# Em Direção a um Modelo para Desenvolvimento de Sistemas Computationais de Qualidade para Aplicações Onivalentes

**Álvaro Moreira, Érika Cota, Leila Ribeiro, Luciano Gaspary, Luigi Carro, Marcus Ritt, Taisy Weber**

[1]Instituto de Informática
Universidade Federal do Rio Grande do Sul (UFRGS)

{afmoreira,erika,leila,paschoal,carro,mrpritt,taisy}@inf.ufrgs.br

***Resumo.*** *Aplicações computationais atuais e do futuro são marcadas, com uma freqüencia crescente, por características como distribuição, dinamicidade e ubiqüidade. O desenvolvimento de sistemas computacionais de qualidade para aplicações com essas características, chamadas aqui de* aplicações onivalentes*, demanda uma combinação de requisitos muitas vezes conflitantes ou dependentes entre si. Requisitos de dependabilidade, correção, segurança, escalabilidade, e evolutividade fazem parte das necessidades das aplicações onivalentes. Neste artigo discutimos os conflitos gerados pela integração desses diferentes requisitos em um mesmo sistema, e esboçamos as necessidades de um modelo de desenvolvimento de software de qualidade para aplicações onivalentes.*

***Abstract.*** *Distribution, Dynamicity and Ubiquity are becoming prevalent characteristics in computing applications. The development of high quality computational system for applications that have these characteristics, called* omnivalent applications *here, demands a combination of requirements, that are often conflicting or interdependent. For example, dependability, correctness, security, scalability and evolutionability are extremely relevant requirements of omnivalent applications. In this paper, we discuss the conflicts generated by the integration of these requirements in a single computational system and sketch the needs of a software development method suitable for developing high quality omnivalent applications.*

## 1. Introdução

Sistemas computacionais estão cada vez mais intrusivos nas nossas vidas. Inicialmente restritos a grandes instituições, as pessoas tinham contato somente com as listagens decorrentes das execuções dos programas em grandes máquinas. Na medida em que seus componentes foram se tornando mais baratos e com maior velocidade, novos sistemas computacionais foram surgindo e nossa dependência a eles foi aumentando acentuadamente (computadores de mesa, controladores programáveis para várias aplicações industriais, computadores embarcados nos carros, nos celulares e em inúmeros outros dispositivos cotidianos). Para o futuro, prevê-se uma realidade onde processadores estarão ainda mais integrados ao cotidiano de maneira quase transparente, idealmente nos informando, nos apoiando e nos protegendo ou, no pior cenário, nos enganando e prejudicando.

Em algumas aplicações, essa dependência não é crítica, apesar de potencialmente perturbadora. Por exemplo, se um sistema de correio eletrônico não funcionar durante

um dia, haverá muito incômodo, mas como o sistema é assíncrono por natureza, espera-se que algumas horas de atraso não sejam mais que apenas um incômodo. Em outras aplicações, como controle de tráfego aéreo, um alto grau de confiança nos sistemas computacionais é essencial. Essas aplicações críticas são cada vez mais freqüentes, e mesmo para aplicações não consideradas críticas um grau mínimo de qualidade é necessário, pois os usuários estão se mostrando cada vez mais exigentes. Em uma imagem futurística muito provável, todos os carros serão equipados com sistemas de piloto automático, que evitarão colisões através de sensores e comunicação com outros veículos enquanto os passageiros lêem as últimas notícias e compram ações. Falha de um controlador autônomo pode levar a desastres em cascata, mesmo em aplicações não críticas não correlacionadas.

Os sistemas computacionais estão se tornando altamente distribuídos, compostos de inúmeros componentes autônomos que se comunicam através de sinais e protocolos. Neste cenário, há diversos problemas relativos a falhas tanto de software quanto de hardware, segurança, confiabilidade do sistema como um todo, configuração dinâmica e não conhecida a priori, componentes heterogêneos de inúmeros fabricantes (muitos sem especificação conhecida ou comportamento previsível).

A questão que abordaremos neste artigo é: quais os desafios que a comunidade de computação deve enfrentar para construir sistemas distribuídos, heterogêneos, escaláveis, construídos a partir de componentes imperfeitos, mas robustos o suficiente para que possamos realmente confiar no serviço provido por esses sistemas?

Cada sistema computacional tem um conjunto diferente de requisitos que devem ser satisfeitos, dependente da aplicação ao qual esse sistema se destina. Os requisitos normalmente são classificados em funcionais (relativos ao comportamento esperado do sistema - do tipo “dadas entradas, o sistema deve gerar as saídas esperadas”) e não-funcionais (usabilidade, disponibilidade, escalabilidade, tempo de resposta, preço, segurança, etc).

Alguns requisitos são essenciais para as aplicações distribuídas, dinâmicas e ubíquas: dependabilidade, correção, segurança, escalabilidade e evolutividade. Os métodos existentes para desenvolvimento de sistemas computacionais complexos não atingem o objetivo de garantir qualidade na prática, geralmente em nenhum desses requisitos. Há abordagens teóricas, mas estas normalmente impõem tantas restrições que acabam não sendo viáveis. Um grave problema da maioria dos métodos que garantem, por exemplo, correção ou confiabilidade, é a escalabilidade: eles só são aplicáveis a sistemas extremamente pequenos, o que obviamente está em grave conflito com o tamanho e as demandas dos sistemas reais já atualmente em operação e com potencial de continuar seu crescimento acelerado nas próximas décadas. A integração desses requisitos em um mesmo sistema é um problema ainda maior, pois quase nunca é possível tratá-los separadamente pelo alto grau de interdependência entre eles, como será visto a seguir.

Para que se possam tratar requisitos diferentes, dependentes e/ou conflitantes, o primeiro passo é conhecer a fundo as implicações da inclusão de cada característica em um sistema, e, principalmente, as interdependências existentes entre as características que se deseja que um sistema possua. Cada uma dessas características apresenta desafios, problemas em abertos e empecilhos para sua plena utilização em sistemas atuais, não totalmente solucionados e, algumas, vezes sequer totalmente identificados.

