based High Performance Computing (CBHPC)*. : IEEE, 2010. p. 339–348. ISBN 9781424493470.
CARVALHO JUNIOR, F. H. de; LINS, R.; CORREA, R. C.; ARAÚJO, G. A. Towards an Architecture for Component-Oriented Parallel Programming. *Concurrency and Computation: Practice and Experience*, v. 19, n. 5, p. 697–719, 2007. ISSN 1532-0626.
CARVALHO JUNIOR, F. H. de; LINS, R. Dueire. The # Model: Separation of Concerns for Reconciling Modularity, Abstraction and Efficiency in Distributed Parallel Programming. In: *Proceedings of the ACM Symposium on Applied Computing (SAC), Santa Fe, New Mexico, USA*. 2005. p. 1357–1364. Disponível em: <http://doi.acm.org/10.1145/1066677.1066984>.
CARVALHO JUNIOR, F. H. de; REZENDE, C. A. A Case Study on Expressiveness and Performance of Component-Oriented Parallel Programming. *Journal of Parallel and Distributed Computing*, v. 73, n. 5, p. 557–569, maio 2013. ISSN 0743-7315. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0743731512002882>.
CARVALHO JUNIOR, F. H. de; REZENDE, C. A.; SILVA, J. C.; ALAM, W. G. Al. Contextual Abstraction in a Type System for Component-Based High Performance Computing Platforms. In: *Lect. Notes in Computer Science*. : Springer, 2013. v. 8129, p. 90–104.
CARVALHO JUNIOR, F. H. de; REZENDE, C. A.; SILVA, J. C.; ALAM, W. G. Al. Contextual Abstraction in a Type System for Component-based High Performance Computing Platforms. *Science of Computer Programming*, 2016. ISSN 0167-6423. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0167642316300892>.
CHLIPALA, A.; MALECHA, G.; MORRISETT, G.; SHINNAR, A.; WISNESKY, R. Effective Interactive Proofs for Higher-order Imperative Programs. In: ACM. *ACM Sigplan Notices*. 2009. v. 44, p. 79–90.
CHLIPALA Adam. *Certified Programming with Dependent Types*. : MIT Press, 2011.
CHURCH, P.; WONG, A.; BROCK, M.; GOSCINSKI, A. Toward Exposing and Accessing HPC Applications in a SaaS Cloud. In: IEEE. *Web Services (ICWS), IEEE 19th International Conference on*. 2012. p. 692–699.
CLARKE, E. M.; GRUMBERG, O.; LONG, D. E. Model Checking and Abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)*, ACM, v. 16, n. 5, p. 1512–1542, 1994.
CLARKE JR., E. M.; GRUMBERG, O.; PELED, D. A. *Model Checking*. Cambridge, MA, USA: MIT Press, 1999. ISBN 0-262-03270-8.
CLEARY, A.; KOHN, S.; SMITH, S. G.; SMOLINSKI, B. Language Interoperability Mechanisms for High-performance Scientific Applications. In: *SIAM Workshop on*

Object-Oriented Methods for Interoperable Scientific and Engineering Computing. 1998.
COHEN, E.; DAHLWEID, M.; HILLEBRAND, M.; LEINENBACH, D.; MOSKAL, M.; SANTEN, T.; SCHULTE, W.; TOBIES, S. VCC: A Practical System for Verifying Concurrent C. In: *Theorem Proving in Higher Order Logics*. : Springer, 2009. p. 23–42.
CUBO, J.; SAMA, M.; RAIMONDI, F.; ROSENBLUM, D. A Model to Design and Verify Context-aware Adaptive Service Composition. In: IEEE. *Services Computing, SCC. IEEE International Conference on*. 2009. p. 184–191.
CUOQ, P.; KIRCHNER, F.; KOSMATOV, N.; PREVOSTO, V.; SIGNOLES, J.; YAKOBOWSKI, B. Frama-C. In: SPRINGER. *International Conference on Software Engineering and Formal Methods*. 2012. p. 233–247.
DANTAS, A. B. de Oliveira; CARVALHO JUNIOR, F. H. de; BARBOSA, L. Soares. A Framework for Certification of Large-scale Component-based Parallel Computing Systems in a Cloud Computing Platform for HPC Services. In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER*. Porto, Portugal: ScitePress, 2017. p. 229–240.
DANTAS, A. B. de Oliveira; CARVALHO JUNIOR, F. H. de; BARBOSA, L. Soares. Certification of Workflows in a Component-Based Cloud of High Performance Computing Services. In: *Formal Aspects of Component Software: 14th International Conference, FACS, Proceedings*. Braga, Portugal: Springer International Publishing, 2017. p. 198–215.
DANTAS, A. B. de Oliveira; CARVALHO JUNIOR, F. H. de; BARBOSA, L. Soares; PROENÇA, J. A Framework for Certification of Parallel Components in a Cloud Computing Platform for HPC Services. In: *Cloud Computing and Services Science: 7th International Conference, CLOSER, Revised Selected Papers*. : Springer International Publishing, 2017.
DAVULCU, H.; KIFER, M.; RAMAKRISHNAN, C. R.; RAMAKRISHNAN, I. V. Logic Based Modeling and Analysis of Workflows. In: ACM. *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 1998. p. 25–33.
DAYAL, U.; HSU, M.; LADIN, R. Organizing Long-running Activities with Triggers and Transactions. In: ACM. *ACM SIGMOD Record*. 1990. v. 19, n. 2, p. 204–214.
DEAN, J.; GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, ACM, v. 51, n. 1, p. 107–113, 2008.
DEELMAN, E.; SINGH, G.; SU, M.-H.; BLYTHE, J.; GIL, Y.; KESSELMAN, C.; MEHTA, G.; VAHI, K.; BERRIMAN, G. B.; GOOD, J.; LAITY, A.; JACOB, J. C.; KATZ, D. S. Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming*, Hindawi Publishing Corporation, v. 13, n. 3, p. 219–237, 2005.
DONGARRA, J. Basic Linear Algebra Subprograms Technical Forum Standard I. *International Journal of High Performance Applications and Supercomputing*, v. 16, n. 2, p. 115–199, feb 2002. ISSN 1094-3420.
DONGARRA, J.; OTTO, S. W.; SNIR, M.; WALKER, D. *An Introduction to the MPI Standard*. 1995. Disponível em: <http://www.netlib.org/tennessee/ut-cs-95-274.ps>.
ECMA International. *Common Language Infrastructure (CLI), Partitions I to VI*. 4. ed. 2006. Disponível em: <http://www.ecma-international.org/publications/standards/Ecma-335.htm>.
EMERSON, E. A.; CLARKE, E. M. Characterizing Correctness Properties of Parallel Programs Using Fixpoints. In: *Proceedings of the 7th Colloquium on Automata,*

Languages and Programming. London, UK, UK: Springer-Verlag, 1980. p. 169–181. ISBN 3-540-10003-2. Disponível em: <http://dl.acm.org/citation.cfm?id=646234.682526>.
ENGELER, E. Algorithmic Properties of Structures. *Theory of Computing Systems*, Springer, v. 1, n. 2, p. 183–195, 1967.
EPPERLY, T. G. W.; KUMFERT, G.; DAHLGREN, T.; EBNER, D.; LEEK, J.; PRANTL, A.; KOHN, S. High-performance Language Interoperability for Scientific Computing through Babel. *International Journal of High Performance Computing Applications*, Sage Publications, p. 1094342011414036, 2011.
FILLIÂTRE, J.-C. Deductive Software Verification. *International Journal on Software Tools for Technology Transfer*, Springer, v. 13, n. 5, p. 397–403, 2011.
FILLIÂTRE, J.-C.; MARCHÉ, C. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In: SPRINGER. *International Conference on Computer Aided Verification*. 2007. p. 173–177.
FILLIÂTRE Jean-Christophe; PASKEVICH Andrei. Why3 — Where Programs Meet Provers. In: FELLEISEN, M.; GARDNER, P. (Ed.). *Proceedings of the 22nd European Symposium on Programming*. : Springer, 2013. (Lecture Notes in Computer Science, v. 7792), p. 125–128.
FLOYD, R. W. Assigning Meanings to Programs. *Mathematical aspects of computer science*, v. 19, n. 19-32, p. 1, 1967.
FOKKINK, W. *Introduction to Process Algebra*. : Springer Science & Business Media, 2013.
FORUM, M. P. I. *MPI: A Message Passing Interface Standard - Version 2.2*. 2009. Disponível em: <http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf>.
FU, X.; BULTAN, T.; HULL, R.; SU, J. Verification of Vortex Workflows. In: *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS, Genova, Italy, April 2-6, Proceedings*. 2001. p. 143–157.
GOPALAKRISHNAN, G.; KIRBY, R. M.; SIEGEL, S.; THAKUR, R.; GROPP, W.; LUSK, E.; SUPINSKI, B. R. D.; SCHULZ, M.; BRONEVETSKY, G. Formal Analysis of MPI-based Parallel Programs. *Commun. ACM*, ACM, New York, NY, USA, v. 54, n. 12, p. 82–91, dez. 2011. ISSN 0001-0782.
GOPALAKRISHNAN, G. L.; KIRBY, R. M. Top Ten Ways to Make Formal Methods for HPC Practical. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. New York, NY, USA: ACM, 2010. (FoSER '10), p. 137–142. ISBN 978-1-4503-0427-6. Disponível em: <http://doi.acm.org/10.1145/1882362.1882392>.
GROOTE, J. F.; MATHIJSSEN, A.; RENIERS, M.; USENKO, Y.; WEERDENBURG, M. van. The Formal Specification Language mCRL2. In: *Methods for Modelling Software Systems: Dagstuhl Seminar 06351*. 2007.
GROOTE, J. F.; RENIERS, M. Modelling and Analysis of Communicating Systems. *Technical University of Eindhoven, rev*, v. 1478, p. 15–18, 2009.
HARRISON, A.; TAYLOR, I.; WANG, I.; SHIELDS, M. WS-RF Workflow in Triana. *International Journal of High Performance Computing Applications*, SAGE Publications, v. 22, n. 3, p. 268–283, 2008.
HAWBLITZEL, C.; PETRANK, E.; QADEER, S.; TASIRAN, S. Automated and Modular Refinement Reasoning for Concurrent Programs. In: SPRINGER. *International Conference on Computer Aided Verification*. 2015. p. 449–465.
HERMANNS Holger. *Interactive Markov Chains: The Quest for Quantified Quality*. : Springer, 2002. v. 2428. (Lecture Notes in Computer Science, v. 2428).
HOARE, C. A. R. An axiomatic Basis for Computer Programming. *Communications of*

