Pontifícia Universidade Católica
do Rio de Janeiro

Carla Galdino Wanderley

Uma sistemática de monitoramento de erros em sistemas distribuídos

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do título de Mestre pelo Programa de Pós-Graduação em Informática da PUC-Rio.

Orientador: Prof. Arndt von Staa

Rio de Janeiro

Abril de 2015

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Carla Galdino Wanderley

Uma sistemática de monitoramento de erros em sistemas distribuídos

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-Graduação em Informática do Departamento de Informática do Centro Técnico e Científico da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Arndt von Staa
Orientador
Departamento de Informática - PUC-Rio

Prof. Markus Endler
Departamento de Informática - PUC-Rio

Prof.ª Noemi de La Rocque Rodriguez
Departamento de Informática - PUC-Rio

Prof. José Eugenio Leal
Coordenador Setorial do Centro
Técnico Científico – PUC-Rio

Rio de Janeiro, 01 de abril de 2015

Carla Galdino Wanderley

Graduou-se em Engenharia de Computação na Pontifícia Universidade Católica do Rio de Janeiro (Brasil, Rio de Janeiro).

Ficha Catalográfica

Wanderley, Carla Galdino

Uma sistemática de monitoramento de erros em sistemas distribuídos / Carla Galdino Wanderley ; orientador: Arndt von Staa. – 2015.

56 f. : il. (color.) ; 30 cm

Dissertação (mestrado)–Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2015.

Inclui bibliografia

1. Informática – Teses. 2. Erros em sistemas distribuídos. 3. Monitoramento de erros. 4. Monitoramento de sistemas distribuídos. I. Staa, Arndt von. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

À família.

Agradecimentos

Ao meu orientador Prof Arndt von Staa, pelo apoio e todas as conversas que tivemos durante este período.

Ao CNPq e à PUC-Rio, pelos auxílios concedidos, sem os quais este trabalho não poderia ter sido realizado.

Aos professores Markus Endler e Noemi que participaram da Comissão examinadora.

À AevoTech por ter cedido a experiência que pode ser aplicada à avaliação deste trabalho.

À Marcelo Blois pelo tempo cedido e compreensão.

Ao meu noivo, Carlos Costa, por toda a ajuda, compreensão e companheirismo.

Aos meus pais e familiares por toda a compreensão.

Aos amigos, Eliana Goldner, Pedro de Goes, Pedro Grojsgold e Walther Maciel pelo apoio, estímulo e por estarem sempre por perto nos momentos difíceis.

Resumo

Wanderley, Carla; Staa, Arndt von. **Uma sistemática de monitoramento de erros em sistemas distribuídos.** Rio de Janeiro, 2015. 56p. Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Sistemas formados por componentes distribuídos possibilitam a ocorrência de falhas provenientes da interação entre componentes. A etapa de testes desta classe de sistemas é árdua, já que prever todas as interações entre componentes de um sistema é inviável. Portanto, ainda que um sistema de componentes seja testado, a ocorrência de erros em tempo de execução continua sendo possível e, evidentemente, esses erros devem ser observados disparando alguma ação que impeça de causarem grandes danos. Este trabalho apresenta um mecanismo de identificação de erros de inconsistência semântica de tipos baseado em logs estruturados. Falhas de inconsistência semântica de tipos são falhas decorrentes da interpretação errônea de valores que são representados sintaticamente sob os mesmos tipos básicos. O mecanismo proposto consiste na geração de logs estruturados conforme a definição de interfaces de comunicação e a identificação de anomalias através de uma técnica existente de verificação de contratos. Além disso, o mecanismo propõe um modelo de gestão taxonômica de tipos semânticos utiliza a técnica de Raciocínio Baseado em Casos (RBC). O mecanismo de identificação de falhas foi implementado através de uma extensão do middleware Robot Operation System (ROS). O mecanismo, além de observar erros, gera informação complementar que visa auxiliar a diagnose da causa do erro observado. Finalmente, uma prova de conceito aplicada a um sistema de controle de locomoção de um robô híbrido, adaptado de um sistema real, foi desenvolvida para a validação da identificação de falhas.

Palavras-chave

Erros em sistemas distribuídos; monitoramento de erros; monitoramento de sistemas distribuídos

Abstract

Wanderley, Carla; Staa, Arndt von (Advisor). **A systematic for error monitoring in distributed systems.** Rio de Janeiro, 2015. 56p. MSc. Dissertation - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Systems formed by distributed components enable the occurrence of faults arising from the interaction between components. The stage of testing this class of systems is difficult, since foresee all interactions between components of a system is not feasible. Therefore, even if a component system is tested, the occurrence of run-time errors is still possible and, of course, these errors should be seen shooting some action that prevents them from causing major damage. This paper presents an identification mechanism of errors given by semantic inconsistency typed data, based on structured logs. Semantic inconsistency of typed data could cause failures due to the misinterpretation of values that are represented syntactically under the same basic types. The proposed mechanism consists in generating structured logs according to the definition of communication interfaces, and identifying anomalies by an existing contract verification technique. In addition, the mechanism proposes a management model of taxonomic semantic types using the Case-Based reasoning technique (CBR). The fault identification mechanism was implemented through an extension of the Robot Operation System middleware (ROS). The mechanism, in addition to observing errors, generates additional information which aims to assist the diagnosis of the cause of the observed error. Finally, a proof of concept applied to a locomotion control system for a hybrid robot, adapted from a real system, has been developed for fault identification validation.

Keywords

Distributted-systems errors; error monitoring; distributed-systems monitoring

Sumário

Lista de figuras

1 Introdução

Segundo (Pfister et al., 1998) um componente é definido como "*a unit of composition with contractually specified interfaces and explicit context dependencies only.*" Considerando esta definição de componente, a natureza distribuída dos sistemas baseados em componentes possibilita a ocorrência de falhas provenientes da interação entre componentes, o que inviabiliza uma garantia de ausência de defeitos remanescentes. A dificuldade de testar ocorre porque não podemos prever todas as interações entre componentes e mesmo que possível, o grupo de combinações entre componentes e interações seria extenso a ponto de inviabilizar a etapa de testes minimamente abrangentes. Tais dificuldades se tornam mais relevantes quando um sistemas baseado em componentes é projetado para desempenhar atividades de alto risco, ou seja, suas falhas podem resultar em perda de vidas, grandes quantidades de dinheiro ou mesmo desastres ambientais.

Sistemas de missão crítica são projetados para desempenhar atividades de alto risco e podem apresentar consequências catastróficas para falhas remanescentes. Dado isto, várias são as abordagens adotadas para mitigar os riscos de falhas em sistemas classificados como tal. Uma das abordagens consiste na criação de sistemas tolerantes a falhas, que são assim chamados por conseguirem identificar falhas em ambiente produtivo e executarem ações sobre o comportamento do sistema que mitigam o risco das consequências de uma falha ocorrida.

Ao considerarmos sistemas tolerantes a falhas, a técnica de injeção de falhas é utilizada para a observação do comportamento de cada componente, do sistema e do mecanismo de comunicação utilizado (Ghosh & Mathur, 1999). Ainda assim, é importante considerar a possibilidade de ocorrência de falhas em ambiente produtivo, visto que as consequências de falhas podem ser catastróficas em sistemas de missão crítica. Assim sendo, a observação de erros que tenham ocorrido em ambiente produtivo não só é de suma importância para a manutenção destes sistemas, bem como deve ser realizada de maneira eficaz, já que falhas entre componentes são raramente replicáveis.