Neste artigo pretende-se analisar e identificar interdependências entre os requisi-

tos de dependabilidade, correção, segurança, escalabilidade e evolutividade para garantir a qualidade dos sistemas computacionais do futuro, que estarão presentes no cotidiano. Pretende-se também esboçar idéias de como tratar essas relações e conflitos de forma integrada já nas fases iniciais do projeto do sistema e apresentar os desafios que precisam ser vencidos para a construção de sistemas distribuídos autônomos escaláveis. Seguramente a solução para o tratamento adequado desses requisitos passa por um esforço de reunificação das várias especialidades da computação, aproveitando de cada área sua experiência, suas tentativas mal sucedidas e suas soluções promissoras.

## 2. Características de Aplicações Onivalentes

O termo onivalente foi cunhado pelos pesquisadores que definiram os desafios da computação para a Sociedade Brasileira de Computação. Sistemas onivalentes, na visão de futuro da comunidade de Informática, estarão presentes em todos os ambientes e atividades humanas prestando serviços essenciais para a saúde, educação, informação, comunicação, produção e entretenimento preservando a integridade do ambiente social, tecnológico e natural. Tais sistemas têm por características distribuição, dinamicidade e ubiqüidade. Para atender as expectativas de qualidade de seus inúmeros usuários, precisam garantir da melhor forma possível requisitos de correção, dependabilidade, segurança, escalabilidade e evolutividade.

Um sistema distribuído é formado por múltiplos nós computacionais, sejam computadores, dispositivos móveis ou embutidos com capacidade de interagir por troca de mensagens. Pode ser caracterizado por não possuir uma memória global e nem um relógio global. Em um sistema distribuído dinâmico, os nós não são conhecidos a priori, mas podem entrar e sair da computação a qualquer momento seja pela demanda de outros nós, por desejo próprio ou por apresentarem defeito. Se o sistema for autônomo, os nós interagem para estabelecer a configuração necessária para prover um dado serviço. Casos contrário, os nós necessitam ser pré-configurados ou inseridos manualmente na configuração.

Um sistema ubíquo ou pervasivo é um sistema que se estende em qualquer ambiente da vida humana [Weiser 1993]. Um exemplo pode ser uma coleção de sistemas embarcados, conectados por uma rede sem fio, que controla todo ambiente dentro uma casa [Davidoff et al. 2006]. Um sistema ubíquo é um sistema distribuído cujo número de componentes ultrapassa o número de pessoas interagindo com ele. De um sistema ubíquo se espera transparência e passividade: idealmente seria invisível, consciente do ambiente e reagindo de forma inteligente às necessidades do usuário. Além dos problemas usuais a sistemas distribuídos dinâmicos, incorporar ubiqüidade impõe componentes miniaturizados capazes de se comunicar sem fio e com baixo consumo de energia. Tais sistemas precisam de protocolos de comunicação que apóiem redes transientes com topologias arbitrárias (com agentes móveis e conexões ad hoc).

## 3. Requisitos de Sistemas Computacionais para Aplicações Onivalentes

Para que se possa fazer uma discussão sobre as dependências e conflitos entre requisitos, é preciso entender profundamente cada requisito. Por isso, definiremos a seguir o que entendemos pelos termos *correção*, *dependabilidade*, *segurança*, *escalabilidade* e *evolutividade* no contexto de requisitos de sistemas computacionais.

### 3.1. Correção

Grande parte das pesquisas realizadas até hoje na área de correção referem-se a aspectos funcionais, ou seja, em responder a questão: *"Como garantir que o sistema faz o que deveria fazer, ou seja, atende à sua especificação?"* É claro que este é um ponto fundamental, pois se um sistema não se comporta como o esperado nem nas condições ideais (sem considerar falhas, tempo de resposta, escalabilidade, aspectos de segurança, ...), ele é inútil. Existem várias abordagens para provar correção (funcional) de sistemas, por exemplo lógica de Hoare [Hoare 1969], verificação de modelos [Clarke et al. 2000, Burch et al. 1992], refinamentos sucessivos [Jones 1990]. Todas essas abordagens baseiam-se na construção de análise de um modelo matemático que representa o comportamento do sistema (semântica formal). As grandes limitações no uso desses métodos formais de maneira mais ampla são o tamanho dos modelos gerados e a impossibilidade de automatizar totalmente a análise.

Apesar dessas limitações, métodos para garantir a correção funcional de sistemas computacionais vêm sendo utilizados cada vez mais na prática, pois a medida em que os sistemas ficam mais complexos, argumentações informais sobre a correção do software tornam-se impossíveis. É importante salientar, também, que, nos últimos anos, os métodos usados para garantir a correção têm evoluído muito (incluindo técnicas de redução de espaço de estados, interpretação abstrata, automatização de novas técnicas de prova, verificação composicional, entre outras) no sentido de minimizar as limitações listadas acima.

É possível integrar requisitos não-funcionais em modelos computacionais formais. Os requisitos que podem ser integrados são aqueles que podem ser objetivamente medidos ou verificados (por exemplo, tempo de resposta). Requisitos como usabilidade são difíceis de incluir em modelos formais, pois são conceitos bastante subjetivos. Apesar de se poder considerar o termo *"correção"* com um sentido mais amplo, no restante deste artigo, quando se falar em correção estar-se á referindo à correção funcional.