the ACM, ACM, v. 12, n. 10, p. 576–580, 1969.
HOARE, C. A. R. *Communicating Sequential Processes.* : Prentice-Hall International Series in Computer Science, 1985.
HU Kai; LEI Lei; TSAI Wei-Tek. Multi-tenant Verification-as-a-Service (VaaS) in a cloud. *Simulation Modelling Practice and Theory*, v. 60, p. 122 – 143, 2016. ISSN 1569-190X. Disponível em: <http://www.sciencedirect.com/science/article/pii/S1569190X15001227>.
HULL, R.; LLIRBAT, F.; SIMAN, E.; SU, J.; DONG, G.; KUMAR, B.; ZHOU, G. Declarative Workflows that Support easy Modification and Dynamic Browsing. *ACM SIGSOFT Software Engineering Notes*, ACM, v. 24, n. 2, p. 69–78, 1999.
JACOBS, B.; PIESSENS, F. The VeriFast Program Verifier. *CW Reports*, 2008.
JACOBS, B.; SMANS, J.; PHILIPPAERTS, P.; VOGELS, F.; PENNINCKX, W.; PIESSENS, F. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In: *NASA Formal Methods.* : Springer, 2011. p. 41–55.
JACOBS, B.; SMANS, J.; PIESSENS, F. A Quick Tour of the VeriFast Program Verifier. In: *Programming Languages and Systems.* : Springer, 2010. p. 304–311.
JACOBS, B.; SMANS, J.; PIESSENS, F. VeriFast: Imperative Programs as Proofs. In: *VS-Tools workshop at VSTTE.* 2010.
KASSIOS, I. T. The Dynamic Frames Theory. *Formal Aspects of Computing*, Springer, v. 23, n. 3, p. 267–288, 2011.
KIRCHNER, F.; KOSMATOV, N.; PREVOSTO, V.; SIGNOLES, J.; YAKOBOWSKI, B. Frama-C: a Software Analysis Perspective. *Formal Aspects of Computing*, Springer, v. 27, n. 3, p. 573–609, 2015.
KLEIN, G. Proof Engineering Considered Essential. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, v. 8442 LNCS, p. 16–21, 2014.
KLEIN, G.; ELPHINSTONE, K.; HEISER, G.; ANDRONICK, J.; COCK, D.; DERRIN, P.; ELKADUWE, D.; ENGELHARDT, K.; KOLANSKI, R.; NORRISH, M.; SeWELL, T.; TUCH, H.; WINWOOD, S. seL4: Formal Verification of an OS Kernel. In: ACM. *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles.* 2009. p. 207–220.
KOZEN, D. Results on the Propositional μ-calculus. *tcs*, n. 27, p. 333–354, 1983.
KUPFERMAN, O.; VARDI, M. Y. Modular Model Checking. In: *Compositionality: The Significant Difference.* : Springer, 1998. p. 381–401.
LEINO, K. R. M. Dafny: An Automatic Program Verifier for Functional Correctness. In: *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning.* Berlin, Heidelberg: Springer-Verlag, 2010. (LPAR), p. 348–370. ISBN 3-642-17510-4, 978-3-642-17510-7.
LEINO, R. This is Boogie 2. June 2008. Disponível em: <https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/>.
LENZI, G. The modal μ-calculus: a survey. *TASK Quarterly*, v. 9, n. 3, p. 293–316, 2005.
LEROY, X. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. *ACM SIGPLAN Notices*, ACM, v. 41, n. 1, p. 42–54, 2006.
LÓPEZ, H. A.; MARQUES, E. R. B.; MARTINS, F.; NG, N.; SANTOS, C.; VASCONCELOS, V. T.; YOSHIDA, N. Protocol-based Verification of Message-passing Parallel Programs. In: ACM. *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications.*

2015. p. 280–298.
LUDÄSCHER, B.; ALTINTAS, I.; BERKLEY, C.; HIGGINS, D.; JAEGER, E.; JONES, M.; LEE, E. A.; TAO, J.; ZHAO, Y. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice and Experience*, Wiley Online Library, v. 18, n. 10, p. 1039–1065, 2006.
MALAWSKI, M.; BUBAK, M.; BAUDE, F.; CAROMEL, D.; HENRIO, L.; MOREL, M. Interoperability of Grid Component Models: GCM and CCA Case Study. In: *Towards Next Generation Grids.* : Springer, 2007. p. 95–105.
MANCINI, T.; MARI, F.; MASSINI, A.; MELATTI, I.; TRONCI, E. SyLVaaS: System Level Formal Verification as a Service. In: IEEE. *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 2015. p. 476–483.
MARCHÉ, C.; MOHRING, C. Paulin; URBAIN, X. The Krakatoa Tool for Certification of Java/Javacard Programs Annotated in JML. *The Journal of Logic and Algebraic Programming*, Elsevier, v. 58, n. 1, p. 89–106, 2004.
MARCHÉ, C.; TAFAT, A. *Weakest Precondition Calculus, Revisited using Why3*. 2012. 32 p. Disponível em: <https://hal.inria.fr/hal-00766171>.
MARCILON, T. B.; CARVALHO JUNIOR, F. H. de. Derivation and Verification of Parallel Components for the Needs of an HPC Cloud. In: *SBMF*. : Springer, 2013. (Lecture Notes in Computer Science, v. 8195), p. 51–66. ISBN 978-3-642-41070-3.
MCCANCE, J. A Brief Introduction to Modal Logic. 1999. Disponível em: <http://documents.kenyon.edu/math/McCance.pdf>.
MELL, P. M.; GRANCE, T. *SP 800-145. The NIST Definition of Cloud Computing*. Gaithersburg, MD, United States, 2011.
Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications and High Performance Computing*, v. 8, n. 3-4, p. 169–416, 1994.
MILNER, R. *Communication and Concurrency*. : Prentice hall New York etc., 1989. v. 84.
NANEVSKI, A.; MORRISETT, G.; SHINNAR, A.; GOVEREAU, P.; BIRKEDAL, L. Ynot: Dependent Types for Imperative Programs. In: ACM. *ACM Sigplan Notices*. 2008. v. 43, n. 9, p. 229–240.
NECULA, G. C. Proof-carrying Code. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 1997. (POPL), p. 106–119. ISBN 0-89791-853-3. Disponível em: <https://doi.acm.org/10.1145/263699.263712>.
NICKOLLS, J. GPU Parallel Computing Architecture and CUDA Programming Model. *Session on Multi-Core and Parallelism I, Hot Chips*, v. 19, p. 19–21, 2007.
NIPKOW, T.; WENZEL, M.; PAULSON, L. C. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002. ISBN 3-540-43376-7.
O'HEARN, P. W. A Primer on Separation Logic (and Automatic Program Verification and Analysis). *Software Safety and Security; Tools for Analysis and Verification. NATO Science for Peace and Security Series*, v. 33, p. 286–318, 2012.
OPENMP, O. A Proposed Industry Standard API for Shared Memory Programming. *OpenMP Architecture Review Board*, v. 27, 1997.
OWICKI, S.; GRIES, D. An Axiomatic Proof Technique for Parallel Programs I. *Acta informatica*, Springer, v. 6, n. 4, p. 319–340, 1976.
OWRE, S.; RUSHBY, J. M.; SHANKAR, N. PVS: A Prototype Verification System. In: *Proceedings of the 11th International Conference on Automated Deduction: Automated*

Deduction. London, UK, UK: Springer-Verlag, 1992. (CADE-11), p. 748–752. ISBN 3-540-55602-8.
PARROW, J. An Introduction to the π-calculus. In: BERGSTRA, J.; PONSE, A.; SMOLKA, S. (Ed.). *Handbook of Process Algebra*. : Elsevier, 2001.
PERVEZ, S.; GOPALAKRISHNAN, G.; KIRBY, R. M.; THAKUR, R.; GROPP, W. Formal Methods Applied to High-performance Computing Software Design: A Case Study of MPI One-sided Communication-based Locking. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 40, n. 1, p. 23–43, jan. 2010. ISSN 0038-0644. Disponível em: <http://dx.doi.org/10.1002/spe.v40:1>.
PETRI, C. A. *Kommunikation mit Automaten*. Tese (Doutorado) — Technische Hochschule Darmstadt, 1962.
PIERCE Benjamin C.; CASINGHINO Chris; GABOARDI Marco; GREENBERG Michael; HRITCU Catalin; SJOBERG Vilhelm; YORGEY Brent. *Software Foundations*. Electronic textbook, 2014. Disponível em: <http://www.cis.upenn.edu/~bcpierce/sf>.
PLIMPTON, S. J.; DEVINE, K. D. MapReduce in MPI for Large-scale Graph Algorithms. *Parallel Computing*, Elsevier, v. 37, n. 9, p. 610–632, 2011.
POST, D. E.; VOTTA, L. G. Computational Science Demands a New Paradigm. *Physics Today*, v. 58, n. 1, p. 35–41, 2005.
QIN, J.; FAHRINGER, T.; PLLANA, S. UML based Grid Workflow Modeling under ASKALON. In: *Proceedings of the Distributed and Parallel Systems: From Cluster to Grid Computing (DAPSYS)*. : Springer, 2006.
REISIG, W. *Petri Nets: An Introduction*. : sv, 1985. (EATCS Monographs on Theoretical Computer Science).
REYNOLDS, J. An Overview of Separation Logic. *Verified Software: Theories, Tools, Experiments*, Springer, p. 460–469, 2008.
REYNOLDS, J. C. Separation Logic: A Logic for Shared Mutable Data Structures. In: IEEE. *Logic in Computer Science. Proceedings. 17th Annual IEEE Symposium on*. 2002. p. 55–74.
REZENDE, C. A. d. *Um Arcabouço Baseado em Componentes para Computação Paralela de Larga Escala sobre Grafos*. 177 f. Tese (Doutorado em Ciência da Computação) — Programa de Pós-Graduação em Ciência da Computação (MDCC), Universidade Federal do Ceará, Fortaleza, 2017.
ROSCOE, A. W. *Understanding Concurrent Systems*. : Springer, 2010. (Texts in Computer Science).
SAMA, M.; ELBAUM, S.; RAIMONDI, F.; ROSENBLUM, D. S.; WANG, Z. Context-aware Adaptive Applications: Fault Patterns and Their Automated Identification. *IEEE Transactions on Software Engineering*, IEEE, v. 36, n. 5, p. 644–661, 2010.
SANNELLA, D.; TARLECKI, A. *Foundations of Algebraic Specifications and Formal Program Development*. : Cambridge University Press, 2011.
SCHAEFER, I.; SAUER, T. Towards Verification as a Service. In: SPRINGER. *International Workshop on Eternal Systems*. 2011. p. 16–24.
SENKUL, P.; KIFER, M.; TOROSLU, I. H. A Logical Framework for Scheduling Workflows under Resource Allocation Constraints. In: VLDB ENDOWMENT. *Proceedings of the 28th international conference on Very Large Data Bases*. 2002. p. 694–705.
SHAO, Z. Certified Software. *Communications of the ACM*, v. 53, n. 12, p. 56–66, 2010.
SIEGEL, S. F.; GOPALAKRISHNAN, G. Formal Analysis of Message Passing. In: *Proceedings of the 12th International Conference on Verification, Model Checking, and*

Abstract Interpretation. Berlin, Heidelberg: Springer-Verlag, 2011. (VMCAI), p. 2–18. ISBN 978-3-642-18274-7. Disponível em: <http://dl.acm.org/citation.cfm?id=1946284.1946286>.

SIEGEL, S. F.; ZHENG, M.; LUO, Z.; ZIRKEL, T. K.; MARIANIELLO, A. V.; EDENHOFNER, J. G.; DWYER, M. B.; ROGERS, M. S. CIVL: the Concurrency Intermediate Verification Language. In: ACM. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015. p. 61.

SIEGEL, S. F.; ZIRKEL, T. K. *The Toolkit for Accurate Scientific Software*. 2011.

SILVA, J. C.; CARVALHO JUNIOR, F. H. d. A Platform of Scientific Workflows for Orchestration of Parallel Components in a Cloud of High Performance Computing Applications. In: *Lecture Notes in Computer Science*: Proceedings of the XX Brazilian Symposium on Programming Languages (SBLP). : Springer, 2016. v. 9889, p. 156–170.

SILVA, J. d. C. *Um Arcabouço para a Construção de Aplicações Baseadas em Componentes sobre uma Plataforma de Nuvem Computacional para Serviços de Computação de Alto Desempenho*. 188 f. Tese (Doutorado em Ciência da Computação) — Programa de Pós-Graduação em Ciência da Computação (MDCC), Universidade Federal do Ceará, Fortaleza, 2016.

SØRENSEN, M. H.; URZYCZYN, P. *Lectures on the Curry-Howard Isomorphism*. : Elsevier, 2006. v. 149.

STEEN, A. J. van der. Issues in Computational Frameworks. *Concurrency and Computation: Practice and Experience*, Wiley, v. 18, n. 2, p. 141–150, 2006.

STERLING, T.; LUSK, E.; GROPP, W. (Ed.). *Beowulf Cluster Computing with Linux*. 2. ed. Cambridge, MA, USA: MIT Press, 2003. ISBN 0262692929.

SUKUMAR, K.; VECCHIOLA, C.; BUYYA, R. The Structure of the New IT Frontier: Aneka Platform for Elastic Cloud Computing Applications–Part III. *Manjrasoft Pty Ltd, Melbourne, Australia*, 2010.

TAYLOR, I. J.; DEELMAN, E.; GANNON, D. B.; SHIELDS, M. *Workflows for e-Science: Scientific Workflows for Grids*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN 1846285194.

The Coq development team. *The Coq Proof Assistant Reference Manual*. 2004. Version 8.0. Disponível em: <http://coq.inria.fr>.

TRETMANS, J. Model Based Testing with Labelled Transition Systems. In: *Formal methods and testing*. : Springer, 2008. p. 1–38.

VAKKALANKA, S.; DELISI, M.; GOPALAKRISHNAN, G.; KIRBY, R. M.; THAKUR, R.; GROPP, W. Implementing Efficient Dynamic Formal Verification Methods for MPI Programs. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. : Springer, 2008. p. 248–256.