Uma solução para a observação de falhas em ambiente produtivo é o monitoramento comportamental de componentes de um sistema. Este tipo de monitoramento pode ser realizado de diversas formas, inclusive através da utilização de logs. O trabalho de (Araújo et al., 2014) apresenta um mecanismo de introspecção baseado em logs estruturados que aumenta o entendimento acerca do comportamento de sistemas distribuídos através de informações relacionadas à execução de cada um dos componentes que compõem o sistema.

Durante a etapa de teste, podemos gerar logs com informações em excesso, com alto grau de detalhamento e frequência, para potencializar o entendimento acerca da execução do sistema. Grandes volumes de eventos armazenados, embora úteis para diagnosticar sem necessidade de replicação da falha, dificultam sobremaneira a diagnose de falhas em ambiente produtivo no contexto de sistemas distribuídos. Além do custo de armazenamento e do consumo de recursos para sua geração, um dos principais problemas causados pelo grande volume de informações armazenadas é o aumento do esforço de pesquisas nos bancos de dados que armazenam o histórico de eventos. O aumento do esforço pode influenciar diretamente no tempo médio de reparo (MTTR) (Staa, 2000) de sistemas, muitas vezes, dificultando ainda mais a diagnose. Outro problema causado pelo aumento de volume de logs é a dificuldade de extrair os elementos importantes do meio de tantos dados. A percepção de causa de falhas pode ser comprometida pelo excesso de informações apresentadas durante a diagnose.

Em suma, a deficiência de informações nos logs impossibilita a diagnose de falhas no sistema-alvo e o excesso não só pode dificultar a diagnose, mas tende a gerar problemas consideráveis relacionados ao volume de armazenamento.

Portanto, a solução ideal para a questão apresentada é a geração de eventos diretamente relacionados à ocorrência de uma falha. Através desta solução, torna-se possível identificar a ocorrência de falhas tão logo elas ocorram, provendo informações suficientes para a criação de sistemas tolerantes a falhas. Além disso, esta solução possibilitaria a utilização de uma mesma estratégia de obtenção de informações para a diagnose, seja em ambiente de testes ou em produção, já que a geração de eventos estaria sob controle. A utilização de uma mesma abordagem em ambos ambientes evita a ocorrência de perturbações no comportamento do sistema causadas pela geração de informações do processo de diagnose.

Sabe-se que não é possível prever a ocorrência de todas as possíveis falhas de um sistema, principalmente ao se tratar de falhas entre componentes de um sistema. Porém, é possível identificar certas classes de anomalias decorrentes da comunicação entre componentes durante a execução do sistema de componentes, como é o caso da classe de falhas de inconsistência semântica de tipos. Falhas de inconsistência semântica de tipos são falhas decorrentes da interpretação errônea de valores que são representados sintaticamente sob os mesmos tipos básicos.

Sistemas distribuídos, bem como sistemas de componentes podem ser classificados quanto a forma de comunicação adotada. Tais classificações são comumente enumeradas como: *Remote Procedure Call* (RPC), comunicação orientada a mensagem, comunicação orientada a *stream* e comunicação de *multicast* (Tanenbaum & Steen, 2006). Falhas de inconsistência semântica de tipos permeiam a comunicação de sistemas baseados em componentes e podem ocorrer em todas as classificações apresentadas, dependendo da infraestrutura utilizada para a comunicação.

Neste trabalho apresentamos uma técnica de geração de informações que produz informações suficientes para a identificação de falhas relacionadas à inconsistência semântica de tipos decorrentes da comunicação entre componentes. A técnica apresentada tem foco em sistemas que utilizam a troca de mensagens como mecanismo de comunicação e utiliza informações relacionadas à descrição de estrutura de mensagens como base para a validação de contratos. Esta técnica provê a validação de contratos semânticos em ambiente de produção, possibilitando a tomada de decisão para mitigação de riscos de causas de falhas. Além disso, a técnica tem como consequência o aumento de processamento da camada de comunicação do sistema, o que torna a sua aplicação inviável para sistemas de processamento em tempo real e mais proveitosa para sistemas de missão crítica baseados em componentes.

2 Estado da arte

Erros estão relacionados a riscos, uma vez que erros podem ocasionar falhas com graves consequências dependendo do ambiente em que o erro ocorre. Existem inúmeras abordagens relacionadas ao gerenciamento de qualidade de sistemas que vão desde a prevenção de faltas de código até a mitigação do risco representado por um erro.

Abordagens que previnem falhas, chamadas de VV&T (Validação, Verificação e Teste) (DELAMARO et al*.,* 2007), são implantadas principalmente em sistemas que têm alto nível de qualidade como um dos requisitos básicos. Dentre as técnicas utilizadas para prevenção de falhas, destacam-se a utilização de métodos formais e testes de software.

2.1 Métodos formais para controle de qualidade de sistemas de software

Métodos formais consistem na utilização de linguagens, técnicas e ferramentas de validação de sistemas com base no rigor matemático. Segundo Ponsard (PONSARD et al*.,* 2004), métodos formais ainda não foram largamente adotados na indústria devido à necessidade de altos investimentos em aprendizado de novas tecnologias e uma contínua resistência à matemática por parte das equipes de desenvolvimento.

Existem técnicas de métodos formais endereçadas para diferentes etapas do processo de criação de sistemas. Durante o processo de especificação de software, as técnicas existentes promovem uma definição precisa do que o software deve realizar. Alguns exemplos são ASM (Borger & Stark, 2003) e VDM (JONES, 1986). Especificações formais são decompostas em outras menores que especificam componentes do sistema e consecutivamente podem ser enviadas aos desenvolvedores de maneira que os componentes desenvolvidos possam assumir papéis de clientes de outros componentes através do conceito *Design by Contract* (Meyer, 1992).

A verificação é a principal forma de aplicação de métodos formais à nível de implementação. A especificação de um programa pode ser mapeada em

teoremas de corretude que quando satisfeitos comprovam que o programa assume o comportamento descrito na especificação. O método de asserção indutiva (Floyd, 1967) consiste na inserção de anotações com assertivas ao longo do código contendo expressões matemáticas que envolvem valores de entrada e variáveis do código. Outras abordagens são baseadas na geração automática de código através de métodos formais, como em (Abrial, 1996) e (Berry, 2007).

Conforme mencionado, métodos formais são uma poderosa forma de criar software com alto nível de qualidade porém sua implantação é demasiadamente custosa. O alto custo faz com que esta abordagem não seja considerada viável, favorecendo o aumento de esforços relacionados a testes de sistemas.

2.2 Testes de sistemas distribuídos

Uma das principais dificuldades provenientes da etapa de teste de sistemas distribuídos está relacionada ao multi-processamento. Problemas relacionados a multi-processamento e *multi-threading* são constantemente abordados na literatura de sistemas distribuídos (Babaoglu & Drumond, 1985; Barbara & Garcia-Molina, 1989; Cristian, 1989). Esta dificuldade se dá principalmente pela inviabilidade de enumeração de todas as possibilidade de comunicação entre os integrantes do sistema distribuído. Mesmo que possível, o grupo de combinações entre os integrantes seria extenso a ponto de inviabilizar a etapa de testes minimamente abrangentes.

Outra dificuldade decorrente do multi-processamento é a existência de características temporais que também podem influenciar na comunicação. As características temporais de comunicação de sistemas distribuídos dificultam a identificação de condições de contorno para geração de casos de testes efetivos. A etapa de criação de casos de testes para identificação de erros deste tipo pode ser demasiadamente custosa considerando que o domínio de variáveis temporais é contínuo, ou seja, variáveis temporais podem assumir infinitos valores.

As dificuldades inerentes à classe de sistemas distribuídos previamente apresentadas inviabilizam a garantia de ausência de defeitos remanescentes ao fim da etapa de testes. Assim sendo, torna-se importante identificar as causas de falhas em sistemas distribuídos, principalmente ao considerarmos que devido aos pontos apresentados, falhas em sistemas desta classe são raramente replicáveis.