### 3.2. Dependabilidade

Dependabilidade (do inglês *dependability*), está relacionada à confiança justificada que se pode colocar no serviço oferecido por um dado sistema. Dependabilidade envolve atributos como confiabilidade, disponibilidade, segurança funcional crítica (safety), integridade, facilidade de manutenção e garantias de desempenho adequado mesmo na ocorrência de falhas aleatórias e imprevisíveis [Avizienis et al. 2004]. Disponibilidade e confiabilidade são os atributos principais. Disponibilidade refere-se à probabilidade do sistema estar operacional no instante em que for solicitado, enquanto que a confiabilidade refere-se à probabilidade de que um sistema funcione corretamente durante um dado intervalo de tempo correspondente ao tempo de missão.

Aplicações distribuídas executando sobre uma rede de comunicação estão sujeitas a inúmeros problemas devido à baixa disponibilidade, baixa confiabilidade, domínios administrativos múltiplos, políticas de controle de recursos conflitantes, grande ocorrência de falhas com efeitos catastróficos sobre as aplicações distribuídas e equipamentos. Uma falha pode provocar perda de estado de uma aplicação e desconexão do resto do sistema impedindo avanço na computação e levando o sistema a um estado inconsistente. Falhas podem ter diversas causas, como fadiga de componentes de hardware, interferências

ambientais, erros de projeto ou programação [Avizienis et al. 2004].

A área de sistemas distribuídos é permeada de desafios. Se o ambiente distribuído fosse totalmente livre da ocorrência de falhas, os algoritmos distribuídos seriam simples, diretos e eficientes. As dificuldades de executar programas em um ambiente sujeito a falhas mesmo em sistemas de pequena escala são conhecidas: impossibilidade de consenso em ambientes assíncronos, ausência de relógio global, algoritmos ineficientes em termos de rodadas de trocas de mensagens [Gärtner 1999]. Os modelos usuais de sistemas distribuídos, o modelo síncrono e o modelo assíncrono, parecem cada vez mais distantes de prover o arcabouço teórico necessário para solucionar de forma eficiente problemas vitais como consenso distribuído, eleição de líder, gerência de grupos de processo, visão de estado global, consistência de dados replicados e vários outros relacionados.

Para tratar desses problemas, técnicas para tornar os sistemas mais robustos como multicast confiável [Défago et al. 2004], recuperação a um estado anterior [Elnozahy et al. 2002] e replicação [Saito and Shapiro 2005] podem ser aplicadas. Mas as redes de comunicação mantém-se lentas e não confiáveis. Sua latência de comunicação e sua disponibilidade não parecem estar melhorando. Adicionalmente, dispositivos móveis com conectividade intermitente estão se tornando populares num ritmo acelerado [Saito and Shapiro 2005]. Algoritmos distribuídos tolerantes a falhas criados para redes convencionais de pequena escala, baseados em propriedades fortes e premissas pessimistas, não escalam adequadamente [Eugster et al. 2004]. É difícil, senão impossível, construir ambientes para replicação com consistência forte ou sistemas de comunicação de grupo com visão global e ordenação total de mensagens em sistemas dinâmicos escaláveis porque o desempenho e a disponibilidade são demasiadamente penalizados quando o número de nós computacionais explode.

### 3.3. Segurança

Sistema dito seguro é aquele capaz de resistir a ações maliciosas com o objetivo de comprometer a satisfação de um ou mais requisitos de segurança essenciais para o seu funcionamento. Ações maliciosas bem sucedidas podem (a) representar risco a vidas humanas, (b) provocar prejuízos financeiros, (c) afetar a privacidade de indivíduos [Anderson 2001]. Os requisitos a serem satisfeitos são dependentes do tipo de aplicação e do grau de segurança exigido [Stallings 2002]. Alguns dos principais são: *disponibilidade* (garante que o sistema está pronto para uso quando necessário); *autenticidade* (determina se a entidade é quem afirma ser); *confidencialidade* (protege dados contra observação por entidades não autorizadas); *integridade* (protege dados contra modificação, seja ela maliciosa ou acidental); *autorização* (restringe, com base em direitos, o acesso a recursos e dados no sistema); *não-repúdio* (evita que uma entidade negue responsabilidade por ações executadas no sistema).

Satisfazer requisitos como os supracitados implica incorporar ao sistema uma combinação de mecanismos como autenticação, controle de acesso, auditoria e criptografia. Uma vez "protegido", é muito difícil, se não impossível, demonstrar formalmente que o sistema não apresenta vulnerabilidades.

Pesquisas têm procurado propor formas sistemáticas para o tratamento de requisitos de segurança em todo o ciclo de vida de sistemas [Devanbu and Stubblebine 2000b]. Uma abordagem atrativa consiste em estender padrões como UML para permitir a

modelagem de aspectos como privacidade e integridade já desde o início do processo [Lodderstedt et al. 2002].

O desenvolvimento de sistemas, sobretudo os de larga escala, tem sido marcado pela integração de componentes de diversas naturezas e procedências. Ao mesmo tempo em que este modelo é atrativo, ele expõe os sistemas resultantes a uma nova dimensão de riscos. As vulnerabilidades são decorrentes da não divulgação – para não comprometer a propriedade intelectual do fornecedor – de detalhes suficientes que permitam caracterizar como os componentes operam para satisfazer determinados requisitos de segurança. Entre as alternativas para contornar essa limitação, destaca-se a investigação de técnicas *gray-box* [Devanbu and Stubblebine 2000a], que permitem aos fornecedores de componentes informar detalhes suficientes sobre o processo de verificação de seus componentes e, ao mesmo tempo, proteger sua propriedade intelectual.