VO, A.; VAKKALANKA, S.; DELISI, M.; GOPALAKRISHNAN, G.; KIRBY, R. M.; THAKUR, R. Formal Verification of Practical MPI Programs. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 44, n. 4, p. 261–270, fev. 2009. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/1594835.1504214>.

WANG, A. J. A.; QIAN, K. *Component-Oriented Programming*. : Wiley-Interscience, 2005.

WASSERMANN, B.; EMMERICH, W.; BUTCHART, B.; CAMERON, N.; CHEN, L.; PATEL, J. Sedna: A BPEL-Based Environment for Visual Scientific Workflow Modeling. Springer London, London, p. 428–449, 2007.

WIEDIJK Freek. Comparing Mathematical Provers. In: *Mathematical Knowledge*

Management, Second International Conference, MKM, Bertinoro, Italy, February 16-18, Proceedings. 2003. p. 188–202.
WINSKEL, G.; NIELSEN, M. Models for Concurrency. In: ABRAMSKY, S.; GABBAY, D. M.; MAIBAUM, T. S. E. (Ed.). *Handbook of Logic in Computer Science (vol. 4)*. : Oxford Science Publications, 1995. p. 1–148.
WODTKE, D.; WEIKUM, G. A formal Foundation for Distributed Workflow Execution Based on State Charts. In: SPRINGER. *International Conference on Database Theory*. 1997. p. 230–246.
WOLSTENCROFT, K.; HAINES, R.; FELLOWS, D.; WILLIAMS, A.; WITHERS, D.; OWEN, S.; REYES, S. Soiland; DUNLOP, I.; NENADIC, A.; FISHER, P.; BHAGAT, J.; BELHAJJAME, K.; BACALL, F.; HARDISTY, A.; HIDALGA, A. N. d. l.; VARGAS, M. P. B.; SUFI, S.; GOBLE, C. A. The Taverna Workflow Suite: Designing and Executing Workflows of Web Services on the Desktop, Web or in the Cloud. *Nucleic Acids Research*, v. 41, n. W1, p. W557, 2013.
ZASPEL, P.; GRIEBEL, M. Massively Parallel Fluid Simulations on Amazon's HPC Cloud. In: IEEE. *Network Cloud Computing and Applications (NCCA), First International Symposium on*. 2011. p. 73–78.

Appendices

A.1 Regras de Tradução do Algoritmo S2m

Este apêndice se destina a descrever as regras de tradução do algoritmo S2m. A abordagem consistirá em apresentar cada regra, que traduz uma tarefa SAFeSWL em um conjunto de processos, quem devem ser, em alguns pontos, sincronizados com alguns dos demais processos do sistema. Para cada regra, apresentará-se também um ou mais exemplos que ilustram sua aplicação.

Uma observação que se faz é que as ações computacionais dos componentes (ações que não pertencem à porta padrão *LifeCycle*) são genericamente representadas nas traduções mCRL2 feitas neste trabalho através de ações **compute** *componente ação*. Atualmente, na modelagem, os identificadores de componentes e ações são representados através de números inteiros, apesar de na gramática XSD de descrição arquitetural de SAFeSWL eles serem representados por *strings*. Apesar de o *toolset* mCRL2 oferecer suporte ao tipo *string*, optou-se por manter o tipo *Nat* no protótipo corrente, uma vez que no estudo de caso em que se realizou a tradução (Arquitetura de Processamento MapReduce, Capítulo 6), os componentes e suas ações computacionais são números inteiros. Em versões futuras do arcabouço de certificação, será realizada essa modificação.

Outra observação é que em códigos de orquestração descritos adiante que possuírem somente ações do tipo **compute**, omitem-se, por questões de espaço, as ativações das ações relativas ao ciclo de vida dos componentes (**resolve**, **deploy**, **instantiate** e **release**).

A.1.1 Ações e Sincronizações mCRL2

As ações (observáveis em negrito) e sincronizações que podem aparecer nas traduções são as seguintes:

- **resolve**, **deploy**, **instantiate**, **release**: ações do ciclo de vida dos componentes;
- **compute**, **guard**: ações computacionais e ações de guarda de tarefas select, respectivamente;
- $w\_act \mid c\_act \rightarrow$ **act**: uma ação **act** do ciclo de vida, computacional ou de guarda de um componente presente nas *regras de componente* do *workflow* interno desse componente é subdividida e sincronizada por uma ação que deve ocorrer em processos traduzidos a partir do *workflow* da aplicação (w_act) e por uma ação que deve ocorrer em processos que implementam as regras do *workflow* interno do componente (c_act);
- $pass \mid obtain \rightarrow$ **transfer**: a sincronização em **transfer**, de uma ação *pass* em um processo com uma ação *obtain* em outro processo, denota a passagem do fluxo de execução do primeiro para o segundo;
- $m\_repeat \mid r\_repeat \rightarrow$ **repeat**: a sincronização em **repeat**, de uma ação

m_repeat em um processo gerenciador de uma tarefa de iteração com uma ação r_repeat no processo que implementa o corpo do laço, faz com que o segundo seja executado novamente;

- $p\_break \mid m\_break \rightarrow$ **break**: a sincronização em **break**, de uma ação p_break em qualquer processo com uma ação m_break em um processo gerenciador de tarefa de iteração, faz com que o processo que implementa o corpo laço seja encerrado;
- $p\_continue \mid m\_continue \rightarrow$ **continue**: a sincronização em **continue**, de uma ação $p\_continue$ em qualquer processo com uma ação $m\_continue$ em um processo gerenciador de tarefa de iteração, faz com que o processo que implementa o corpo do laço volte ao início;
- $p\_add\_start \mid m\_add\_start \rightarrow$ **add_start**: a sincronização em **add_start**, de uma ação p_add_start em qualquer processo responsável por lançar uma ação assíncrona com uma ação m_add_start no processo gerenciador da lista de ações assíncronas, faz com que o *handle* dessa ação seja adicionado nessa lista;
- $s\_start\_fin \mid m\_start\_fin \rightarrow$ **start_fin**: a sincronização em **start_fin**, de uma ação s_start_fin em um processo que implementa uma ação ativada assincronamente com uma ação m_start_fin no processo gerenciador da lista de ações assíncronas, indica que a ação assíncrona foi concluída e que seu *handle* deve ser removido da lista;
- $p\_iterate\_start\_list \mid i\_iterate\_start\_list \rightarrow$ **iterate_start_list**: quando o processo principal (*workflow*) recebe de volta o fluxo de execução, antes de encerrar sua execução, ele solicita a um processo especial iterador que se comunique com o processo gerenciador da lista de ações assíncronas para iterar a lista e esperar por possíveis processos que implementam ações assíncronas que ainda não tenham sido concluídos. Esse procedimento é desencadeado a partir da sincronização em **iterate_start_list**, de $p\_iterate\_start\_list$ no processo *workflow* com $i\_iterate\_start\_list$ no processo iterador da lista de ações assíncronas;
- $i\_next\_start \mid m\_next\_start \rightarrow$ **next_start**: a sincronização em **next_start**, de i_next_start no processo iterador da lista de ações assíncronas com m_next_start no gerenciador dessa lista, faz com que o segundo espere o processo que implementa a próxima ação assíncrona da lista, caso este já não esteja concluído;
- $m\_empty\_list \mid i\_empty\_list \rightarrow$ **empty_list**: a sincronização em **empty_list**, de m_empty_list no processo gerenciador da lista de ações assíncronas com i_empty_list no processo iterador dessa lista, indica ao segundo que a lista se tornou vazia;
- $i\_iterate\_fin \mid w\_iterate\_fin \rightarrow$ **iterate_fin**: a sincronização em **iterate_fin**, de $i\_iterate\_fin$ no processo iterador da lista de ações assíncronas com $w\_iterate\_fin$ no processo *workflow*, diz ao segundo que a iteração foi concluída e que ele pode encerrar a computação;
- $p\_wait \mid m\_wait \rightarrow$ **wait**: a sinalização de que um processo quer esperar pelo

fim do processo que implementa uma ação assincronamente ativada é materializada pelo a sincronização em **wait**, de p_wait nesse processo com m_wait no processo gerenciador da lista de ações assíncronas;

- $m\_unlock \mid p\_unlock \rightarrow$ **unlock**: a sincronização em **unlock**, de m_unlock no processo gerenciador da lista de ações assíncronas com p_unlock em um processo que está esperando a conclusão do processo que implementa uma ação assíncrona, libera a execução do segundo;
- $p\_cancel \mid m\_cancel \rightarrow$ **cancel**: a sinalização de que um processo quer cancelar a ativação de uma ação assíncrona, através do encerramento do processo que a implementa, é materializada pela sincronização em **cancel**, de p_cancel no processo solicitante com m_cancel no processo gerenciador da lista de ações assíncronas;
- $m\_start\_cancel \mid s\_start\_cancel \rightarrow$ **start_cancel**: o gerenciador da lista de ações assíncronas pode cancelar um processo que implementa uma ação assíncrona quando há a sincronização em **start_cancel**, de sua ação m_start_cancel com a ação s_start_cancel no referido processo;
- $pass\_enable \mid obtain\_enable \rightarrow$ **enable**: na tradução, cada ação do lado direito de regras de componente dos *workflows* internos dos componentes possui um processo associado que recorda seu estado (habilitada/desabilitada). Dada uma regra $act_1 \rightarrow act_2\downarrow$ de um componente qualquer, a sincronização em **enable**, da ação $pass\_enable$ em um processo qualquer que executa act_1 com a ação $obtain\_enable$ no processo que gerencia o estado de act_2, torna act_2 habilitada;
- $pass\_disable \mid obtain\_disable \rightarrow$ **disable**: dada uma regra $act_1 \rightarrow act_2 \nmid$ de um componente qualquer, a sincronização em **disable**, da ação $pass\_disable$ em um processo qualquer que executa act_1 com a ação $obtain\_disable$ no processo que gerencia o estado de act_2, torna act_2 desabilitada.

A.1.2 Função de Tradução ($(\![\ ]\!)$)

Em linhas gerais, o algoritmo S2m recebe como entrada uma tarefa T_W, que corresponde ao *workflow* da aplicação, e um conjunto R, que consiste no conjunto fixo de regras de componentes dos *workflows* internos dos componentes orquestrados, e os repassa a uma *função de tradução* ($(\![\ ]\!)$), que retorna uma função que mapeia os nomes dos processos mCRL2 correspondentes à tradução da tarefa às suas definições. A função $(\![\ ]\!)$ é definida no estilo da programação funcional e, a depender do tipo de tarefa, executa uma das regras de tradução definidas mais adiante. Eis sua definição:

$$(\![T,\ R,\ C,\ i,\ i_r,\ i_b,\ i_c,\ \Gamma]\!) = \langle P,\ o,\ o_r,\ o_b,\ o_c,\ \Gamma_o \rangle$$

Entradas:

- T: uma tarefa SAFeSWL a ser traduzida recursivamente em processos mCRL2;
- R: o conjunto fixo de regras de componentes dos *workflows* internos dos componen-

tes presentes na orquestração;

- C: o nome do processo corrente (que está sendo criado no momento), no qual ações mCRL2 serão incorporadas. É relevante somente para a tradução de tarefas continue e break, uma vez que suas traduções fazem a execução do processo corrente ser interrompida;
- i: o número de ações de transferência de fluxo ($transfer$) criadas até o momento;
- i_r: o número de sincronizações de repetição ($repeat$) criadas até o momento;
- i_b: o número de sincronizações de fim de laço ($break$) criadas até o momento;
- i_c: o número de sincronizações de continuação de laço ($continue$) criadas até o momento;
- Γ: uma função que mapeia os nomes dos processos criados até o momento pela tradução às suas definições.

Saídas:

- P: uma definição de processo a ser incorporada no processo corrente;
- o: i atualizado pela tradução realizada;
- o_r: i_r atualizado pela tradução realizada;
- o_b: i_b atualizado pela tradução realizada;
- o_c: i_c atualizado pela tradução realizada;
- Γ_o: Γ atualizado pela tradução realizada.