Existem várias abordagens relacionadas à depuração de sistemas distribuídos, as quais têm o objetivo de identificar as causas de falhas em sistemas distribuídos. Uma delas é a verificação de modelos (Killian et al., 2007), que utiliza verificadores baseados nas especificações do sistema para encontrar estados inconsistentes. O trabalho de Killian (Killian et al., 2007) especificamente fornece mecanismos para identificar estados suspeitos do sistema e uma ferramenta para reexecutá-lo de forma interativa. Esta ferramenta é baseada em uma máquina virtual com controles para voltar ou avançar a execução, permitindo que o desenvolvedor estude o comportamento do sistema em cada ponto da execução. Esta abordagem pode ser poderosa para o processo de depuração, no entanto, para detectar estados suspeitos, assume-se que a especificação seja corretamente implementada além dela própria estar correta do ponto de vista formal. Outra desvantagem desta abordagem é a dependência de um ambiente virtualizado para reproduzir a falha.

Uma segunda abordagem seria a verificação baseada em reprodução, que disponibiliza ao programador a capacidade de reprodução da execução de um programa, mimetizando sua ordem e ambiente. Como exemplo de trabalho baseado nesta abordagem, temos “Liblog” (Geels et al., 2007). Esta biblioteca cria logs durante a execução de um sistema e promove um “replay” deterministicamente. O grande benefício deste tipo de ferramenta é a capacidade de reprodução de falhas de forma consistente, mas por outro lado, possui um custo bastante elevado de registro e reprodução de uma execução inteira.

Outra abordagem possível é a criação de depuração e verificação automatizada. Esta abordagem promove o uso de predicados e representação destes predicados em grafos de fluxo de dados. Como exemplo desta abordagem, temos D³S (Liu et al., 2008) e MaceODB (Dao et al., 2009). Ambos promovem extensões para a linguagem C++, mas D³S utiliza uma linguagem de scripts além destas extensões. Esta abordagem pode ser eficaz, já que verifica automaticamente estados de um sistema distribuído como um todo, mas diminui bastante o desempenho das aplicações que a utilizam por conta do “overhead” referente às verificações.

Por fim, temos a depuração através de logs, abordagem que promove a geração de informações para a análise de registros pós-execução. Um exemplo desta metodologia é Pip (Reynolds et al., 2006). Pip possibilita aos programadores especificar expectativas acerca da estrutura do sistema, assim

como outras propriedades, gerando logs para o sistema e, além disso, fornece uma interface visual para explorar comportamentos normais e anormais.

2.3 Testes de sistemas baseados em componentes

Como apresentado anteriormente, existem características intrínsecas da classe de sistemas de componentes que dificultam a aplicação de métodos tradicionais de teste de software. Ainda que estas dificuldades sejam conhecidas, requisitos relacionados à qualidade de sistemas aumentam a importância da aplicação de modelos de testes consistentes, visando atestar um grau de qualidade satisfatório segundo as expectativas do cliente.

O trabalho de (Toroi et al., 2004) apresenta elementos-chave para o teste de sistemas de componentes que obedecem a uma infraestrutura genérica. Além disso, resume as questões relacionadas a teste e manutenção de sistemas baseados em componentes sob os aspectos de heterogeneidade, disponibilidade de código fonte e evolução.

A interação entre componentes implementados em diversas linguagens de programação é um dos pilares dos sistemas baseados em componentes. Ao mesmo tempo que esta característica provê um alto grau de flexibilidade, requer a interoperabilidade entre componentes. A interoperabilidade é uma questão importante neste contexto porque diferentes linguagens, ambientes de desenvolvimento e sistemas operacionais podem ter suas próprias formas de processamento de dados. Os componentes, por sua vez, podem ter diferentes interpretações para um mesmo dado, o que dificulta a transferência de dados entre componentes.

A criação de sistemas baseados em componentes é uma abordagem conhecida por promover o aumento de produtividade e reuso por meio da utilização de commercial-of-the-shelf (COTS) sempre que possível. No entanto, os COTS normalmente são disponibilizados sob a forma binária e não se tem acesso ao seu código fonte.

Sistemas baseados em componentes são dinâmicos, uma vez que possibilitam a atualização de componentes sem necessidade de compilar ou até configurar novamente o sistema. Esta característica é interessante sob o ponto de vista de usabilidade do sistema, mas é potencialmente perigosa para a qualidade do sistema como um todo, uma vez que estas atualizações podem gerar inconsistências na interação entre componentes.

Em sistemas de missão crítica, falhas de sistema frequentemente resultam em catástrofes, incluindo mortes, grande perda de capital ou dados importantes. Considerando esta classe de sistemas, não só a prevenção de erros é necessária, mas a identificação de erros remanescentes é extremamente importante para que decisões que diminuam o risco de catástrofes sejam tomadas sempre que possível.

2.4 Detecção de conflitos semânticos em tipos de dados

Conflitos semânticos em tipos de dados são caracterizados pela interpretação errônea de valores que são representados sintaticamente sob os mesmos tipos básicos. Tais conflitos semânticos podem ocasionar falhas em sistemas baseados em software, as quais são nomeadas falhas de inconsistência semântica de tipos.

O trabalho de (André & Staa, 2013) apresenta uma avaliação do uso de análise estática na detecção de conflitos semânticos em tipos de dados. Este trabalho apresenta motivações para o estudo da classe de falhas de conflitos semânticos, como a catástrofe relacionada ao Mars Climate Orbiter em 1999, causada por uma falha desta classe. Outra motivação apresentada é o atual cenário de intercâmbio e processamento de dados com grande volume de informação e heterogeneidade de participantes.

Embora este trabalho apresente uma solução baseada na prevenção de falhas da classe abordada, não é considerada uma alternativa para a mitigação de riscos relacionados aos erros remanescentes. Além disso, a utilização de técnicas de análise estática pode não ser aplicável a sistemas baseados em componentes, uma vez que componentes de um sistema podem ser desenvolvidos e testados por diferentes equipes, considerando o cenário de heterogeneidade de participantes envolvidos no desenvolvimento. Cenário este que dificulta a análise estática aplicada à integração de componentes em um sistema, principalmente se tratando de integração de componentes em ambiente produtivo.

Assim sendo, este trabalho apresenta uma sistemática de monitoramento com vistas à detecção de erros de conflitos semânticos de tipos, porém, aplicados à sistemas baseados em componentes.

3 Conceitos Fundamentais

O termo componente é definido em diversos níveis de abstração. (Pfister et al., 1998) apresenta a definição de componente adotada pelo presente trabalho: "*a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*".

O entendimento de ambas definições está relacionado ao conhecimento sobre outro termo, a interface. Segundo Wu (Wu et al., 2001), "*interfaces are the access points of components, through which a client component can request a service declared in an interface of the service providing component*."

3.1 Falhas em sistemas de componentes

Além de apresentar as questões de testes de sistemas de componentes, (Wu et al., 2001) apresenta tipos de falhas relacionadas a esta classe de sistemas: falhas entre componentes e falhas de interoperabilidade. Falhas entre componentes estão relacionadas as falhas de programação decorrentes da combinação entre componentes e falhas de interoperabilidade às falhas causadas por características da classe de sistemas de componentes. Falhas de interoperabilidade foram classificadas em: falhas a nível de sistema, nível de programação e de especificação.

Falha de interoperabilidade a nível de sistema é uma classe de falhas relacionadas à características dos sistemas operacionais nos quais os componentes são criados assim como o conjuntos de bibliotecas utilizadas.