Sobre a aplicação de métodos formais na área de segurança, ressalta-se que os mesmos não permitem afirmar que um sistema é 100% seguro [Wing 1998]. A especificação formal de um sistema requer que sejam definidas as premissas sobre o ambiente onde o sistema será executado. Uma prova de corretude só é válida se todas as premissas forem satisfeitas. O atacante, neste contexto, procura violar essas premissas. Caso seja bem sucedido em comprometer pelo menos uma, a prova deixa de ser válida. Diante de tal problemática, a comunidade tem optado por investigar métodos específicos, cuja aplicação se restringe à verificação de requisitos pontuais de segurança (resultando em grande variedade de formalismos). Ainda, as verificações são realizadas sobre modelos que, em geral, exigem um conjunto de simplificações para serem computáveis, além de não capturarem vulnerabilidades a que a implementação dos modelos está sujeita. Para minimizar este problema, duas abordagens vêm sendo investigadas: emprego de verificadores sobre abstrações derivadas automaticamente do código fonte do sistema sob análise e "injeção" coordenada de ataques ao sistema com o objetivo de expor problemas de segurança.

### 3.4. Escalabilidade

Formalmente a noção de ser escalável não é fácil de definir e freqüentemente precisa considerar a aplicação concreta que precisa que ser escalável [Hill 1990, Rana and Stout 2000]. No contexto desse artigo consideramos um sistema escalável se ele é capaz de satisfazer maior demanda ou prestar um volume maior de serviços com um aumento adequado de recursos. Para um sistema distribuído, uma das características mais importantes é ser escalável no número de componentes envolvidos: o sistema deve manter, por exemplo, o mesmo nível de disponibilidade (ou ao menos garantir que a disponibilidade degrada pouco) à medida em que o número de componentes aumenta.

Um outro aspecto da escalabilidade são técnicas para mitigar ou esconder efeitos de características que não escalam devido a restrições fundamentais. A latência de comunicação, por exemplo, é limitada pela velocidade de transmissão de sinais. Portanto, técnicas como *"overlapping communication and computation"* para esconder latências em tarefas computacionais [Sancho et al. 2006], redudância e *"caching"* tornam-se importantes no desenvolvimento de algoritmos.

A maioria dos sistemas atuais são pouco escaláveis. Entre os mais escaláveis são a World Wide Web (approx. $4 \times 10^9$ nós) e o computador paralelo Blue Gene L (approx. $13 \times$

$10^4$ processadores) [ISC 2007, TOP 500 2006]. Para implementar os sistemas pervasivos do futuro, sistemas locais já vão atingir esse número de componentes: por exemplo um edifício com $100$ apartamentos e com componentes distribuídos em todo edifício (em eletrodomésticos, vestidos, canetas, etc.) facilmente pode conter $\approx 10^5$ componentes.

### 3.5. Evolutividade

O requisito de evolutividade de software [Duchien et al. 2006, Cazzola et al. 2006] têm por objetivo diminuir esforços e custos na manutenção e na reestruturação de software visando corrigir erros ou visando atender mudanças de requisitos funcionais e não funcionais. Estas técnicas envolvem, nos casos mais simples, modificação de código, recompilação e nova distribuição. As técnicas existentes hoje para tratar a evolução de software têm várias limitações: são dependentes das linguagens de programação, não escalam, são difíceis de integrar e não possuem embasamento teórico.

Num contexto em que sistemas computacionais se tornarão cada vez mais onipresentes, exercendo funções essenciais, essa manutenção e reestruturação terá um custo cada vez maior, uma vez que especificações e ambientes estarão em constante modificação. Será, portanto, necessário o desenvolvimento de técnicas que permitam a evolução do software ao longo do tempo, com mínima interferência de programadores.

## 4. Interdependências entre Requisitos

Correção, disponibilidade, segurança, escalabilidade e evolutividade são requisitos de quase todos os sistemas computacionais do presente e de um futuro próximo. Como visto na Seção 3, sistemas computacionais tradicionais já apresentam desafios importantes para o atendimento a estes requisitos de forma isolada. Quando sistemas ubíquos são considerados, o atendimento simultâneo a todos estes requisitos será essencial e pode se tornar impraticável. De fato, estes sistemas terão dimensões bem maiores (em número e complexidade) às atuais, com características de interdependência e cooperação entre um grande número de componentes em diferentes níveis de abstração. Além disso, os componentes destes sistemas serão projetados isoladamente (provavelmente em momentos distintos) e, em vários casos, sem uma ligação explícita (física ou lógica) ou mesmo previsível. Como será visto adiante, nestas condições o atendimento de um requisito implica ou exige o relaxamento de outro, e mecanismos para auxílio à tomada de decisão precisam ser cuidadosamente definidos. Por outro lado, esta divisão do sistema como um todo em partes operacionais autônomas pode permitir a definição de compromissos e espaços finitos (ou limitados) para exploração das alternativas de projeto que levem ao atendimento de determinadas condições ou grupos de condições de acordo com as demandas de cada aplicação. Neste caso, alguns requisitos impõem um alto nível de dependência em relação a outros para que o sistema como um todo responda a um conjunto de requisitos.

Alguns conflitos e dependências entre os requisitos dos sistemas onivalentes são discutidos abaixo.

### 4.1. Correção e demais requisitos

Conforme explicado na Seção 3, para se garantir o requisito de correção de um sistema é preciso se definir um modelo matemático que representa seu comportamento. A partir deste modelo, pode-se elaborar um conjunto de asserções que provem sua correção ou apontem eventuais problemas.

À medida em que a complexidade do sistema computacional aumenta, torna-se mais difícil modelar suas funcionalidaes ou comportamento esperado de uma forma clara e não ambígua. Os aspectos de dependabilidade e segurança são transversais aos requisitos funcionais, mas eles podem afetar significativamente o comportamento de um sistema. Para se garantir a correção em sistemas com requisitos de dependabilidade e segurança, não basta apenas analisar cada requisito separadamente. Normalmente, o tratamento dos aspectos de confiabilidade, disponibilidade e segurança ocorre espalhado por todos os módulos que compõem um sistema (incluindo hardware e software), modificando o comportamento do sistema caso uma situação de exceção (falha ou violação de segurança) seja detectada. Existem abordagens de desenvolvimento de software (por exemplo, programação orientada a aspectos) que tentam construir modelos onde estas duas características convivam de forma integrada, mas ainda não há uma base formal sólida para essas novas abordagens.