A.1.3 Chamada do Algoritmo S2m

A chamada do algoritmo S2m para a tradução de um *workflow* SAFeSWL T_W é a seguinte:

$\mathsf{S2m}\ (T_W,\ R) =$

$\mathsf{let}\ (\![T_W,\ R,\ Workflow,\ 0,\ 0,\ 0,\ 0,\ \emptyset]\!) = \langle P,\ i,\ i_r,\ i_b,\ i_c,\ \Gamma_f \rangle$

$Def_1 = P.p\_iterate\_start\_list.w\_iterate\_fin.Workflow$

$Def_2 = \sum_h(m\_add\_start(h).$
$!(h\ in\ L) \rightarrow ManagerStartList([h]\ +\!+\ L) <>$
$m\_start\_fin(h).ManagerStartList(L)) +$
$\sum_h(m\_wait(h).m\_start\_fin(h).$
$m\_unlock(h).ManagerStartList(remove(L,h))) +$
$\sum_h(m\_cancel(h).(m\_start\_fin(h) +$
$m\_start\_cancel(h)).$
$ManagerStartList(remove(L,h)))) +$
$m\_next\_start.$
$\#L > 0 \rightarrow m\_start\_fin(head(L)).$
$p\_iterate\_start\_list.ManagerStartList(tail(L)) <>$
$m\_empty\_list.ManagerStartList([\,])$

$Def_3 = (i\_iterate\_start\_list.i\_next\_start +$
$i\_empty\_list.i\_iterate\_fin).IteratorStartList$

$\mathsf{in}\ \Gamma_f \cup \{Workflow \mapsto Def_1,\ ManagerStartList(L : List(Nat)) \mapsto Def_2,\ IteratorStartList \mapsto I$
$\cup\ ProcActRightSide(R)$

Como se pode ver, o algoritmo cria no mínimo três processos padrão, a saber:

- $Workflow$: recebe a definição Def_1, que corresponde à tradução recursiva do *workflow*, seguida, respectivamente, de um comando para que o processo iterador da lista de ações assíncronas ($IteratorStartList$) inicie a iteração que espera por todos os processos que implementam ações assíncronas ainda não concluídos ($p\_iterate\_start\_list$) e da recepção de um comando do iterador ($w\_iterate\_fin$) dizendo que a lista já foi iterada e que a computação pode ser encerrada;
- $ManagerStartList$: recebe a definição Def_2, que corresponde a um processo que gerencia a lista ações ativadas assincronamente. Ao receber uma solicitação de adição de um *handle* h de uma ação na lista ($m\_add\_start(h)$), se o *handle* não existe, ele é adicionado. Caso já exista o *handle* na lista, a ação já foi ativada assincronamente antes e ainda está pendente (seu processo ainda não foi concluído). Nesse caso, o gerenciador espera que o processo que implementa a ação termine ($m\_start\_fin(h)$) e mantém o *handle* na lista. Caso receba uma solicitação de um processo que deseja esperar por uma ação assíncrona de *handle* h ($m\_wait(h)$), mantém o processo em espera até que a ação seja concluída ($m\_start\_fin(h)$) e então o libera ($m\_unlock(h)$), removendo o *handle* da lista. Caso receba uma solicitação de cancelamento de uma ativação assíncrona de uma ação de *handle* h ($m\_cancel(h)$), não faz nada, caso o processo que implementa a ação já esteja concluído ($m\_start\_fin(h)$), ou o cancela ($m\_start\_cancel(h)$), caso contrário;

- *IteratorStartList*: recebe a definição Def_3, que corresponde a um processo que é responsável por iterar a lista de ações ativadas assincronamente, ao final da execução do *workflow*. Ao receber o comando do *workflow* para iniciar a iteração ($i\_iterate\_start\_list$), se comunica iterativamente com o gerenciador da lista (i_next_start) até que todos os processos que implementam ações assíncronas estejam concluídos (i_empty_list). Por fim, sinaliza ao *workflow* que a iteração foi concluída ($i\_iterate\_fin$).

A função *ProcActRightSide* é uma abstração que cria, para cada ação que aparece do lado direito de regras de componente em R, um único processo (de fato, um mapeamento de nome de processo a definição de processo) que será encarregado de gerenciar o estado da ação (habilitada/desabilitada). Por questões didáticas, concentrarão-se todos os detalhes de tradução relativos à composição do *workflow* da aplicação com os *workflows* internos dos componentes na seção A.1.10.

Existe ainda um processo padrão S (sistema), que é chamado pelo ambiente mCRL2 e tem a função de lançar paralelamente todos os processos que virão a compor o LTS. O processo S deve conter ainda todas as sincronizações de ações do sistema, bem como deve definir quais as ações devem ser observáveis pelo sistema. Embora S inicie todos os processos paralelamente, somente o $Workflow$ iniciará desbloqueado, podendo passar em um momento posterior o fluxo de execução a outro processo. Uma observação importante que se faz é que, por questões de legibilidade, nos códigos mCRL2 descritos nesta Tese, serão apresentados somente ações, sincronizações e processos diretamente envolvidos nas respectivas traduções.

A geração dos processos padrão cria o LTS descrito na Figura 35. Os passos posteriores do algoritmo incrementarão este programa, possivelmente incluindo novos processos, ações e pontos de sincronização.

A.1.4 Tarefa de Ativação Síncrona de Ação

A regra de tradução de uma tarefa de ativação síncrona de ação SAFeSWL (invoke) é a seguinte:

$$[\![\text{invoke } a,\ R,\ C,\ i,\ i_r,\ i_b,\ i_c,\ \Gamma]\!] = \langle a,\ i,\ i_r,\ i_b,\ i_c,\ \Gamma \rangle$$

Como se pode ver, essa tarefa é modelada em mCRL2 simplesmente através da criação de uma ação. A exemplo dessa regra, suponha um código de orquestração que somente executa uma tarefa invoke da ação computacional identificada por 1 do componente identificado por 1. A tradução deste código para mCRL2 ficaria como é mostrado na Figura 36.

A.1.5 Tarefa de Sequenciamento de Tarefas

A regra de tradução de uma tarefa de sequenciamento de tarefas (sequence) é a seguinte:

Figura 35: Geração dos processos padrão pelo algoritmo S2m

```
map remove: List(Nat) # Nat -> List(Nat);
var x, y: Nat;
    l: List(Nat);
eqn remove([],x) = [];
    x == y -> remove(x |> l, y) = l;
    x != y -> remove(x |> l, y) = x |> remove(l, y);

act resolve, deploy, instantiate, release: Nat;
act compute, guard: Nat # Nat;
act pass, obtain, transfer: Nat;
act m_repeat, r_repeat, repeat: Nat;
act p_break, m_break, break: Nat;
act p_continue, m_continue, continue: Nat;
act p_add_start, m_add_start, add_start: Nat;
act s_start_fin, m_start_fin, start_fin: Nat;
act p_iterate_start_list, i_iterate_start_list, iterate_start_list;
act i_next_start, m_next_start, next_start;
act m_empty_list, i_empty_list, empty_list;
act i_iterate_fin, w_iterate_fin, iterate_fin;
act p_wait, m_wait, wait: Nat;
act m_unlock, p_unlock, unlock: Nat;
act p_cancel, m_cancel, cancel: Nat;
act m_start_cancel, s_start_cancel, start_cancel: Nat;
act pass_enable, obtain_enable, enable: Nat;
act pass_disable, obtain_disable, disable: Nat;


proc Workflow = p_iterate_start_list.w_iterate_fin.Workflow;


proc ManagerStartList (L:List(Nat)) =
                sum h:Nat.(m_add_start(h).
                    (!(h in L) -> ManagerStartList([h]++L) <> m_start_fin(h).ManagerStartList(L))) +
                sum h:Nat.(m_wait(h).m_start_fin(h).m_unlock(h).ManagerStartList(remove(L,h))) +
                sum h:Nat.(m_cancel(h).(m_start_fin(h) + m_start_cancel(h)).
                    ManagerStartList(remove(L,h))) +
                m_next_start.
                    ((#L > 0) -> m_start_fin(head(L)).p_iterate_start_list.ManagerStartList(tail(L)) <>
                    m_empty_list.ManagerStartList([]));

proc IteratorStartList = (i_iterate_start_list.i_next_start + i_empty_list.i_iterate_fin).IteratorStartList;


proc S = allow({resolve, deploy, instantiate, release, compute, guard, transfer, repeat, break, continue,
                add_start, start_fin, iterate_start_list, next_start, empty_list, iterate_fin, wait,
                unlock, cancel, start_cancel, enable, disable},
         comm ({pass | obtain -> transfer,
                m_repeat | r_repeat -> repeat,
                p_break | m_break -> break,
                p_continue | m_continue -> continue,
                p_add_start | m_add_start -> add_start,
                s_start_fin | m_start_fin -> start_fin,
                p_iterate_start_list | i_iterate_start_list -> iterate_start_list,
                i_next_start | m_next_start -> next_start,
                m_empty_list | i_empty_list -> empty_list,
                i_iterate_fin | w_iterate_fin -> iterate_fin,
                p_wait | m_wait -> wait,
                m_unlock | p_unlock -> unlock,
                p_cancel | m_cancel -> cancel,
                m_start_cancel | s_start_cancel -> start_cancel,
                pass_enable | obtain_enable -> enable,
                pass_disable | obtain_disable -> disable},
         Workflow||ManagerStartList([])||IteratorStartList));
init S;
```

Fonte: Elaborado pelo autor.

```
act compute: Nat # Nat;

proc Workflow = compute(1,1).Workflow;

proc S = allow({compute},
         Workflow));
init S;
```

Figura 36: Tradução de uma tarefa de ativação síncrona (invoke) pelo algoritmo S2m
Fonte: Elaborado pelo autor.

Figura 37: Um código de orquestração com a tarefa sequence

```
<sequence>
        <invoke action="resolve" id_port="life-cycle-1" />
        <invoke action="deploy" id_port="life-cycle-1" />
        <invoke action="instantiate" id_port="life-cycle-1" />
</sequence>
```

Fonte: Elaborado pelo autor.

Figura 38: O código mCRL2 gerado pelo algoritmo S2m para o código da Figura 37 (sequence)

```
act resolve, deploy, instantiate: Nat;

proc Workflow = resolve(1).deploy(1).instantiate(1).Workflow;

proc S = allow({resolve, deploy, instantiate},
         comm ({},
         Workflow);
init S;
```

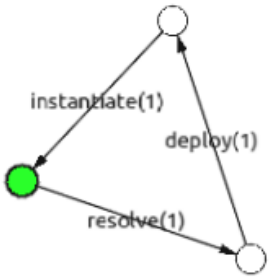
Fonte: Elaborado pelo autor.

$$
\begin{aligned}
&(\![\text{sequence } T1\ T2,\ R,\ C,\ i,\ i_r,\ i_b,\ i_c,\ \Gamma]\!) = \\
&\quad \text{let } (\![T1,\ R,\ C,\ i,\ i_r,\ i_b,\ i_c,\ \Gamma]\!) \qquad\qquad = \langle P_1,\ i_{T1},\ i_{T1_r},\ i_{T1_b},\ i_{T1_c},\ \Gamma_{T1}\rangle \\
&\qquad\quad (\![T2,\ R,\ C,\ i_{T1},\ i_{T1_r},\ i_{T1_b},\ i_{T1_c},\ \Gamma_{T1}]\!) = \langle P_2,\ i_{T2},\ i_{T2_r},\ i_{T2_b},\ i_{T2_c},\ \Gamma_{T2}\rangle \\
&\quad \text{in } \langle P_1.P_2,\ i_{T2},\ i_{T2_r},\ i_{T2_b},\ i_{T2_c},\ \Gamma_{T2}\rangle
\end{aligned}
$$

Como se pode ver, a tradução dessa tarefa se dá pela tradução recursiva da primeira tarefa, seguida da tradução recursiva da segunda tarefa, passando como argumento os valores das variáveis que foram obtidos na tradução da primeira. O resultado final é concatenação das duas definições de processos obtidas e os valores das variáveis retornadas na tradução da segunda tarefa.

A título de ilustração, considere o código de orquestração da Figura 37, a qual realiza três invocações síncronas em sequência. O código mCRL2 resultante é visto na Figura 38. Para fins de compreensão, mostra-se, na Figura 39, o grafo LTS gerado a partir do código mCRL2 da Figura 38.

Figura 39: Grafo LTS para o código mCRL2 da Figura 38 (sequence)



Fonte: Elaborado pelo autor.