Falha de interoperabilidade a nível de programação é uma classe de falhas relacionadas a particularidades das diferentes linguagens utilizadas para a criação de componentes ou mesmo a incompatibilidade entre elas.

Falha de interoperabilidade a nível de especificação é uma classe de falhas relacionada à interpretação errônea das especificações. Falhas desta classe podem ocorrer devido à má interpretação de dados transferidos através de interfaces ou mesmo do padrão de interações entre componentes, como sequência de execução de interfaces.

3.2 Escopo de trabalho: identificação de falhas de inconsistência semântica de tipos

Conforme apresentado, conflitos semânticos em tipos de dados são caracterizados pela interpretação errônea de valores que são representados sintaticamente sob os mesmos tipos básicos. Tais conflitos semânticos podem ocasionar falhas em sistemas baseados em software, as quais são nomeadas falhas de inconsistência semântica de tipos.

Este trabalho tem foco na obtenção de informações necessárias para auxiliar a diagnose de falhas decorrentes da interação entre componentes de sistemas de componentes. De forma mais específica, consideramos apenas a classe de falhas relacionadas à inconsistência semântica de tipos representados pelas interfaces de componentes, classificada por (Wu et al., 2001) como uma classe de falhas de interoperabilidade a nível de especificação. A seguir, apresentamos um exemplo desta classe de falhas:

3.3 Exemplo de falha

O trabalho de Wu apresenta classes de falhas relacionadas a sistemas de componentes, dentre elas, as falhas de interoperabilidade. Segundo Wu, as falhas de interoperabilidade surgem das características intrínsecas de sistemas baseados em componentes, como heterogeneidade, indisponibilidade de código fonte e reuso e são classificadas em diferentes níveis de abstração: sistema, programação e especificação.

Assim sendo, este trabalho tem foco na obtenção de informações necessárias para auxiliar a diagnose de falhas decorrentes da interação entre componentes de sistemas de componentes. De forma mais específica, consideramos apenas a classe de falhas relacionadas à inconsistência semântica de tipos representados pelas interfaces de componentes, classificada por (Wu et al., 2001) como uma classe de falhas de interoperabilidade a nível de especificação.

Consideremos como exemplo, um sistema de componentes que controla elevadores de um prédio auto-sustentável. Neste exemplo, os componentes foram desenvolvidos por diferentes grupos seguindo uma mesma especificação, devidamente testados através de testes unitários e testes de integração e posteriormente implantados. A seguir, apresentamos as interfaces dos componentes que compõem o sistema exemplo.

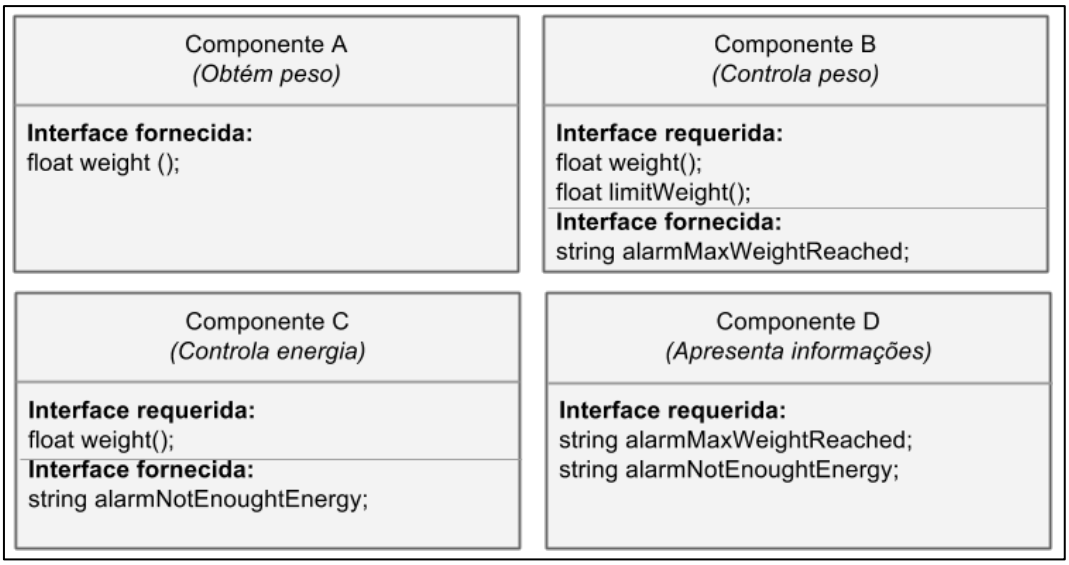

Figura 1 - Interface de componentes de sistema de controlador de elevador

Considere agora que houve um problema com o sensor que obtém o peso incidente no elevador, o qual foi substituído por um outro modelo previamente homologado. O grupo responsável pela homologação do modelo, por questões de compatibilidade, desenvolveu um novo componente de obtenção de peso, seguindo a interface A, porém fornecendo o peso em lbf, conforme o sistema inglês do novo sensor.

Conforme necessário para a manutenção do sistema, o novo componente A será inserido no sistema apresentado anteriormente, sem que ocorram falhas aparentes. No entanto, o novo componente A fornece peso em lbf enquanto os componentes B e C foram projetados para obter peso em Kgf. Assim sendo, os componentes B e C receberão valores de peso 2,20 vezes menor e poderão apresentar falhas. A seguir apresentamos algumas possibilidades de falhas e respectivas consequências:

• Falha 1: Não geração de alarme alarmMaxWeightReached quando há excesso de peso no elevador.

• Consequências:

- Problemas com freio
- Sobrecarga do cabo sustentador

• Falha 2: Não geração do alarme alarmNotEnoughtEnergy quando não há energia suficiente para transportar o peso incidente.

• Consequência:

- Parada inesperada do elevador entre andares

4 Identificação de falhas de inconsistência semântica de tipos – Visão Global

Conforme apresentado anteriormente, falhas provenientes da interação entre componentes são difíceis de testar por conta da natureza distribuída da classe de sistemas considerada. Falhas decorrentes da inconsistência de tipos semânticos, por sua vez, podem permear sistemas devidamente testados, muitas vezes, sem apresentar indícios que possibilitem a identificação de ocorrência de falhas. Com vistas neste problema, desenvolvemos um método de identificação de falhas relacionadas a inconsistência semântica de tipos com base na análise de logs estruturados. As etapas relacionadas a identificação de falhas decorrentes de inconsistência de semântica de tipos estão representadas, de forma abstrata, na figura a seguir:

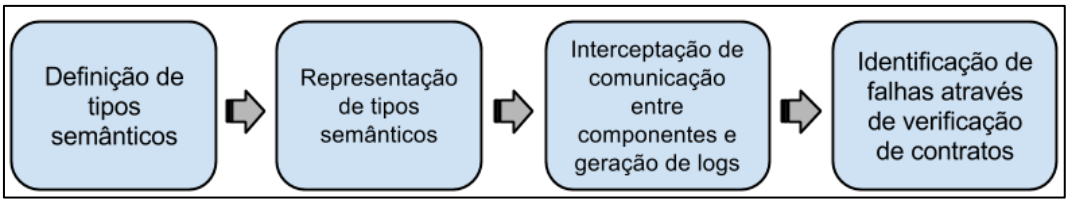

Figura 2 - Etapas para identificação de falhas de inconsistência semântica de tipos

De forma genérica, a definição dos tipos semânticos utilizados por cada um dos componentes pode ser realizada sob demanda, ou seja, no momento da criação das interfaces de um componente. Neste momento, define-se os tipos semânticos relacionados a cada parâmetro e retorno de método. É importante salientar a importância de um padrão entre os tipos dos diversos componentes de um sistema, pois inconsistências de nomenclatura de tipos semânticos podem gerar falsos positivos na identificação de falhas.