A previsão de que um número cada vez maior de componentes funcionais participarão de um mesmo sistema (escalabilidade) só confirma que o número de modelos e asserções possíveis crescerá a um passo provavelmente bem maior que os avanços feitos nos métodos tradicionais de verificação. Apesar das limitações de se analisar a correção de sistemas muito grandes, não necessariamente essas provas não escalam. Por exemplo, se é mostrado que para uma determinada solução do problema dos filósofos jantando não há postergação indefinida (ou seja, nenhum filósofo morre de fome), esta prova valeria para sistemas com 5, 10, 1000 filósofos, desde que os "novos" filósofos se comportem exatamente como os que já existem, e a configuração inicial da mesa seja análoga (um palito entre cada dois filósofos). Porém, sistemas distribuídos são naturalmente passíveis de modificação (evolutividade). A funcionalidade do sistema pode ser definida dinâmicamente, de acordo com os nodos que participam de uma comunicação em um dado momento, por exemplo. Além disso, a quantidade de componentes que interagem em uma computação pode ser não apenas variável, mas bastante numerosa. A convivência de componentes construídos em diferentes momentos e sob premissas e condições distintas traz um grau de variabilidade de funcionalidades e cooperações possíveis que dificilmente podem ser previstas. Continuando com a analogia com o problema dos filósofos, considere por exemplo a inclusão do tempo de resposta como um novo requisito. Neste caso, obviamente provas para sistemas pequenos dificilmente escalam para instâncias maiores (por exemplo, se o requisito de tempo for que o filósofo não pode passar mais que 12 horas sem comer, um sistema que tem 5 filósofos pode satisfazer esse requisito, mas se colocarmos 1000 filósofos, o sistema pode não conseguir atender essa restrição). Se mais filósofos forem incorporados ao longo do jantar, o sistema pode mudar totalmente sua configuração original.

Especificar propriedades funcionais e verificá-las é praticamente inviável para sistemas com capacidade de evolução. A complexidade de sistemas evolutivos se justifica na medida em que estão inseridos em situações de constantes modificações tanto de ambiente como de requisitos. Outras noções de correção serão necessárias: como em uma sociedade, agentes deverão ter a capacidade de se adequar a normas pré-definidas, deverão respeitar noções de hierarquia e, ao longo do tempo adquirir ou perder confiança dos demais agentes de acordo com o seu comportamento social. Dado que estabelecer correção funcional será extremamente difícil, senão impossível, as garantias de qualidade dos sistemas computacionais virão, em boa parte, da verificação de aspectos do seu com-

portamento social.

Pode-se observar, então, que o atendimento aos requisitos de escalabilidade e evolutividade estão em conflito com o requisito de correção. Quanto mais escalável e evolutivo o sistema, mais difícil se torna a verificação de sua correção. Requisitos de dependabilidade e segurança, por outro lado, devem ser considerados de forma integrada ao requisito de correção. Caso contrário, a correção do sistema sob determinadas condições não poderá ser assegurada.

### 4.2. Dependabilidade e demais requisitos

A experiência com o atendimento dos requisitos de dependabilidade em sistemas escaláveis é relativamente recente e está longe de ser estável. Muitos dos atributos de dependabilidade necessários a sistemas distribuídos correspondem a propriedades fortes e, portanto, não escalam adequadamente, ou seja, com custo administrável. Sem esquecer que sistemas que escalam de centenas a milhões de nós dinamicamente poderiam ser enquadrados no modelo de computação distribuída assíncrona. Neste modelo não é possível distinguir um computador atrasado de outro em colapso ou particionado. Não é possível usar time-out para detecção de falhas. Não é possível chegar a consenso em sistemas assíncronos sujeitos a falhas. Em sistemas distribuídos, consenso é uma necessidade recorrente: para eleição de líder, cada compor grupos de participantes, para diagnosticar sub-sistemas com defeito, para ordenação total de mensagens, para consistência de dados replicados e várias outras atividades essenciais. Se tentássemos tornar sistemas escaláveis, impondo sincronização de relógios através de onerosos protocolos baseados em troca de mensagens, ainda assim, consenso é uma propriedade forte e o custo relacionado seria demasiado para sistemas dinâmicos escaláveis.

Apesar da possibilidade de evolução do sistema onivalente tanto em quantidade quanto em qualidade (funcionalidades disponíveis) de nodos computacionais, a garantia de desempenho e confiabilidade destes sistemas dinâmicos será um grande desafio. Dispositivos móveis com conectividade intermitente e com diferentes níveis de garantias de dependabilidade podem comprometer a confiabilidade de todo o sistema.

Por fim, associando-se dependabilidade com segurança computacional pode-se obter um ambiente robusto para execução de aplicações distribuídas. É sabido que diversas brechas na segurança de sistemas ocorrem devido a modelos de protocolos de comunicação, políticas de controle de recursos, etc, que afetam da mesma forma a dependabilidade do sistema.

Portanto, pode-se observar um conflito importante entre os requisitos de dependabilidade e escalabilidade e evolutividade dos sistemas onivalentes. Por outro lado, o requisito de segurança afeta e é afetado pela dependabilidade. Falhas de segurança podem comprometer a dependabilidade do sistema, mas decisões que garantam a dependabilidade também podem afetar os níveis de segurança que podem ser assegurados.