Figura 40: Um código de orquestração com a tarefa select

```
<select>
    <choice action="1" id_port="compute-99" >
            <sequence>
                    <invoke action="1" id_port="compute-1" />
                    <invoke action="2" id_port="compute-2" />
            </sequence>
    </choice>
    <choice action="2" id_port="compute-99" >
            <invoke action="1" id_port="compute-3" />
    </choice>
</select>
```

Fonte: Elaborado pelo autor.

Figura 41: O código mCRL2 gerado pelo algoritmo S2m para o código da Figura 40 (select)

```
act compute, guard: Nat # Nat;

proc Workflow = (guard(99,1).compute(1,1).compute(2,1) + guard(99,2).compute(3,1)).Workflow;

proc S = allow({compute, guard},
         comm ({},
         Workflow));
init S;
```

Fonte: Elaborado pelo autor.

A.1.6 Tarefa de Seleção de Tarefas

A regra de tradução de uma tarefa de seleção de tarefas (select) é a seguinte:

$$
\begin{aligned}
&(\!|\text{select } g1 : T1\ g2 : T2,\ R,\ C,\ i,\ i_r,\ i_b,\ i_c,\ \Gamma|\!) = \\
&\quad \text{let } (\!|T1,\ R,\ C,\ i,\ i_r,\ i_b,\ i_c,\ \Gamma|\!) = \langle P_1,\ i_{T1},\ i_{T1_r},\ i_{T1_b},\ i_{T1_c},\ \Gamma_{T1}\rangle \\
&\qquad\quad (\!|T2,\ R,\ C,\ i_{T1},\ i_{T1_r},\ i_{T1_b},\ i_{T1_c},\ \Gamma_{T1}|\!) = \langle P_2,\ i_{T2},\ i_{T2_r},\ i_{T2_b},\ i_{T2_c},\ \Gamma_{T2}\rangle \\
&\quad \text{in } \langle g1.P_1 + g2.P_2,\ i_{T2},\ i_{T2_r},\ i_{T2_b},\ i_{T2_c},\ \Gamma_{T2}\rangle
\end{aligned}
$$

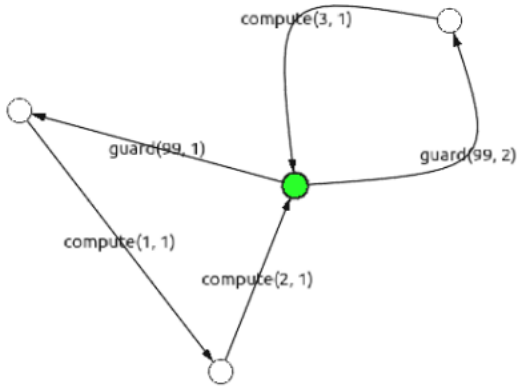
Como se pode ver, a tradução dessa tarefa segue o mesmo princípio da tarefa de sequenciamento, contudo utilizando um comportamento alternativo (operador "+" do mCRL2) entre as duas definições de processos obtidas nas traduções recursivas e utilizando guardas nelas. Como exemplo de um programa SAFeSWL que utiliza tarefas de seleção, considere a Figura 40, a qual realiza uma tarefa select com duas escolhas possíveis. Nesse exemplo, escolhe-se uma dentre as duas tarefas (uma sequence ou uma invoke), de acordo com qual ação de guarda, 1 ou 2, de uma porta de computação qualquer do componente 99, se tornou ativada no componente.

Nas figuras 41 e 42 mostram-se, respectivamente, o código gerado pelo S2m para o exemplo da Figura 40 e o grafo LTS associado ao código mCRL2 gerado.

A.1.7 Tarefa de Paralelismo de Tarefas

A regra de tradução de uma tarefa de paralelismo de tarefas (parallel) é a seguinte:

Figura 42: Grafo LTS para o código mCRL2 da Figura 41 (select)



Fonte: Elaborado pelo autor.

$$
\begin{aligned}
&(\![\text{parallel } T1\ T2,\ R,\ C,\ i,\ i_r,\ i_b,\ i_c,\ \Gamma]\!) = \\
&\quad \text{let } P_f && = (pass(i+1) + pass(i+2)).\\
&\qquad && \quad (pass(i+1) + pass(i+2)).\\
&\qquad && \quad (obtain(i+3) + obtain(i+4)).\\
&\qquad && \quad (obtain(i+3) + obtain(i+4))\\
&\qquad (\![T1,\ R,\ Parallel_1,\ i+4,\ i_r,\ i_b,\ i_c,\ \Gamma]\!) && = \langle P_1,\ i_{T1},\ i_{T1_r},\ i_{T1_b},\ i_{T1_c},\ \Gamma_{T1}\rangle\\
&\qquad (\![T2,\ R,\ Parallel_2,\ i_{T1},\ i_{T1_r},\ i_{T1_b},\ i_{T1_c},\ \Gamma_{T2}]\!) && = \langle P_2,\ i_{T2},\ i_{T2_r},\ i_{T2_b},\ i_{T2_c},\ \Gamma_{T2}\rangle\\
&\qquad Def_1 && = obtain(i+1).P_1.pass(i+3).Parallel_1\\
&\qquad Def_2 && = obtain(i+2).P_2.pass(i+4).Parallel_2\\
&\qquad \Gamma_f && = \Gamma_{T2} \cup \{Parallel1 \mapsto Def_1,\ Parallel2 \mapsto Def_2\\
&\quad \text{in } \langle P_f,\ i_{T2},\ i_{T2_r},\ i_{T2_b},\ i_{T2_c},\ \Gamma_f\rangle
\end{aligned}
$$

Como se pode ver, essa regra cria os processos $Parallel1$ e $Parallel2$, cujas definições são as traduções recursivas das tarefas $T1$ e $T2$. O processo corrente então transfere o fluxo de execução paralelamente aos dois processos, respectivamente, através das ações $pass(i+1)$ e $pass(i+2)$, e espera o retorno desses dois fluxos, através das ações $obtain(i+3)$ e $obtain(i+4)$, respectivamente, em um modelo *fork-join*. Note que a definição de processo a ser incorporada no processo corrente (P_f) utiliza a alternativa ("+") para simular um paralelismo na passagem e retorno dos fluxos de execução nos dois processos.

Como exemplo de tradução de um código contendo essa tarefa, observe a Figura 44, que é o código traduzido a partir do código SAFeSWL da Figura 43, e a Figura 45, a qual representa o grafo LTS resultante.

Figura 43: Um código de orquestração com a tarefa parallel

```
<parallel>
        <sequence>
                <invoke action="1" id_port="compute-1" />
                <invoke action="1" id_port="compute-2" />
        </sequence>
        <invoke action="1" id_port="compute-3" />
</parallel>
```

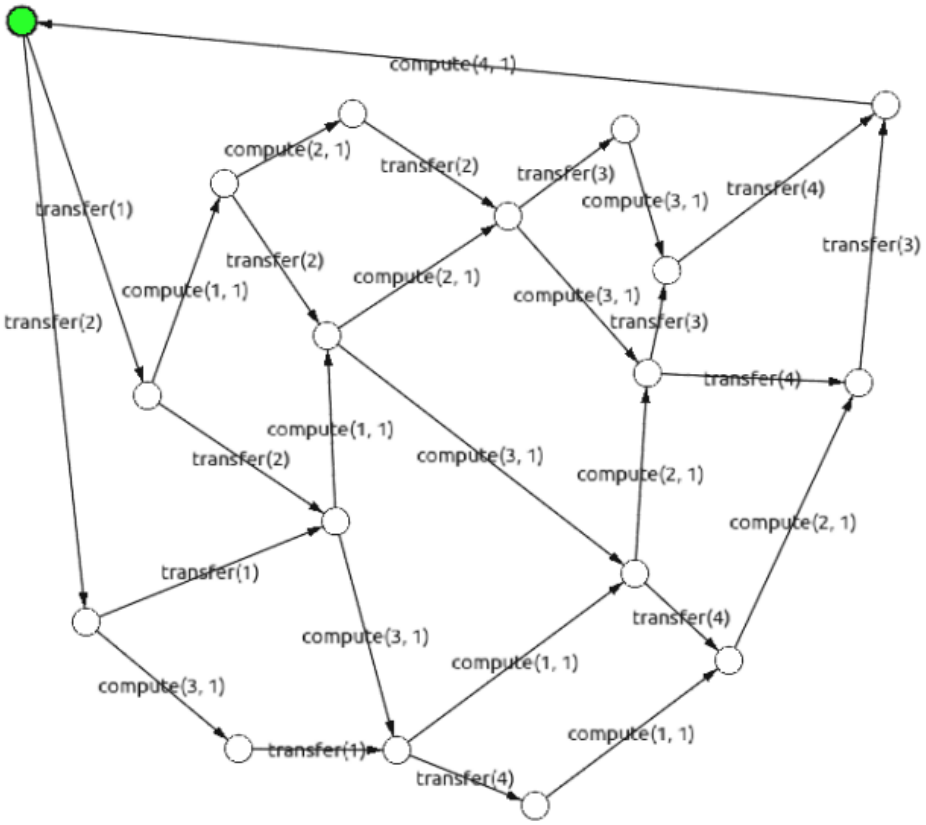
Fonte: Elaborado pelo autor.

Figura 44: O código mCRL2 gerado pelo algoritmo S2m para o código da Figura 43 (parallel)

```
act compute: Nat # Nat;
act pass, obtain, transfer: Nat;

proc Workflow = sum n:Nat.(n == 1 || n == 2) -> pass(n).sum n:Nat.(n == 1 || n == 2) -> pass(n).sum m:Nat.(m == 3 || m == 4) ->
                obtain(m).sum m:Nat.(m == 3 || m == 4) -> obtain(m).compute(4,1).Workflow;

proc Parallel1 = obtain(1).compute(1,1).compute(2,1).pass(3).Parallel1;

proc Parallel2 = obtain(2).compute(3,1).pass(4).Parallel2;

proc S = allow({compute, transfer},
         comm ({pass | obtain -> transfer},
         Workflow||Parallel1||Parallel2));
init S;
```

Fonte: Elaborado pelo autor.

Figura 45: Grafo LTS para o código mCRL2 da Figura 44 (parallel)



Fonte: Elaborado pelo autor.

A.1.8 Tarefas de Iteração de Tarefa, Continuação de Iteração de Tarefa e Fim de Iteração de Tarefa

As regras de tradução de uma tarefa de iteração de tarefa (repeat), de uma tarefa de continuação de iteração de tarefa (continue) e de uma tarefa de fim de iteração de tarefa (break) estão fortemente relacionadas e serão apresentadas ao mesmo tempo nessa seção.

A regra de tradução de uma tarefa de iteração de tarefa (repeat) é como se segue:

$$
\begin{aligned}
&(\![\textsf{repeat}\ T,\ R,\ C,\ i,\ i_r,\ i_b,\ i_c,\ \Gamma]\!) = \\
&\quad \textsf{let}\ P_f && = m\_repeat(i_r+1).obtain(i+2) \\
&\qquad (\![T,\ R,\ Repeat,\ i+2,\ i_r+1,\ i_b+1,\ i_c+1,\ \Gamma]\!) && = \langle P_T,\ i_T,\ i_{T_r},\ i_{T_b},\ i_{T_c},\ \Gamma_T\rangle \\
&\qquad Def_1 && = r\_repeat(i_r+1).P_T.pass(i+1).Repeat \\
&\qquad Def_2 && = (obtain(i+1).m\_repeat(i_r+1)+ \\
&\qquad\qquad && \quad m\_break(i_b+1).pass(i+2)+ \\
&\qquad\qquad && \quad m\_continue(i_c+1).m\_repeat(i_r+1)). \\
&\qquad\qquad && \quad ManagerRepeat \\
&\qquad \Gamma_f && = \Gamma_T \cup \{Repeat \mapsto Def_1,\ ManagerRepeat \mapsto \\
&\quad \textsf{in}\ \langle P_f,\ i_T,\ i_{T_r},\ i_{T_b},\ i_{T_c},\ \Gamma_f\rangle
\end{aligned}
$$

Como se pode ver, essa regra cria dois processos, *Repeat* e $ManagerRepeat$. O processo *Repeat* implementa o corpo do laço (tradução recursiva da tarefa T armazenada em P_T) e, ao receber o comando do processo corrente (primeira vez) ou do gerenciador da tarefa de iteração (nova iteração), através da ação $r\_repeat(i_r+1)$, executa o corpo do laço. Se a execução do corpo do laço chegar ao fim, isso significa que nenhum break ou continue foi detectado e o fluxo de execução deve ser transferido de volta para o gerenciador ($pass(i+1)$). O processo $ManagerRepeat$ é o gerenciador da tarefa de iteração sendo traduzida. Ele, ao receber um fluxo de execução normal ($obtain(i+1)$) ou uma sinalização de continue ($m\_continue(i_c+1)$), comanda uma nova execução e, ao receber um uma sinalização de break ($m\_break(i_b+1)$), devolve o fluxo de execução ao processo corrente.