4.1 Definição de tipos semânticos

Uma das etapas relacionadas a criação de um novo componente é a definição dos tipos semânticos de cada uma de suas interfaces. Esta tarefa pode ser realizada através da busca por tipos semânticos já conhecidos em um

sistema ou mesmo com a utilização de uma nomenclatura própria para os novos tipos.

Conforme mencionado anteriormente, definições inconsistentes de simbologia podem apontar falhas que não existem. Um exemplo seria a definição de um parâmetro da interface fornecida com o tipo semântico m_s e a utilização desta interface por um outro componente que espera um parâmetro do tipo meter_per_second. Neste caso, ambos fazem referência ao mesmo tipo semântico, porém, os símbolos léxicos utilizados são diferentes.

Esta questão poderia ser resolvida através da análise léxica das interfaces segundo um conjunto pré-estabelecido de símbolos léxicos. Esta abordagem, porém, dificulta a manutenção de sistemas baseados em componentes, pois nem sempre é possível dimensionar a gama de tipos semânticos necessários para a representação de interfaces de sistemas de componentes em geral, devido a sua dinamicidade. Dessa forma, desenvolvemos um mecanismo de manutenção do dicionário de tipos semânticos através da atualização de símbolos léxicos conforme a identificação de falsos positivos. O mecanismo de manutenção de dicionário tem o objetivo de facilitar a identificação de tipos semânticos semelhantes, auxiliando a manutenção da consistência deste dicionário durante a evolução do sistema de componentes. O mecanismo será apresentado com maiores detalhes posteriormente.

4.2 Representação de tipos semânticos conforme interfaces de componentes

Uma vez que os tipos semânticos tenham sido definidos, é necessário representá-los. Como mencionado anteriormente, interfaces de componentes são comumente representadas por uma IDL (Interface Description Language), a qual possibilita a descrição de interfaces independentemente das linguagens de implementação de componentes. Existem inúmeros padrões de IDLs na literatura, alguns são utilizados por mais de um middleware, porém, não há um padrão de IDLs que possibilite a interoperabilidade de componentes entre diferentes middlewares. Por esta razão, a representação de tipos semânticos deve depender da forma de definição de interfaces utilizada para o middleware do sistema de componentes a ser utilizado.

Middlewares cuja comunicação é baseada em trocas de mensagens apresentam arquivos de definição de estrutura de mensagens. Nestes casos, os

tipos semânticos dos parâmetros devem ser representados nos arquivos de descrição estrutural de cada uma das mensagens utilizadas.

4.3 Interpretação e registro de comunicação entre componentes

Após a representação de tipos semânticos, a abordagem do trabalho apresentado prevê a geração de logs estruturados, conforme o trabalho de Araújo (Araújo et al., 2014) sempre que houver comunicação entre componentes. A geração de logs pode ser realizada através de uma extensão da camada de comunicação fornecida pelo middleware utilizado. Os eventos de log gerados devem conter os tipos semânticos para cada parâmetro ou campo de cada mensagem, como no exemplo a seguir:

[**transmitter**:001, receptor:002, **msg_name**:convertValue, **ret_type_t**:meter**, ret_type_r**:centimeter**, p1_type_t:**meter**, p1_type_r**:meter]

Neste evento, transmitter e receptor apresentam o índice dos componentes que assumem o papel de transmissor e receptor, respectivamente, nesta instância de comunicação. A tag msg_name assume um valor identificador da mensagem. A tag ret_type_t informa o tipo semântico esperado pelo transmissor e ret_type_r, o tipo semântico retornado pelo receptor da mensagem. Já as tags p1_type_t e p1_type_r representam os tipos de parâmetros no componente transmissor e receptor, respectivamente. Tags de nome p<i>_type_<t|r> serão criadas conforme o número de parâmetros da mensagem.

4.4 Identificação de falhas de inconsistência semântica de tipos segundo a verificação de contratos

Uma vez que os logs estruturados tenham sido gerados, a identificação de suspeitas de falhas é realizada através de uma ferramenta de verificação de contratos (Rocha & Staa, 2014). Através desta ferramenta, aplicamos um contrato aos logs estruturados, de maneira que rotinas de notificação sejam executadas sempre que um evento que não obedece o contrato é encontrado.

Um exemplo de contrato a ser aplicado no evento de log apresentado na seção anterior segundo o trabalho de (Rocha & Staa, 2014):

```
SINGLE_EVENT_EXPRESSION ->
   {{$exists: 'p1_type_t'},
    {$exists: 'p1_type_r'},
    {$if: {$compare: 'p1_type_t', ==, 'p1_type_r'}}}
```

Figura 3 - Exemplo de contrato

Este contrato provê a comparação entre os parâmentros *p1_type_t* e *p1_type_r.* Eventos de log com valores diferentes para os parâmentros apresentados constituem uma quebra de contrato, a qual resulta em na execução de uma função de tratamento segundo a ferramenta de (Rocha & Staa, 2014).

4.5 Padronização de tipos semânticos

Consideremos uma notificação de falha em um sistema de componentes representado pelo rompimento do contrato definido pela ferramenta de (Rocha & Staa, 2014). Após a identificação de rompimento de contrato, uma rotina de notificação será executada. Esta rotina de notificação deve disponibilizar o evento de log que gerou a suspeita e possibilitará que um humano ou mesmo um motor de inferências identifique se a suspeita de falha realmente procede.

No caso em que a suspeita de falha é apenas um falso-positivo, os tipos semânticos relacionados ao evento de log apresentado são armazenados como equivalentes em um dicionário de tipos semânticos. Este dicionário é utilizado como fonte de conhecimento para a aplicação de filtros nas posteriores suspeitas de falhas. Esta abordagem diminui a ocorrência de falsos-positivos segundo a evolução do sistema, uma vez que os tipos semânticos de parâmetros são comparados segundo um mesmo padrão.

4.6 Inferência de falhas de inconsistência semântica de tipos

Após a padronização de nomenclatura dos tipos semânticos, utilizamos um motor de inferências baseado em regras para identificar falhas de conversão entre diferentes tipos semânticos. As regras de conversão são representadas através de ontologias e são aplicadas aos eventos de log que representam suspeitas de falha.

Quando há uma suspeita de falha de conversão entre tipos semânticos, ou seja, os tipos semânticos representados no evento de log estão relacionados

por ao menos uma das regras do motor de inferências, realizamos a análise de tipos semânticos do componente transmissor. Esta análise é realizada através da criação de uma árvore de sintaxe abstrata com informações semânticas (ASTSI) do trecho de código em que a comunicação suspeita é iniciada, ou seja, no método em que o componente acessa a interface de comunicação do middleware utilizado. Uma vez que a ASTSI tenha sido criada, a árvore é percorrida para que a regra de semântica de tipos também seja verificada.

4.7 Aplicação em sistemas de componentes

Sistemas distribuídos, bem como sistemas de componentes podem ser classificados quanto a forma de comunicação adotada. Tais classificações são comumente enumeradas como: *Remote Procedure Call* (RPC), comunicação orientada a mensagem, comunicação orientada a *stream* e comunicação de *multicast* (Tanenbaum & Steen, 2006). Falhas de inconsistência semântica de tipos permeiam a comunicação de sistemas baseados em componentes e podem ocorrer em todas as classificações apresentadas, dependendo da infraestrutura utilizada para a comunicação.

Remote Procedure Call é um protocolo disponível para requisição de serviço a ser executado em outro componente ou até mesmo em outro computador, sem que haja a necessidade de entendimento de detalhes de redes de comunicação. Esta forma de comunicação é baseada no modelo cliente-servidor e possui duas etapas características chamadas *marshalling* e *unmarshalling,* responsáveis, pelo empacotamento e desempacotamento de paramentos relacionados à chamada de procedimentos, respectivamente.