### 4.3. Segurança e demais requisitos

Segurança e escalabilidade são aspectos que, dificilmente, convivem harmoniosamente no projeto e desenvolvimento de sistemas de qualidade [Barcellos and Gaspary 2006]. Por exemplo, para gerenciar identidades e evitar que um nodo malicioso assuma múltiplas

identificações em um sistema de larga escala, como os sistemas peer-to-peer (P2P), é preciso lançar mão de uma autoridade certificadora de confiança [Douceur 2002]. Tal abordagem, contudo, é pouco indicada para sistemas distribuídos com potencial para lidar com milhões de nodos por representar um "gargalo´´. Mesmo que se relaxe o aspecto escalabilidade, o emprego de autoridade certificadora dificulta a satisfação do aspecto de tolerância a falhas, posto que tal autoridade configura ponto central de falhas. Por outro lado, caso se admita como premissa a possibilidade de um usuário assumir múltiplas identificações, então outros mecanismos como replicação (comumente utilizados para conferir confiabilidade e disponibilidade ao sistema) acabam comprometidos. No caso de um sistema de armazenamento de arquivos em rede, por exemplo, as múltiplas réplicas de um objeto podem ficar sob controle de um único nodo malicioso.

Em relação ao requisito de evolutividade, pode-se observar não um conflito, mas uma relação de interdependência. Em um sistema dinâmico, a definição de novas funcionalidades ou mudanças sobre um sistema que já se encontra em funcionamento pode acarretar em inconsistências ou alterações de determinadas políticas que garantiam a segurança do sistema original. A dificuldade, neste caso, está na previsibilidade de como uma nova funcionalidade (ou a modificação de uma funcionalidade ou implementação existente) impactará o sistema em seus diversos níveis. Por exemplo, ao se trocar a implementação de um algoritmo de criptografia de software para hardware, deve-se prever que a garantia de segurança dada pelo primeiro não será a mesma quando implementada por outro componente. Dessa forma, se se deseja um sistema evolutivo e seguro, os mecanismos que garantem a evolutividade não podem deixar de considerar as eventuais mudanças nas garantias de segurança.

### 4.4. Escalabilidade e Evolutividade

Estes dois requisitos apresentam uma relação de interdependência no caso dos sistemas onivalentes definidos no contexto deste artigo. Sistemas onivalentes pressupõem, de acordo com o que foi dito na Seção 2, a possibilidade de inserção de nodos computacionais heterogêneos dinamicamente e o uso do sistema computacional em um conjunto cada vez maior e mais diverso de aplicações. A construção deste tipo de sistema só será viável economicamente se o paradigma do reuso for intensamente utilizado. Reuso de hardware se dá pela construção de plataformas de processamento, dispositivos com um conjunto básico de componentes de hardware que sejam facilmente programáveis (processadores, lógicas reconfiguráveis, etc). Reuso de software se dá pela definição de programas também parametrizáveis ou de fácil reprogramação, ou ainda pela possibilidade de se integrar novos programas a uma unidade funcional sem tirá-la de operação (reprogramação em funcionamento). Dessa forma, o requisito de escalabilidade só será atendido se o requisito de evolutividade também o for.

## 5. Em Busca de um Modelo para Construção de Aplicações Onivalentes

Na seções anteriores descrevemos as aplicações onivalentes, os requisitos de qualidade que consideramos essenciais para este tipo de sistemas, bem como as interdependências entre esses requisitos. Resumindo, podemos chegar às seguintes conclusões:

- Os sistemas distribuídos, dinâmicos e ubíquos são cada vez mais freqüentes e tendem a controlar várias atividades potencialmente críticas. Portanto, precisamos

poder depositar um grau de confiança neles, que depende do quão crítico o sistema é.

- Os problemas relacionados a falhas nos componentes físicos (tanto no sentido de componentes deixarem de funcionar como gerarem resultados incorretos) serão cada vez mais freqüentes, e a ponto de não poderem ser desconsiderados no projeto de sistemas de software que executam nesses componentes.
- Usualmente os sistemas distribuídos, dinâmicos e ubíquos são compostos por um número imenso de componentes (de software e hardware). Portanto, o desenvolvimento de sistemas deve ser guiado pelas práticas de *"dividir para conquistar"* (desenvolvimento baseado em componentes) e níveis de abstração.
- Determinados requisitos de qualidade devem ser tratados de forma conjunta, pois existem fortes dependências entre eles. É o caso da correção, dependabilidade e segurança, da segurança e evolutividade e da escalabilidade e evolutividade, por exemplo.
- Alguns requisitos são conflitantes e também precisam ser considerados de forma conjunta (dependabilidade e evolutividade, correção e evolutividade).
- O requisito da escalabilidade pode dificultar enormemente o atendimento simultâneo de outros requisitos (correção, dependabilidade e segurança).

A partir desta análise, pode-se inferir que não é possível atender a todos os requisitos de qualidade simultâneamente para sistemas distribuídos, dinâmicos e ubíquos. Não há como garantir, por exemplo, que o hardware não vai falhar (deixar de funcionar ou gerar resultados imprecisos) ou que o software esteja livre de erros ou imune a ataques a qualquer momento. Devido às dimensões e complexidade do sistema onivalente, qualquer método que vise proteção total terá um custo impraticável. Dessa forma, novos métodos de projeto, verificação e teste devem ser definidos para sistemas distribuídos, dinâmicos e ubíquos. Devem estar previstas etapas de exploração do espaço de projeto, com o auxílio de ferramentas de apoio à tomada de decisões, para que um conjunto de compromissos entre custos e níveis de qualidade sejam definidos e implementados de acordo com as demandas de cada aplicação alvo. Portanto, a questão que se coloca agora é:

*Como podemos desenvolver sistemas distribuídos, dinâmicos e ubíquos de maneira a podermos associar um "grau ou função de confiabilidade"ao produto final?*

Para responder essa questão, uma série de modificações no processo de desenvolvimento de sistemas é necessária. Dentre elas, três são consideradas fundamentais: *(i)* trabalhar com modelos probabilísticos, pois o que queremos saber na realidade é qual a probabilidade do sistema funcionar sem defeitos; *(ii)* modificar o desenvolvimento baseado em componentes, em contratos, etc, para incluir, nas interfaces/contratos, informações sobre o nível de garantia que o componente oferece com relação às várias características consideradas no sistema (dependabilidade, segurança, escalabilidade, ...); *(iii)* verificar correção levando em consideração as outras características do sistema, ou seja, deve haver uma integração dos requisitos impostos por algumas características (pois vimos, por exemplo, que correção depende da segurança ou da disponibilidade de um sistema).