A regra de tradução de uma tarefa de continuação de iteração de tarefa (continue) é definida como:

$$(\![\textsf{continue},\ C,\ i,\ i_r,\ i_b,\ i_c,\ \Gamma]\!) = \langle pass\_continue(i_c).C,\ i,\ i_r,\ i_b,\ i_c,\ \Gamma\rangle$$

Como se pode ver, essa regra diz somente que, no processo corrente, ao ser detectado um continue, ele sinaliza isso ao gerenciador da tarefa de iteração ($pass\_continue(i_c)$) e encerra sua execução imediatamente.

A regra de tradução de uma tarefa de fim de iteração de tarefa (break) é assim definida:

$$(\![\textsf{break},\ C,\ i,\ i_r,\ i_b,\ i_c,\ \Gamma]\!) = \langle pass\_break(i_b).C,\ i,\ i_r,\ i_b,\ i_c,\ \Gamma\rangle$$

De forma análoga à tarefa de continuação de iteração de tarefa, essa regra

Figura 46: Um código de orquestração com a tarefa repeat

```
<repeat>
        <sequence>
                <select>
                    <choice action="1" id_port="compute-99" >
                            <parallel>
                                    <invoke action="1" id_port="compute-1" />
                                    <invoke action="1" id_port="compute-2" />
                            </parallel>
                    </choice>
                    <choice action="2" id_port="compute-99" >
                            <break />
                    </choice>
                    <choice action="3" id_port="compute-99" >
                            <continue />
                    </choice>
                </select>
                <invoke action="1" id_port="compute-3" />
        </sequence>
</repeat>
```

Fonte: Elaborado pelo autor.

Figura 47: O código mCRL2 gerado pelo algoritmo S2m para o código da Figura 46 (repeat)

```
act compute, guard: Nat # Nat;
act pass, obtain, transfer: Nat;
act m_repeat, r_repeat, repeat: Nat;
act p_break, m_break, break: Nat;
act p_continue, m_continue, continue: Nat;

proc Workflow = m_repeat(1).obtain(2).Workflow;

proc Repeat1 = r_repeat(1).(guard(99,1).sum n:Nat. (n==3 || n==4) -> pass(n).sum n:Nat. (n==3 || n==4) -> pass(n).
               sum m:Nat.(m==5 || m==6) -> obtain(m).sum m:Nat.(m==5 || m==6) -> obtain(m) + guard(99,2).p_break(1).Repeat1 +
               guard(99,3).p_continue(1).Repeat1).compute(3,1).pass(1).Repeat1;

proc ManagerRepeat1 = (obtain(1).m_repeat(1)+ m_break(1).pass(2)+ m_continue(1).m_repeat(1)).ManagerRepeat1;

proc Parallel1 = obtain(3).compute(1,1).pass(5).Parallel1;

proc Parallel2 = obtain(4).compute(2,1).pass(6).Parallel2;

proc S = allow({compute, guard, transfer, repeat, break, continue},
         comm ({pass | obtain -> transfer,
                m_repeat | r_repeat -> repeat,
                p_break | m_break -> break,
                p_continue | m_continue -> continue},
        Workflow||Repeat1||ManagerRepeat1||Parallel1||Parallel2));
init S;
```

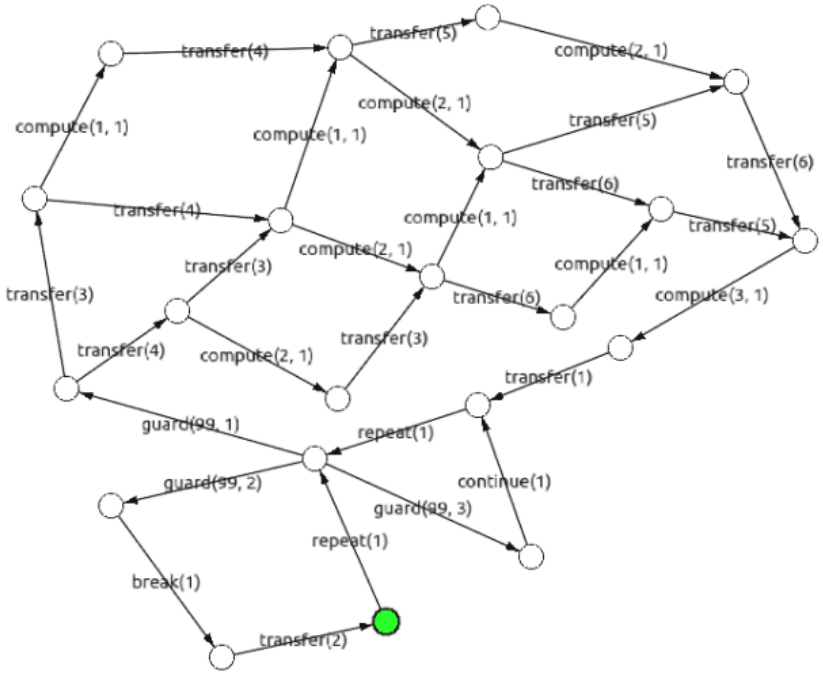
Fonte: Elaborado pelo autor.

de tradução diz apenas que o processo corrente, ao detectar um break, deve sinalizar isso ao gerenciador da tarefa de iteração ($pass\_break(i_b)$) e encerrar imediatamente sua execução.

Para exemplificar essas traduções, observe a Figura 46. Nela, é executada uma tarefa repeat, que itera uma tarefa de sequenciamento. Essa tarefa de sequenciamento primeiro executa uma tarefa de escolha depois executa a ação **compute** 3 1 sincronamente. A tarefa de escolha pode lançar dois processos em paralelo para executarem as ações **compute** 1 1 e **compute** 2 1, sincronamente, executar uma tarefa break ou executar uma tarefa continue.

A Figura 47 apresenta o código gerado pelo algoritmo S2m para o código SAFeSWL da Figura 46. A Figura 48, por sua vez, apresenta o grafo LTS gerado para o código traduzido.

Figura 48: Grafo LTS para o código mCRL2 da Figura 47 (repeat)



Fonte: Elaborado pelo autor.

A.1.9 Tarefas de Ativação Assíncrona de Ação, Espera por Ação Assincronamente Ativada e Cancelamento de Ativação Assíncrona de Ação

As traduções da tarefa de ativação assíncrona de ação (start), da tarefa de cancelamento de ativação assíncrona de ação (cancel) e da tarefa de espera por ação assincronamente ativada (wait) estão inter-relacionadas e serão descritas ao mesmo tempo nessa seção.

A regra de tradução de uma tarefa de ativação assíncrona de ação SAFeSWL (start) é a seguinte:

$$
\begin{aligned}
&(\!|\textsf{start}\ h\ a,\ R,\ C,\ i,\ i_r,\ i_b,\ i_c,\ \Gamma|\!) = \\
&\quad \textsf{let}\ Def = obtain(i+1).(a.s\_start\_fin(h)\ +\ p\_start\_cancel(h)).Start \\
&\qquad \Gamma_f\ \ = \Gamma \cup \{Start \mapsto Def\} \\
&\quad \textsf{in}\ \langle p\_add\_start(h).pass(i+1),\ i+1,\ i_r,\ i_b,\ i_c,\ \Gamma_f\rangle
\end{aligned}
$$

Como se pode ver, essa regra cria um processo, $Start$, que implementa a ativação assíncrona da ação a. Esse processo somente recebe assincronamente (sem retorno) o fluxo de execução do processo corrente ($obtain(i+1)$) e, ou executa a ação, ou é cancelado. O processo corrente fica encarregado, respectivamente, de avisar ao gerenciador da lista de ações assíncronas que irá ativar assincronamente a ação ($p\_add\_start(h)$) e ativá-la ($pass(i+1)$).

A Figura 49 apresenta um exemplo de orquestração SAFeSWL que realiza a escolha de uma entre três ações para ser ativada assincronamente. A tradução desse código para mCRL2 é mostrada na Figura 50. Já a Figura 51, por sua vez, apresenta o

Figura 49: Um código de orquestração com a tarefa start

```
<select>
    <choice action="1" id_port="compute-99" >
            <start action="1" id_port="compute-1" handle_id="1" />
    </choice>
    <choice action="2" id_port="compute-99" >
            <start action="1" id_port="compute-2" handle_id="2" />
    </choice>
    <choice action="3" id_port="compute-99" >
            <start action="1" id_port="compute-3" handle_id="3" />
    </choice>
</select>
```

Fonte: Elaborado pelo autor.

Figura 50: O código mCRL2 gerado pelo algoritmo S2m para o código da Figura 49 (start)

```
act compute, guard: Nat # Nat;
act pass, obtain, transfer: Nat;
act p_add_start, m_add_start, add_start: Nat;
act s_start_fin, m_start_fin, start_fin: Nat;
act p_iterate_start_list, i_iterate_start_list, iterate_start_list;
act i_next_start, m_next_start, next_start;
act m_empty_list, i_empty_list, empty_list;
act i_iterate_fin, w_iterate_fin, iterate_fin;

proc Workflow = (guard(99,1).p_add_start(1).pass(1) + guard(99,2).p_add_start(2).pass(2) +
                guard(99,3).p_add_start(3).pass(3)).p_iterate_start_list.w_iterate_fin.Workflow;

ManagerStartList (L:List(Nat)) =
                 sum h:Nat.(m_add_start(h).
                    (!(h in L) -> ManagerStartList([h]++L) <> m_start_fin(h).ManagerStartList(L))) +
                m_next_start.
                    ((#L > 0) -> m_start_fin(head(L)).p_iterate_start_list.ManagerStartList(tail(L)) <>
                     m_empty_list.ManagerStartList([]));

IteratorStartList = (i_iterate_start_list.i_next_start + i_empty_list.i_iterate_fin).IteratorStartList;

Start1 = obtain(1).compute(1,1).s_start_fin(1).Start1;

Start2 = obtain(2).compute(2,1).s_start_fin(2).Start2;

Start3 = obtain(3).compute(3,1).s_start_fin(3).Start3;

proc S = allow({transfer, compute, guard, add_start, start_fin, iterate_start_list, next_start,
                empty_list, iterate_fin},
         comm ({pass | obtain -> transfer,
                p_add_start | m_add_start -> add_start,
                s_start_fin | m_start_fin -> start_fin,
                p_iterate_start_list | i_iterate_start_list -> iterate_start_list,
                i_next_start | m_next_start -> next_start,
                m_empty_list | i_empty_list -> empty_list,
                i_iterate_fin | w_iterate_fin -> iterate_fin},
         Workflow||ManagerStartList([])||IteratorStartList||Start1||Start2||Start3));
init S;
```

Fonte: Elaborado pelo autor.

grafo LTS gerado para o código traduzido.