A aplicação da sistemática apresentada em sistemas desta categoria diferencia-se pela necessidade de extensão das etapas de *marshaling* e *unmarshaling* para a interceptação e registro de comunicação. Neste caso, deve-se criar uma entidade contida no *middleware* de comunicação que identifique os tipos semânticos definidos nas interfaces de procedimento. Além da identificação de tipos semânticos, esta entidade também detém a responsabilidade de gerar um evento de log com os tipos semânticos dos parâmetros esperados pela etapa *unmarshaling.*

Multicast communication possibilita a comunicação entre um componente de origem e um grupo de componentes. A aplicação da sistemática proposta nesta classe de sistemas se assemelha à aplicação em sistemas baseados em troca de mensagens, prevendo a interceptação de comunicação e geração de

um evento de log no momento do recebimento de cada um dos componentes destinatários. A seção a seguir apresentará a aplicação da sistemática apresentada em um *middleware* que promove a comunicação de componentes através da troca de mensagens.

A comunicação via *stream* é utilizada sempre que há necessidade de alto desempenho na transmissão de dados, uma das características de sistemas de tempo real. A sistemática apresentada aumenta o processamento inerente à comunicação devido à geração de eventos de log que registram os tipos semânticos. Assim sendo, a aplicação da sistemática apresentada não é recomendada para esta classe de sistemas, dado que a possível diminuição de desempenho pode ser influenciar na qualidade de serviço destes sistemas.

5 Aplicação de sistemática no *middleware* ROS (Robot Operating System)

Existem inúmeras classes de middlewares responsáveis pela comunicação entre componentes. Um dos middlewares existentes é o ROS (ROS, 2014) (Robot Operation System). Embora seja chamado de sistema operacional, ROS é uma camada de implementação que utiliza chamadas de sistemas operacionais de terceiros para a criação de uma infra-estrutura que possibilita a interação entre componentes.

ROS possui um componente central, chamado nó Master, o qual é responsável por registrar, encontrar e prover a comunicação entre componentes. A comunicação entre componentes é realizada através de trocas de mensagens, seja através da criação de tópicos ou serviços. Enquanto a troca de mensagens via serviços promove a comunicação bidirecional entre um par de componentes, a troca de mensagens via tópicos promove a comunicação entre muitos componentes, utilizando a semântica Publisher/Subscriber. No caso de tópicos, podem existir inúmeros componentes que publicam informações em um tópico, assim como inúmeros componentes podem se cadastrar para receber as informações publicadas. Ambos mecanismos de comunicação estão representados na figura a seguir:

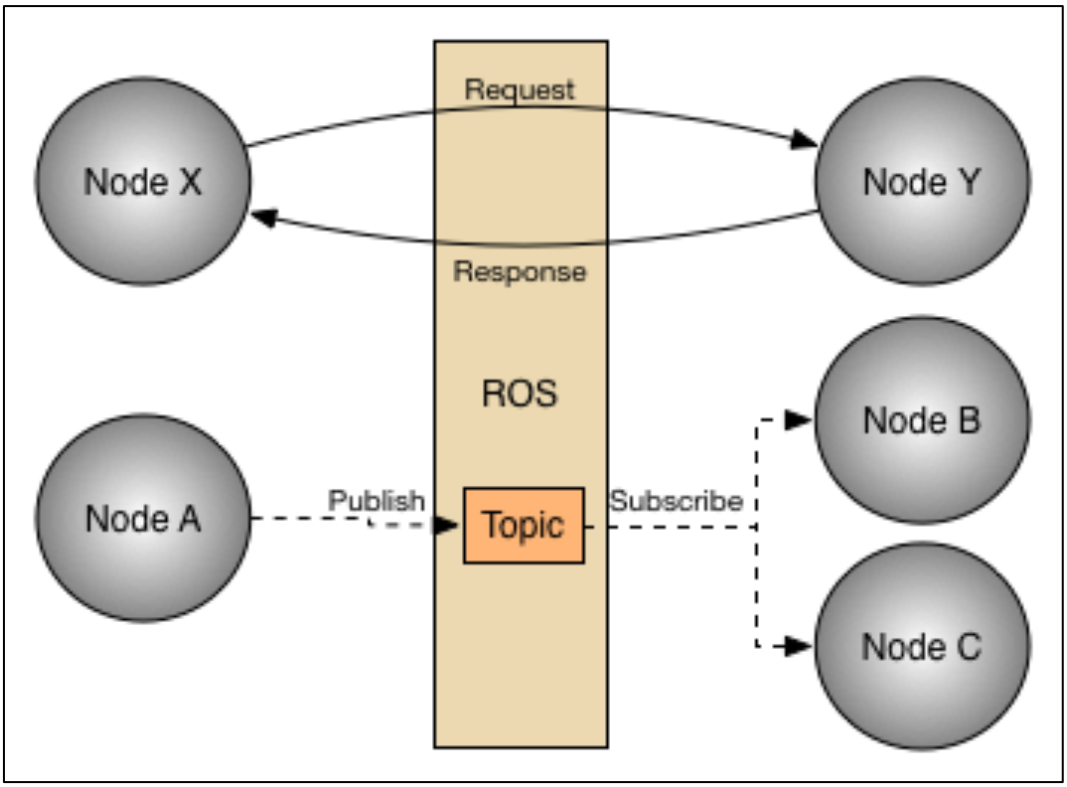

Figura 4 - Mecanismos de comunicação ROS

Em ambas as formas de comunicação, ROS restringe os tipos sintáticos que compõem as mensagens que podem ser transmitidas entre componentes. A criação de tópicos e serviços estão atreladas ao registro de um arquivo de descrição de estrutura de mensagem.

Sendo assim, apenas a estrutura sintática da mensagem é verificada durante a troca de mensagens. Este fato expõe vulnerabilidades relacionadas à comunicação entre componentes, uma vez que a verificação de estrutura sintática não é suficiente para a identificação de falhas relacionadas à incompatibilidade semântica de tipos.

Portanto, apresentamos uma solução para a identificação de falhas de tipos semânticos que consiste na geração de logs estruturados através da interceptação de mensagens trocadas entre componentes e posterior avaliação deste logs por uma ferramenta de verificação de contratos.

5.1 Definição de tipos semânticos de mensagens ROS

Conforme apresentado, a comunicação entre componentes do middleware ROS é realizada através de troca de mensagens. O formato das mensagens utilizadas por componentes ROS é definido através de arquivos de descrição de

mensagem. Arquivos de descrição de mensagens possuem a extensão .msg e são armazenados no diretório /msg de cada pacote. As mensagens utilizadas para serviços possuem arquivos de descrição com estrutura semelhante. Estes arquivos tem a extensão .srv e são armazenados no diretório /srv.

As descrições de mensagem são realizadas sob a forma de texto, onde cada linha define um campo da mensagem com tipo e nome separados por um espaço. No caso do arquivo de descrição de mensagens de serviço, os parâmetros são divididos em duas seções. A primeira seção contem os parâmetros correspondentes à mensagem de requisição do serviço e a segunda seção, os parâmetros que compõem a mensagem de resposta. Neste caso, as duas seções são separadas por uma linha contendo a seguinte sequência de caracteres: “---“.