A seguir, citaremos as pesquisas que estão sendo realizadas no Instituto de Informática da UFRGS que contribuem para a solução do problema de desenvolver aplicações onivalentes confiáveis.

A noção de probabilidade para a execução de determinadas tarefas deve ser incluída já na fase de especificação de software, pois a noção de correção depende da especificação. Sendo assim, métodos de especificação de sistemas distribuídos que incluem noções de probabilidades são necessários, como por exemplo, gramáticas de grafos estocásticas [Mendizabal et al. 2005]. Usando este tipo de modelos, pode-se analisar propriedades do tipo: *"Qual a probabilidade do sistema atingir um determinado estado?"*, que são bastantes relevantes em aplicações onivalentes.

Os comportamentos de componentes de aplicações distribuídas raramente podem ser descritos por funções. Eles são melhor descritos como interações entre o componente e seu ambiente. Portanto, a interface que descreve de maneira abstrata a funcionalidade de um desses componentes deve conter os padrões de interação nos quais o componente espera de engajar [Ribeiro et al. 2006]. Além disso, essas interfaces devem expressar, também de maneira probabilística, e provavelmente através de funções dependendo do custo/desempenho esperados e dos graus de confiabilidade dos componentes utilizados, o grau de satisfação do componente em relação a cada requisito (correção, segurança, dependabilidade, ...). Do ponto de vista da arquitetura do sistema, precisamos prover técnicas de composição que permitam não somente gerar um modelo comportamental do sistema, mas também permitam calcular a qualidade do sistema com base nas qualidades oferecidas por seus componentes. Isso permitiria, por exemplo, desenvolver técnicas de integração e otimização para, dados um conjunto de componentes alternativos, gerar o melhor sistema (ou seja, o que melhor se adequa aos graus de confiabilidade dos requisitos, levando em consideração o custo e desempenho esperados).

No aspecto de evolutividade de sistemas computacionais, técnicas de Inteligência Artificial, mas precisamente de Sistemas Multiagentes, serão de extrema utilidade. Sistemas tipicamente serão compostos de vários agentes, provavelmente desenvolvidos de forma independente. Estes agentes deverão cooperar, negociar e competir entre si por recursos e serviços de forma mais sofisticada e complexa do que verificado atualmente. Um sistema computacional poderá ser visto como uma sociedade com toda a sua complexidade onde fatores tais como papéis, relações de hierarquia e confiança, e crenças serão determinantes. Propostas iniciais de modelos teóricos incluindo algumas características relevantes para evolução de software podem ser encontradas em [Vieira et al. 2007, Bordini and Moreira 2004] .

Diversos itens abordados neste texto são cobertos, até certo ponto, pelo conceito de computação autonômica onde um sistema autonômico é capaz, por exemplo, de se recuperar de falhas, se auto-otimizar, e ser consciente do seu próprio estado interno. Os princípios da computação autonômica estão sendo investigados na UFRGS na perspectiva do gerenciamento de sistemas altamente distribuídos e de alcance mundial, mais particularmente em relação aos sistemas peer-to-peer (P2P). Acredita-se que sistemas de software desenvolvidos sob esta perspectiva terão uma chance maior de estarem alinhados com os desafios anteriormente considerados.

No que se refere à segurança, sobretudo em sistemas de larga escala, tem-se buscado investigar técnicas que permitam satisfazer requisitos cruciais de segurança – como autenticidade e autorização – sem comprometer outros requisitos não funcionais e outros aspectos importantes, tais como escalabilidade e tolerância a falhas. Nesse contexto particular, um dos focos de investigação tem sido segurança em aplicações peer-to-peer [Barcellos and Gaspary 2006], que constituem uma alternativa promissora para o desenvolvimento de aplicações de grande porte, potencialmente compostas por milhares de usuários.

Injeção de falhas é um método experimental de teste que complementa testes funcionais e visa especificamente validar o sistema sob condições de falha. Para sistemas distribuídos de larga escala, em que os participantes interagem unicamente por troca de mensagens, injetam-se falhas nos subsistemas de comunicação e observa-se como o sistema reage a essas falhas [Jacques-Silva et al. 2006] . Desta forma a disponibilidade e confiabilidade do sistema podem ser aferidas. Emular tão próximo quanto possível falhas reais de comunicação em sistemas de larga escala, permitir o controle do experimento de forma eficiente diminuindo a interferência e intrusividade das ferramentas de injeção de falhas e monitoramento e extrair medidas apropriadas para a validação de sistemas onivalentes sujeitos a falhas são os desafios enfrentados na pesquisa nesta área.

Técnicas de teste integrado de hardware e software serão essenciais para avaliar o funcionamento do sistema em presença de falhas nos dois níveis de abstração. Estas técnicas devem garantir um teste de qualidade e de baixo custo. Teste baseado em software para processadores e estruturas de interconexão foram desenvolvidos recentemente no grupo e estão sendo expandidos atualmente. Da mesma forma, o grupo possui experiência com o planejamento do teste a partir do início do projeto e a exploração do espaço de projeto e teste de uma sistema desde as fases iniciais de desenvolvimento [Cota and Liu 2006]. Por fim, técnicas para garantir a confiabilidade de um sistema de hardware durante seu funcionamento são também alvo de pesquisas deste grupo [Frantz et al. 2006].

## 6. Conclusões

Sistemas computacionais onipresentes de qualidade possuem características de difícil escalabilidade, que imporão severas dificuldades ao desenvolvimento destes futuros sistemas. Pesquisas feitas por grupos multidisciplinares que levem em conta interdependência entre os diferentes requisitos serão a chave para a construção das aplicações que dominarão a sociedade futura. Os modelos e abstrações hoje utilizados deverão ser substituídos por outros onde a integração de soluções prevaleça sobre a otimização de um único parâmetro.