A regra que descreve a tradução de uma tarefa de espera pela conclusão de uma ação assincronamente ativada (wait) é a seguinte:

$$[\![\text{wait } h,\ C,\ i,\ i_r,\ i_b,\ i_c,\ \Gamma]\!] = \langle p\_wait(h).p\_unlock(h),\ i,\ i_r,\ i_b,\ i_c,\ \Gamma\rangle$$

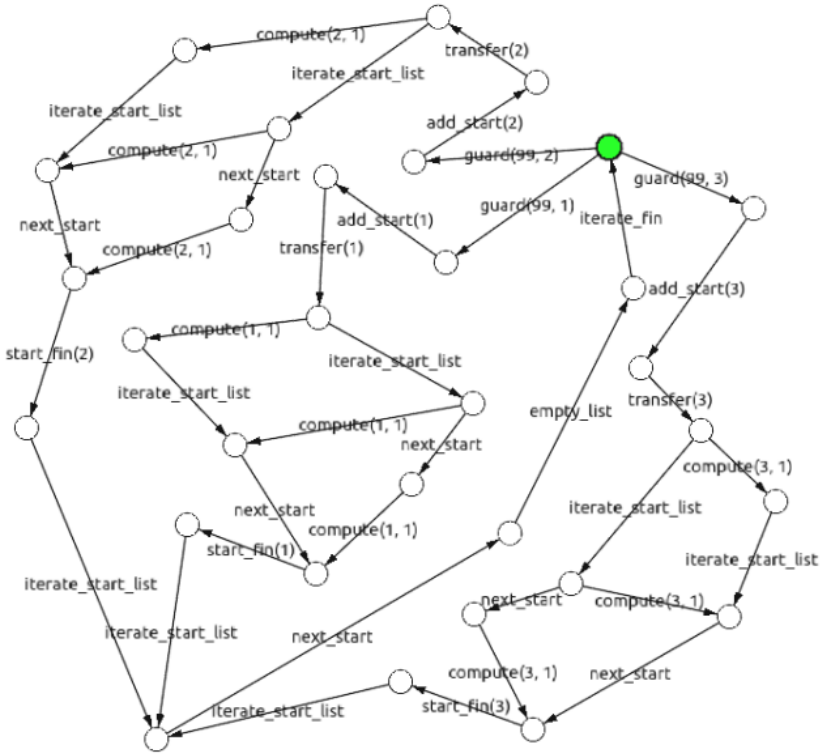
A partir dessa regra, vê-se que o processo corrente, ao solicitar a espera pela conclusão de uma ação assincronamente ativada ($p\_wait(h)$), fica bloqueado até que o gerenciador da lista de ações assíncronas o libere ($p\_unlock(h)$).

A regra que implementa a tradução de uma tarefa de cancelamento de ativação de ação assíncrona (cancel) é definida como se segue:

$$[\![\text{cancel } h,\ C,\ i,\ i_r,\ i_b,\ i_c,\ \Gamma]\!] = \langle p\_cancel(h),\ i,\ i_r,\ i_b,\ i_c,\ \Gamma\rangle$$

Como pode ser visto, o processo corrente somente sinaliza ao gerenciador da lista de ações assíncronas que quer cancelar uma ação de *handle* h ($p\_cancel(h)$).

Figura 51: Grafo LTS para o código mCRL2 da Figura 50 (start)



Fonte: Elaborado pelo autor.

Figura 52: Um código de orquestração com as tarefas start, wait e cancel

```
<sequence>
        <start action="1" id_port="compute-1" handle_id="1" />
        <select>
            <choice action="1" id_port="compute-99" >
                    <wait handle_id="1" />
            </choice>
            <choice action="2" id_port="compute-99" >
                    <cancel handle_id="1" />
            </choice>
        </select>
</sequence>
```

Fonte: Elaborado pelo autor.

Observe a Figura 52, a qual contém um código de orquestração SAFeSWL que executa uma tarefa start seguida de uma tarefa que espera a conclusão da execução do start ou o cancela. A tradução desse código para mCRL2 é mostrada na Figura 53. Na Figura 54, é apresentado o grafo LTS gerado para o código traduzido.

A.1.10 *Workflows* Internos dos Componentes

Nessa seção, reunem-se todos os detalhes da tradução com relação à composição do *workflow* da aplicação com os *workflows* internos dos componentes.

Primeiramente, será definida precisamente a função $ProcActRightSide$. Dado um conjunto de regras de componente $R = \{R_1,\ R_2, ...,\ R_n\}$, essa função analisa em sequência cada regra R_i e realiza os procedimentos a seguir:

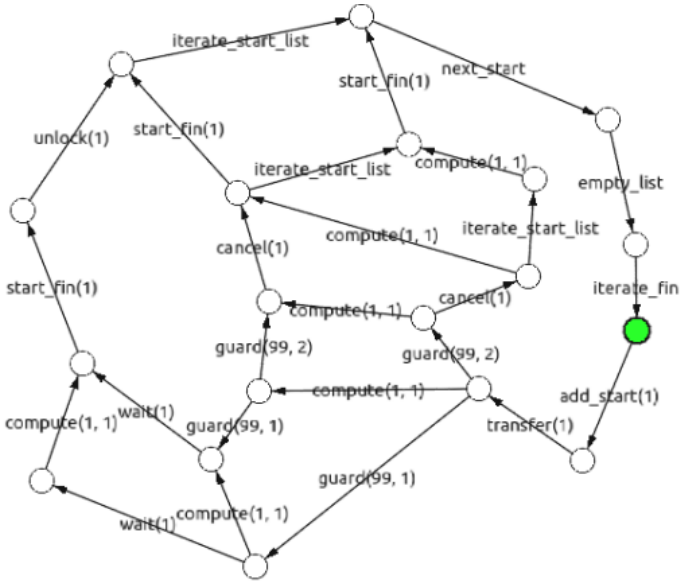
1. Caso ela ainda não tenha criado um processo (mapeamento de nome de processo

Figura 53: O código mCRL2 gerado pelo algoritmo S2m para o código da Figura 52 (start, wait e cancel)

```
map remove: List(Nat) # Nat -> List(Nat);
var x, y: Nat;
    l: List(Nat);
eqn remove([],x) = [];
    x == y -> remove(x |> l, y) = l;
    x != y -> remove(x |> l, y) = x |> remove(l, y);

act compute, guard: Nat # Nat;
act pass, obtain, transfer: Nat;
act p_add_start, m_add_start, add_start: Nat;
act s_start_fin, m_start_fin, start_fin: Nat;
act p_iterate_start_list, i_iterate_start_list, iterate_start_list;
act i_next_start, m_next_start, next_start;
act m_empty_list, i_empty_list, empty_list;
act i_iterate_fin, w_iterate_fin, iterate_fin;
act p_wait, m_wait, wait: Nat;
act m_unlock, p_unlock, unlock: Nat;
act p_cancel, m_cancel, cancel: Nat;
act m_start_cancel, s_start_cancel, start_cancel: Nat;

proc Workflow = p_add_start(1).pass(1).(guard(99,1).p_wait(1).p_unlock(1) + guard(99,2).p_cancel(1)).
                p_iterate_start_list.w_iterate_fin.Workflow;


proc ManagerStartList (L:List(Nat)) =
                sum h:Nat.(m_add_start(h).
                    (!(h in L) -> ManagerStartList([h]++L) <> m_start_fin(h).ManagerStartList(L))) +
                sum h:Nat.(m_wait(h).m_start_fin(h).m_unlock(h).ManagerStartList(remove(L,h))) +
                sum h:Nat.(m_cancel(h).(m_start_fin(h) + m_start_cancel(h)).
                    ManagerStartList(remove(L,h))) +
                m_next_start.
                    ((#L > 0) -> m_start_fin(head(L)).p_iterate_start_list.ManagerStartList(tail(L)) <>
                    m_empty_list.ManagerStartList([]));

proc IteratorStartList = (i_iterate_start_list.i_next_start + i_empty_list.i_iterate_fin).IteratorStartList;

proc Start1 = obtain(1).(compute(1,1).s_start_fin(1) + s_start_cancel(1)).Start1;

proc S = allow({transfer, compute, add_start, start_fin, iterate_start_list, next_start,
              empty_list, iterate_fin, wait, unlock, cancel, guard},
         comm ({pass | obtain -> transfer,
                p_add_start | m_add_start -> add_start,
                s_start_fin | m_start_fin -> start_fin,
                p_iterate_start_list | i_iterate_start_list -> iterate_start_list,
                i_next_start | m_next_start -> next_start,
                m_empty_list | i_empty_list -> empty_list,
                i_iterate_fin | w_iterate_fin -> iterate_fin,
                p_wait | m_wait -> wait,
                m_unlock | p_unlock -> unlock,
                p_cancel | m_cancel -> cancel,
                m_start_cancel | s_start_cancel -> start_cancel},
         Workflow||ManagerStartList([])||IteratorStartList||Start1));
init S;
```

Fonte: Elaborado pelo autor.

Figura 54: Grafo LTS para o código mCRL2 da Figura 53 (start, wait e cancel)



Fonte: Elaborado pelo autor.

Figura 55: Um código de orquestração de componentes que possuem *workflows* internos

```
<repeat>
        <sequence>
                <invoke action="A1" id_port="compute-C1" />
                <select>
                    <choice action="A2" id_port="compute-C1" >
                            <invoke action="A1" id_port="compute-C2" />
                    </choice>
                    <choice action="A2" id_port="compute-C2" >
                            <break />
                    </choice>
                </select>
        </sequence>
</repeat>
```

Fonte: Elaborado pelo autor.

a definição de processo) para gerenciar o *status* **habilitada/desabilitada** da ação do lado direito da regra, diga-se act_2, ela o faz, criando um mapeamento do nome da ação precedido da letra "P" a uma definição de processo que contém somente o nome do processo ($Pact_2 \mapsto Pact_2$);

2. Caso a regra seja da forma $act_1 \rightarrow act_2\downarrow$, a função modifica a definição do processo gerenciador do *status* da ação act_2 ($Pact_2$), acrescentando à esquerda "$obtain\_enable(act_1).c\_ac$ indicando que, para que a ação mCRL2 c_act_2 possa ser executada, ela primeiro deve ter sido liberada por act_1. Note que a ação act_2, quando aparecer em algum processo derivado da tradução da tarefa do *workflow* da aplicação, deve ser trocada por w_act_2, para que haja a sincronização com c_act_2, no processo que gerencia o *status* de act_2;
3. Caso a regra seja da forma $act_1 \rightarrow act_2\not\downarrow$, a função modifica a definição do processo gerenciador do *status* da ação act_2, acrescentando à esquerda "$obtain\_disable(act_1).Pact_2 +$", indicando que, quando act_1 for executada, c_act_2 não pode acontecer, por estar desabilitada;
4. Caso a regra seja da forma $\top \rightarrow act_2\downarrow$, a função desconsidera a definição do processo gerenciador do *status* da ação act_2 calculada até o momento e mantém somente a ação c_act_2 ("$c\_act_2.Pact_2$"), indicando que essa ação está sempre habilitada. Mais ainda, a função não observa mais regras em que a ação act_2 está do lado direito.

Alterações são necessárias nas regras de tradução das tarefas invoke e start para que, no caso em que uma ação a a ser traduzida habilite ou desabilite outra ação em uma regra de componente do conjunto R, após essa ação seja acrescentado $pass\_enable(a)$ ou $pass\_disable(a)$, respectivamente.

A título de ilustração, considere a Figura 55. Nela, há um código comum de um *workflow* de aplicação SAFeSWL. No entanto, a nível de programação dos componentes $C1$ e $C2$, é garantido que as ações $A1$ desses componentes estão sempre habilitadas e que suas execuções habilitam as respectivas guardas $A2$. Esses componentes ofertam esses *workflows* internos nos seus catálogos no Core na seguinte forma:

$$W_{C1} \;=\; W_{C2} \;= \{\top \rightarrow A1\downarrow,\; A1 \rightarrow A2\downarrow\}$$

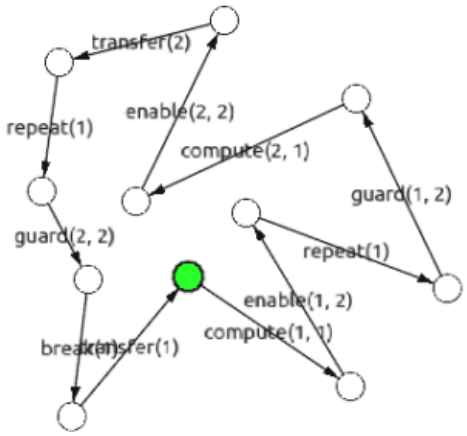
Na Figura 56, apresenta-se o código resultante da tradução da composição do

Figura 56: O código mCRL2 gerado pelo algoritmo S2m para o *workflow* de aplicação da Figura 55 composto com os *workflows* internos dos componentes

```
act compute:Nat # Nat;
act w_guard, c_guard, guard: Nat # Nat;
act pass, obtain, transfer: Nat;
act m_repeat, r_repeat, repeat: Nat;
act p_break, m_break, break:Nat;
act pass_enable, obtain_enable, enable: Nat # Nat;


proc Workflow = compute(1,1).pass_enable(1,2).m_repeat(1).obtain(1).Workflow;

proc ManagerRepeat1 = (obtain(2).m_repeat(1) + m_break(1).pass(1)).ManagerRepeat1;

proc Repeat1 = r_repeat(1).(w_guard(1,2).compute(2,1).pass_enable(2,2) +
               w_guard(2,2).p_break(1).Repeat1).pass(2).Repeat1;

proc P_C1_G = obtain_enable(1,2).c_guard(1,2).P_C1_G;

proc P_C2_G = obtain_enable(2,2).c_guard(2,2).P_C2_G;

proc S = allow({compute, guard, transfer, repeat, break, enable},
         comm ({w_guard | c_guard -> guard,
                pass | obtain -> transfer,
                m_repeat | r_repeat -> repeat,
                p_break | m_break -> break,
                pass_enable | obtain_enable -> enable},
         Workflow||ManagerRepeat1||Repeat1||P_C1_G||P_C2_G));
init S;
```

Fonte: Elaborado pelo autor.