Os tipos de campo da mensagem podem assumir quatro formas:

1. Tipos básicos, os quais são representados por um alias que torna a descrição de mensagem independente das linguagens de implementação de cada componente. Os tipos básicos suportados são: bool, int8, uint8, int16, uint16, int32, uint32, int64, uint64, float32, float64, string, time e duration.
2. Tipos definidos na própria mensagem, para o caso de composição de tipos de mensagens.
3. Arrays de tamanho fixo ou variável
4. O tipo especial Header, o qual é mapeado na mensagem std_msg/Header.

Considerando o exemplo de elevador sustentável apresentado anteriormente, o arquivo de descrição de mensagem seria Weight.msg, contendo apenas o seguinte campo:

float32 value.

Como mencionado anteriormente, esta forma de descrição de campos de mensagens apenas nos possibilita sua definição de estrutura sintática. A estrutura sintática destes campos fornece chaves para a seleção da forma de serialização de cada um dos campos, dada a linguagem de programação utilizada na implementação de cada um dos componentes participantes da comunicação. Este tipo de definição estrutural é importante para a interoperabilidade entre componentes implementados em diferentes linguagens de programação. No entanto, esta forma de descrição não fornece informações suficientes para a identificação do que os tipos de dados representam no mundo

real, ou seja, não apresentam informações relacionadas a semântica dos tipos de cada campo.

Sendo assim, desenvolvemos uma extensão do arquivo de descrição de mensagens para possibilitar a indicação de tipos semânticos de cada um dos campos de mensagem. Para tanto, criamos uma nova extensão de arquivos: sm_msg. Arquivos com esta extensão poderão ter um alias, de livre escolha, para a indicação de tipo semântico de cada um dos campos de mensagem.

Revisitando o exemplo de elevador sustentável, o arquivo de descrição de mensagem seria Weight.sm_msg, contendo o seguinte campo:

Kgf float32 value.

Neste caso, a mensagem está relacionada a grandeza Kgf. Dessa maneira, sabe-se que é esperado que componentes que transmitam ou recebam mensagens deste tipo forneçam ou consumam valores dispostos na mesma grandeza.

5.2 Adaptação de processo de cadastro de mensagens com tipos semânticos

A transmissão de mensagens no middleware ROS supõe o cadastro do tipo de mensagem a ser transmitida e a habilitação de diretivas para a serialização das novas mensagens. O cadastro da uma nova mensagem é realizado em três etapas:

1. Inserção de diretivas no arquivo package.xml, responsável por transformar os arquivos de mensagem em código fonte para a serialização para as diferentes linguagens suportadas pelo middleware. As diretivas a serem adicionadas são:

<build_depend>message_generation</build_depend>

<run_depend>message_runtime</run_depend>

2. Inserção de diretivas no arquivo CMakeLists.txt, responsável pelo controle de dependências dos componentes contidos em um pacote. Neste caso, devem ser adicionadas as diretivas:

```
find_package(catkin REQUIRED COMPONENTS
...
message_generation)
```

```
catkin_package(
...
CATKIN_DEPENDS message_runtime
...)
```

```
add_message_files(
FILES
<Nome_arquivo_nova_mensagem>)
```

Figura 5 - Diretivas a serem inseridas no arquivo CMakeLists.txt

Após estas alterações, a nova mensagem será registrada na próxima vez que os componentes forem recompilados.

A adição de tipos semânticos apresentada na seção anterior invalida o processo de cadastro de mensagens disponibilizado pelo middleware. Assim sendo, criamos um mecanismo que nos possibilita a criação destas mensagens através de uma extensão do mecanismo de cadastro disponibilizado previamente por ROS.

A extensão consistiu na criação de um novo processo de cadastro de mensagem, que processa o novo formato de arquivo de descrição de mensagem. O processamento gera um arquivo de estrutura de mensagens esperado por ROS, e cadastra este novo tipo de mensagem no *middleware*.

5.3 Interceptação de comunicação entre componentes ROS

Conforme apresentado, a comunicação entre componentes do middleware ROS é realizada através de troca de mensagens e este mecanismo é disponibilizado pelo middleware através de tópicos e serviços. Tópicos e serviços são criados através da utilização de bibliotecas disponibilizadas pelo middleware ROS para cada uma das linguagens suportadas para o desenvolvimento de componentes.

Considerando que falhas de inconsistência semântica de tipos são intrínsecas da representação de características do mundo real através de código, torna-se necessário analisar as causas destas falhas através do código de cada componente.

Sendo assim, a interceptação de comunicação não será realizada a nível de middleware, mas através da extensão de bibliotecas disponibilizadas por ele,

para que características da linguagem de programação utilizada e do próprio código que origina a comunicação possam ser consideradas.

5.4 Exemplo de criação de tópico em ROS utilizando a linguagem Python

Uma das linguagens suportadas para a criação de componentes em ROS é a linguagem Python (Python, 2013), a foi escolhida como foco de implementação da sistemática apresentada. Neste contexto, os componentes devem utilizar a biblioteca rospy{/cite} para a criação de diretivas dos tipos Publisher ou Subscriber para a comunicação através de tópicos. A seguir apresentaremos um exemplo de código de componentes que se comunicam através de um tópico chamado ‘chatter’. (ROS, 2014)

```
import rospy
from std_msgs.msg import String
def talker():
        pub = rospy.Publisher('chatter', String, queue_size=10)
        rospy.init_node('talker', anonymous=True)
        rate = rospy.Rate(10) # 10hz
        while not rospy.is_shutdown():
                hello_str = "hello world %s" % rospy.get_time()
                rospy.loginfo(hello_str)
                pub.publish(hello_str)
                rate.sleep()

 if __name__ == '__main__':
    try:
        talker()
        except rospy.ROSInterruptException:
        pass
```

Figura 6 - Código de componente que publica informações no tópico *chatter*

Neste caso, o trecho apresentado acima é responsável pela criação do tópico ‘chatter’ através da criação de um nó chamado “talker e um objeto do tipo rospy.Publisher, o qual é responsável por transmitir mensagens do tipo String à uma taxa de 10 Hertz. A publicação de mensagem é realizada através do método publish e a taxa de transmissão é assegurada através da linha rate.sleep(), a qual faz com que o componente suspenda sua execução por 0.1 segundos.

```
import rospy
from std_msgs.msg
import String

def callback(data):
   rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():
        rospy.init_node('listener', anonymous=True)
        rospy.Subscriber("chatter", String, callback)
        rospy.spin()

if __name__ == '__main__':
   listener()
```

Figura 7 - Código de componente que recebe as informações publicadas no tópico *chatter*

O trecho acima, cria um nó chamado “listener” do tipo Subscriber para o tópico ‘chatter’ através da diretiva rospy.Subscriber. Este nó recebe mensagens do tipo String, executando o método *callback* sempre que uma nova mensagem for recebida.

5.5 Exemplo de criação de serviço ROS utilizando a linguagem Python

Ainda utilizando a linguagem Python, apresentaremos um exemplo de criação de um serviço que adiciona dois números e um cliente que consome este serviço, imprimindo o resultado.

```
from services.srv import *
import rospy

def handle_add_two_ints(req):
   print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b))
   return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
   s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
   print "Ready to add two ints."
   rospy.spin()

if __name__ == "__main__":
   add_two_ints_server()
```

Figura 8 - Código de componente que provê serviço de adição de inteiros

No trecho de código acima, o componente cria o serviço chamado “add_two_ints” através do método rospy.Service. Este serviço utiliza o tipo de mensagem AddTwoInts, executando o método handle_add_two_ints sempre que um componente cliente faz uma requisição deste serviço.