Neste artigo foi dado um primeiro passo na direção de um método para desenvolvimento de sistemas computacionais onivalentes: identificar conflitos e dependências entre os requisitos considerados fundamentais para este tipo de sistemas: correção, dependabilidade, segurança, escalabilidade e evolutividade. Além disso, foram sugeridas linhas de pesquisa que podem levar ao desenvolvimento de aplicações onivalentes de qualidade.

## Referências

Anderson, R. J. (2001). *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley.

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. E. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33.

Barcellos, M. P. and Gaspary, L. P. (2006). *Fundamentos, Tecnologias e Tendências rumo a Redes P2P Seguras*, pages 187–244. Atualizações em Informática. PUC-Rio, Rio de Janeiro.

Bordini, R. H. and Moreira, Á. F. (2004). Proving BDI properties of agent-oriented programming languages: The asymmetry thesis principles in AgentSpeak(L). *Annals of Mathematics and Artificial Intelligence*, 42(1–3):197–226. Special Issue on Computational Logic in Multi-Agent Systems.

Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. J. (1992). Symbolic model checking: $10^{20}$ states and beyond. *Information and Computation*, 98(2):142–170.

Cazzola, W., Chiba, S., Coady, Y., and Saake, G., editors (2006). *Proceedings of RAM-SE'06, 3rd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution Nantes, France, 4th of July*.

Clarke, E., Grunberg, O., and Peled, D. (2000). *Model Checking*. MIT Press.

Cota, É. and Liu, C. (2006). Constraint-driven test scheduling for NoC-based systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25:2465–2478.

Davidoff, S., Lee, M. K., Yiu, C., Zimmerman, J., and Dey, A. K. (2006). Principles of smart home control. In *Ubicomp 2006, LNCS 4206*, pages 19–34.

Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421.

Devanbu, P. T. and Stubblebine, S. (2000a). Cryptographic verification of test coverage claims. *IEEE Transactions on Software Engineering*, 26(2):178–192.

Devanbu, P. T. and Stubblebine, S. (2000b). Software engineering for security: a roadmap. In *International Conference on Software Engineering, Proceedings of the Conference on the Future of Software Engineering*, pages 227–239. ACM Press.

Douceur, J. R. (2002). The sybil attack. In *Peer-to-Peer Systems: First InternationalWorkshop, IPTPS 2002 Cambridge, MA, USA, March 7-8, 2002. Revised Papers*, pages 251–260. Springer Berlin / Heidelberg.

Duchien, L., D'Hondt, M., and Mens, T., editors (2006). *Proceedings of the International ERCIM Workshop on Software Evolution, Lille, France, 6-7th of April*.

Elnozahy, E. N., Alvisi, L., Wang, Y.-M., and Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408.

Eugster, P. T., Guerraoui, R., Kermarrec, A.-M., and Massoulié, L. (2004). Epidemic information dissemination in distributed systems. *IEEE Computer*, 37(5):60–67.

Frantz, A. P., Kastensmidt, F. L., Carro, L., and Cota, É. (2006). Dependable network-on-chip router able to simultaneously tolerate soft errors and crosstalk. In *Proceedings of the IEEE International Test Conference*.

Gärtner, F. C. (1999). Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26.

Hill, M. D. (1990). What is scalability? *SIGARCH Comput. Archit. News*, 18(4):18–21.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

ISC (2007). Internet domain survey. Internet system consortium. `http://www.isc.org`.

Jacques-Silva, G., Drebes, R. J., Gerchman, J., da Trindade, J. M. F., Weber, T. S., and Jansch-Pôrto, I. (2006). A network-level distributed fault injector for experimental validation of dependable distributed systems. In *COMPSAC (1)*, pages 421–428.

Jones, C. (1990). *Systematic Software Development using VDM*. Prentice Hall, 2nd edition.

Lodderstedt, T., Basin, D., and Doser, J. (2002). SecureUML: A UML-based modeling language for model-driven security. In *UML 2002 - The Unified Modeling Language: 5th International Conference, Dresden, Germany, September 30 - October 4, 2002. Proceedings*, pages 426–441. Springer Berlin / Heidelberg.

Mendizabal, O., Dotti, F., and Ribeiro, L. (2005). Stochastic object based graph grammars. In *8th Brazilian Symposium on Formal Methods (SBMF)*, pages 128–143.

Rana, O. F. and Stout, K. (2000). What is scalability in multi-agent systems? In *AGENTS '00: Proceedings of the fourth international conference on Autonomous agents*, pages 56–63, New York, NY, USA. ACM Press.

Ribeiro, L., Dotti, F. L., Santos, O., and Pasini, F. (2006). Verifying object-based graph grammars: An assume-guarantee approach. *Software and Systems Modeling*, 5:289–312.

Saito, Y. and Shapiro, M. (2005). Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81.

Sancho, J. C., Barker, K. J., Kerbyson, D. J., and Davis, K. (2006). Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In *SC 2006*.

Stallings, W. (2002). *Network Security Essentials*. Prentice Hall, 2nd edition.

TOP 500 (2006). TOP500 Supercomputer Sites. http://www.top500.org.

Vieira, R., , Moreira, Á. F., Wooldridge, M., and Bordini, R. (2007). On the formal semantics of speech-act based communication in an agent-oriented programming language (aceito para publicação). *Journal of Artificial Intelligence Research*.

Weiser, M. (1993). Some computer science issues in ubiquitous computing. *Commun. ACM*, 36(7):74–84.

Wing, J. M. (1998). A symbiotic relationship between formal methods and security. In *Computer Security, Dependability and Assurance: From Needs to Solutions. Proceedings*, pages 26–38. IEEE Computer Society Press.