Figura 57: Grafo LTS para o código mCRL2 da Figura 56 (*workflows* internos dos componentes)



Fonte: Elaborado pelo autor.

workflow de aplicação da Figura 55 com os referidos *workflows* internos dos componentes $C1$ e $C2$ (note as ações w_guard e c_guard).

Na Figura 57, é apresentado o grafo LTS referente ao código mCRL2 da Figura 56.

B.2 Códigos SAFeSWL da Arquitetura de Processamento MapReduce

Neste apêndice, são apresentados os códigos arquitetural e de orquestração do estudo de caso referente à arquitetura de processamento MapReduce.

B.2.1 Descrição Arquitetural

A Figura 58 mostra a descrição arquitetural do sistema de computação paralela da aplicação MapReduce. As linhas de 0 a 7 mostram o código referente aos componentes repositórios data_source e data_sink, respectivamente. Esses componentes expõem apenas uma única porta provedora, as quais terão como usuário o componente splitter, detalhado logo em seguida (linhas de 10 a 32). O componente splitter apresenta três portas de serviço usuárias, duas para se conectar aos repositórios e uma para se conectar a algum componente do tipo computação para receber a lista de *chunks*. Além disso, trata-se do único componente que apresenta três portas de ações de tipos diferentes (os demais possuem duas). Pela figura, podem ser notadas as ações de cada uma dessas portas. A *tag* <contract> indica a localização do arquivo que representa o contrato contextual para o componente splitter. Para os outros componentes da arquitetura (reducer, mapper, combiner e shuffler), o código de descrição arquitetural é análogo.

As linhas entre 36 e 51 mostram o trecho do código arquitetural referente ao componente workflow, destacando a definição de suas portas de ações, as quais serão conectadas, através de *bindings* definidos posteriormente, às demais portas de ações dos outros componentes. Por exemplo, o componente workflow se conecta às três portas de ações do componente splitter, através das portas *LifeCycle_splitter*, *Task_source_splitter* e *Task_chunk_splitter*. O mesmo vale para os outros componentes, com portas de ações dos tipos *LifeCycle* e *Task_chunk*. Desta forma, é possível que o *workflow* orquestre suas ações. As portas provedoras *SAFe* e *Go* servem, respectivamente, para o *workflow* receber parâmetros de execução da aplicação e ter sua execução iniciada.

As linhas entre 53 e 65 mostram o trecho do arquivo arquitetural relativo às conexões das portas de serviços do componente splitter. Nesse caso, o componente splitter é usuário dos dois repositórios e também do componente reducer. A porta usuária *Source_uses_splitter* conecta-se à porta provedora de data_source, para ler os dados de entrada. Por sua vez, a porta *Sink_uses_splitter* conecta-se à porta provedora do componente data_sink, para salvar os dados resultantes do processamento. Finalmente, a porta usuária *Chunk_uses_splitter* é usada para leitura de uma lista de *chunks* providos por algum outro componente do *workflow*, notadamente o componente reducer, a fim de iniciar uma nova iteração do processamento MapReduce.

Por fim, as linhas entre 66 e 77 apresentam o trecho da linguagem arquitetural

que relaciona as portas de ações do componente splitter com o componente workflow. Esse relacionamento habilita que o componente workflow para orquestrar as ações das portas *LifeCycle_splitter*, *Task_source_splitter* e *Task_chunk_splitter*. Para os demais componentes, as conexões das portas de serviços e de ações seguem um padrão similar.

B.2.2 Descrição da Orquestração

A orquestração dos componentes de computação e conectores MapReduce, descrita na Figura 59, resumidamente realiza as seguintes atividades. Após a resolução, implantação e instanciação de todos os componentes, a ação **read_source**, de splitter, é ativada, fazendo com que os dados de entrada sejam lidos de data_source. Em seguida, a ação **perform**, de splitter, é ativada, gerando os *chunks* e os distribuindo entre o agente de mapeamento (para esse exemplo, existe somente um agente de mapeamento e um de redução). Após essa computação inicial, um laço é iniciado, o qual escolhe dentre os componentes, o que primeiro possuir um *chunk* pronto (ação **chunk_ready**).

A iteração se repete até que a ação **terminate**, de splitter, seja ativada. Quando isso ocorre, os dados são enviados para data_sink, concluindo o processamento MapReduce. Por fim, os componentes são liberados das respectivas plataformas virtuais.

Figura 58: Código arquitetural SAFeSWL do *workflow* de Processamento MapReduce

```
<!-- repositories -->
<repository name="data_source" id_comp="200" >
    <provides_port id_port="data-src-prov" />
</repository>

<repository name="data_sink" id_comp="201" >
    <provides_port id_port="data-sk-prov" />
</repository>

<!-- connectors -->
<connector name="splitter" id_comp="3" >
    <uses_port id_port="source-uses-splitter" />
    <uses_port id_port="sink-uses-splitter" />
    <uses_port id_port="chunk-uses-splitter" />
    <provides_port id_port="chunk-prov-splitter" />
    <action_port id_port="life-cycle-splitter" > <!-- default life-cycle port -->
        <action id_act="resolve" />
        <action id_act="deploy" />
        <action id_act="instantiate" />
        <action id_act="release" />
    </action_port>
    <action_port id_port="task-source-splitter" >
        <action id_act="341" name="read_source" />
        <action id_act="342" name="terminate" />
        <action id_act="343" name="write_sink" />
    </action_port>
    <action_port id_port="task-chunk-splitter" >
        <action id_act="351" name="read_chunk"/>
        <action id_act="352" name="perform"/>
        <action id_act="353" name="chunk_ready"/>
    </action_port>
    <contract path="/home/safe-user/contracts/splitter_contract.cc" />
</connector>

...

<!-- workflow -->
<workflow name="workflow-mr" id_comp="1">
    <provides_port id_port="safe" />
    <provides_port id_port="go" />
    <action_port id_port="life-cycle-splitter" />
    <action_port id_port="life-cycle-shuffler" />
    <action_port id_port="life-cycle-mapper" />
    <action_port id_port="life-cycle-reducer" />
    <action_port id_port="life-cycle-combiner" />
    <action_port id_port="task-source-splitter" />
    <action_port id_port="task-chunk-splitter" />
    <action_port id_port="task-chunk-shuffler" />
    <action_port id_port="task-chunk-mapper" />
    <action_port id_port="task-chunk-reducer" />
    <action_port id_port="task-chunk-combiner" />
</workflow>

<!-- splitter bindings -->
<service_binding>
    <uses_port id_port="source-uses-splitter" id_comp="3" />
    <provides_port id_port="data-src-prov" id_comp="200" />
</service_binding>
<service_binding>
    <uses_port id_port="sink-uses-splitter" id_comp="3" />
    <provides_port id_port="data-sk-prov" id_comp="201" />
</service_binding>
<service_binding>
    <uses_port id_port="chunk-uses-splitter" id_comp="3" />
    <provides_port id_port="red_prov_ck" id_comp="4" />
</service_binding>
<action_binding>
    <action_port id_port="life-cycle-splitter" id_comp="1" />
    <right_peer id_port="life-cycle-splitter" id_comp="3" />
</action_binding>
<action_binding>
    <action_port id_port="task-source-splitter" id_comp="1" />
    <right_peer id_port="task-source-splitter" id_comp="3" />
</action_binding>
<action_binding>
    <action_port id_port="task-chunk-splitter" id_comp="1" />
    <right_peer id_port="task-chunk-splitter" id_comp="3" />
</action_binding>

...
```

Fonte: Elaborado pelo autor.

Figura 59: Código de orquestração SAFeSWL do *workflow* de Processamento MapReduce

```
<sequence>
    <invoke action="resolve" id_port="life-cycle-splitter" /> <!--splitter.resolve-->
    <invoke action="deploy" id_port="life-cycle-splitter" /> <!--splitter.deploy-->
    <invoke action="instantiate" id_port="life-cycle-splitter" /> <!--splitter.instantiate-->
    <invoke action="resolve" id_port="life-cycle-shuffler" /> <!--shuffler.resolve-->
    <invoke action="deploy" id_port="life-cycle-shuffler" /> <!--shuffler.deploy-->
    <invoke action="instantiate" id_port="life-cycle-shuffler" /> <!--shuffler.instantiate-->
    <invoke action="resolve" id_port="life-cycle-mapper" /> <!--mapper.resolve-->
    <invoke action="deploy" id_port="life-cycle-mapper" /> <!--mapper.deploy-->
    <invoke action="instantiate" id_port="life-cycle-mapper" /> <!--mapper.instantiate-->
    <invoke action="resolve" id_port="life-cycle-reducer" /> <!--reducer.resolve-->
    <invoke action="deploy" id_port="life-cycle-reducer" /> <!--reducer.deploy-->
    <invoke action="instantiate" id_port="life-cycle-reducer" /> <!--reducer.instantiate-->
    <invoke action="resolve" id_port="life-cycle-combiner" /> <!--combiner.resolve-->
    <invoke action="deploy" id_port="life-cycle-combiner" /> <!--combiner.deploy-->
    <invoke action="instantiate" id_port="life-cycle-combiner" /> <!--combiner.instantiate-->
    <invoke action="341" id_port="task-source-splitter" /> <!--splitter.read_source-->
    <invoke action="352" id_port="task-chunk-splitter" /> <!--splitter.perform-->
    <repeat>
        <select>
            <choice action="353" id_port="task-chunk-splitter" > <!--splitter.chunk_ready-->
                <sequence>
                    <invoke action="551" id_port="task-chunk-mapper" /> <!--mapper.read_chunk-->
                    <invoke action="552" id_port="task-chunk-mapper" /> <!--mapper.perform-->
                </sequence>
            </choice>
            <choice action="553" id_port="task-chunk-mapper" > <!--mapper.chunk_ready-->
                <sequence>
                    <invoke action="1051" id_port="task-chunk-combiner" /> <!--combiner.read_chunk-->
                    <invoke action="1052" id_port="task-chunk-combiner" /> <!--combiner.perform-->
                </sequence>
            </choice>
            <choice action="1053" id_port="task-chunk-combiner" > <!--combiner.chunk_ready-->
                <sequence>
                    <invoke action="451" id_port="task-chunk-shuffler" /> <!--shuffler.read_chunk-->
                    <invoke action="452" id_port="task-chunk-shuffler" /> <!--shuffler.perform-->
                </sequence>
            </choice>
            <choice action="453" id_port="task-chunk-shuffler" > <!--shuffler.chunk_ready-->
                <sequence>
                    <invoke action="651" id_port="task-chunk-reducer" /> <!--reducer.read_chunk-->
                    <invoke action="652" id_port="task-chunk-reducer" /> <!--reducer.perform-->
                </sequence>
            </choice>
            <choice action="653" id_port="task-chunk-reducer" > <!--reducer.chunk_ready-->
                <sequence>
                    <invoke action="351" id_port="task-chunk-splitter" /> <!--splitter.read_chunk-->
                    <invoke action="352" id_port="task-chunk-splitter" /> <!--splitter.perform-->
                </sequence>
            </choice>
            <choice action="342" id_port="task-source-splitter" > <!--splitter.terminate-->
                <sequence>
                    <invoke action="343" id_port="task-source-splitter" /> <!--splitter.write_sink-->
                    <break/>
                </sequence>
            </choice>
        </select>
    </repeat>
    <invoke action="release" id_port="life-cycle-splitter" /> <!--splitter.release-->
    <invoke action="release" id_port="life-cycle-shuffler" /> <!--shuffler.release-->
    <invoke action="release" id_port="life-cycle-mapper" /> <!--mapper.release-->
    <invoke action="release" id_port="life-cycle-reducer" /> <!--reducer.release-->
    <invoke action="release" id_port="life-cycle-combiner" /> <!--combiner.release-->
</sequence>
```

Fonte: Elaborado pelo autor.