A seguir, o código de um cliente do serviço que soma dois inteiros:

```
import sys
import rospy
from services.srv import *

def add_two_ints_client(x, y):
  rospy.wait_for_service('add_two_ints')
  try:
     add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp1 = add_two_ints(x, y)
     return resp1.sum
  except rospy.ServiceException, e:
     print "Service call failed: %s"%e

def usage():
  return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":
  if len(sys.argv) == 3:
     x = int(sys.argv[1])
     y = int(sys.argv[2])
   else:
     print usage()
     sys.exit(1)
  print "Requesting %s+%s"%(x, y)
  print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

Figura 9 - Código de componente que utiliza serviço de adição de inteiros

O trecho de código acima apresenta um componente que utiliza o serviço AddTwoInts. Este serviço utiliza dois campos relacionados aos valores inteiros que serão somados e um campo correspondente ao campo de resposta: sum. Neste exemplo, o componente cria um rospy.ServiceProxy que fornece uma interface de acesso ao serviço de forma síncrona.

Assim sendo, a interceptação de mensagens será realizada através da criação de uma nova camada de implementação dos métodos rospy.Publisher, publish, rospy.Subscriber e rospy.Service. Os códigos de implementação de componentes ROS devem, então, utilizar os novos métodos, chamados SEM_Publisher, SEM_publish, SEM_Subscriber e SEM_Service, respectivamente.

O método SEM_Publisher é responsável pelo cadastro dos tipos semânticos da mensagem cadastrada. O cadastro é realizado através de um serviço, em um nó responsável pela agregação de informações dos tipos que descrevem a mensagem cadastrada. Já o método SEM_publish realiza a análise de tipos semânticos dos parâmetros fornecidos na publicação da mensagem. O método SEM_Subscriber é responsável pela geração do log correspondente à comunicação entre o par de componentes e o método SEM_Service cadastra um novo método para o tratamento de requisições. Este método processa os tipos semânticos relacionados aos objetos request e response e gera um evento de log que descreve os tipos semânticos da conexão corrente.

5.6 Análise de tipos semânticos publicados

A verificação de tipos semânticos na troca de mensagens é realizada através da comparação entre tipos semânticos de parâmetros efetivamente publicados e os respectivos tipos semânticos esperados por componentes que receberão as mensagens enviadas.

Os tipos semânticos esperados são definidos através do arquivo de descrição de mensagens, cuja estrutura foi apresentada anteriormente. No entanto, o arquivo de descrição de mensagem não é suficiente para fornecer os tipos semânticos das mensagens publicadas. Isto ocorre porque os tipos semânticos não são interpretados pelas linguagens de programação como tipos básicos, o que faz com que atribuições errôneas não sejam apresentadas como erro de sintaxe.

Dessa maneira, é necessário realizar a análise sintática de tipos semânticos para a identificação dos tipos correspondentes aos parâmetros utilizados na publicação de mensagens. A solução apresentada para esta questão é a criação de uma árvore sintática do código de cada componente e posterior associação de variáveis com os tipos semânticos definidos nos arquivos de descrição de mensagens.

Após a atribuição de tipos semânticos em todas os símbolos terminais da árvore de sintaxe abstrata, o código é analisado para identificar possíveis conversões de tipos. Estas informações são disponibilizadas ao usuário para a criação de conhecimento que será aplicado posteriormente.

5.7 Geração de eventos de log com tipos semânticos

Uma vez que os tipos semânticos dos valores que constituem uma mensagem foram identificados, iniciamos a etapa de geração de logs estruturados. (Araújo et al., 2014) apresentam um mecanismo de apoio a diagnose para sistemas distribuídos. O mecanismo baseia-se na instrumentação do código dos componentes do sistema visando a geração de informações que retratam o estado e o contexto ao longo da execução dos componentes do sistema-alvo. A técnica utilizada para geração de informações contextuais é a criação de logs estruturados baseados em eventos. Utilizamos a forma de representação de informações neste trabalho por conta da flexibilidade proporcionada pela estrutura de logs, apresentada na próxima seção.

5.7.1 Estrutura de informações

O trabalho (Araújo et al, 2014) apresenta um mecanismo de apoio a diagnose cuja unidade básica é o evento, representado através de um conjunto de duplas <Tag, Valor>, um identificador temporal e uma mensagem. Tanto o identificador temporal como a mensagem são considerados tags. A figura 6 ilustra a organização da estrutura de um evento.

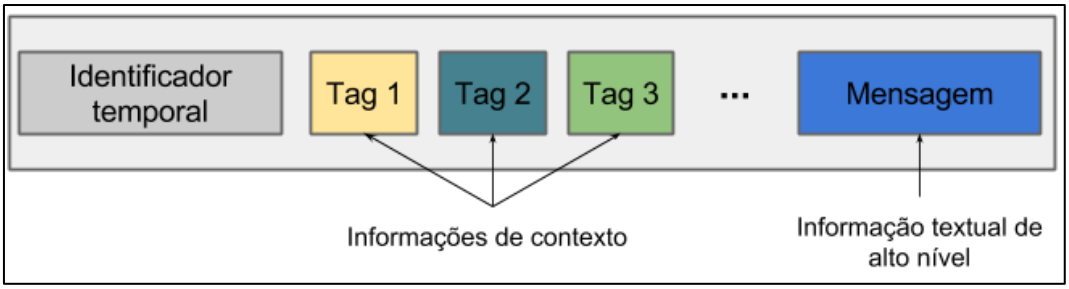

Figura 10 - Estrutura de eventos de log

Cada evento possui informações referentes ao contexto de execução em que ele foi gerado. A seguir, apresentamos um exemplo de evento em sua forma serializada. O evento apresentado é composto por uma sequência de tuplas com uma quebra de linha no final.

[**application**:hello world] [**environment**:server][**component**:web server] [**request** id:1234] [**clientid**:4321] [**operation**:processing client request][**message**: verifying client settings]\n

5.7.2 Bibliotecas de apoio à instrumentação para geração de logs estruturados

Assim como propõe a estrutura de eventos de logs, o trabalho de Araújo também apresenta bibliotecas de instrumentação que apoiam a geração de logs estruturados conforme a estrutura definida. Estas bibliotecas possuem um grande conjunto de funcionalidades para a criação de logs com informações contextuais, as quais facilitam o processo de instrumentação quando realizado manualmente. Estas mesmas bibliotecas podem ser utilizadas para a geração de logs estruturados para outros fins, já que um pequeno conjunto de funcionalidades destas bibliotecas já é suficiente para a geração de eventos de logs conforme a estrutura apresentada.

Sendo assim, utilizamos uma das bibliotecas de apoio à instrumentação apresentadas no trabalho de Araújo para a geração de eventos com os tipos semânticos. Através da biblioteca produzimos eventos que contém tipos

semânticos de parâmetros referentes às mensagens publicadas e respectivos tipos semânticos esperados por componentes que estão cadastrados para receberem estas mesmas mensagens.

5.8 Análise de eventos de log através de verificação de contratos

Uma vez que os eventos tenham sido gerados, é necessário realizar verificações para a identificação de falhas de inconsistências semântica de tipos. Idealmente, a identificação de inconsistências deveria ser realizada em tempo de execução para que o sistema de componentes seja capaz de incidir sobre o próprio sistema, seja para minimizar as consequências do erro identificado ou mesmo, promover uma rotina de recuperação adequada.

A abordagem utilizada para geração de informações de tipos semânticos não restringe os tipos semânticos à um conjunto fixo. Essa flexibilidade quanto à nomenclatura removeu a necessidade de padronização, o que facilita a paralelização de desenvolvimento de componentes. No entanto, a falta de padronização pode gerar um grande número de falsos-positivos nos momentos iniciais de uso do sistema de componentes. Por conta deste fato, optamos por realizar a verificação de tipos semânticos pós-execução, já que um alto número de falsos-positivos pode fazer com que haja a execução errônea de mecanismos de recuperação, pode aumentar demasiadamente o *overhead* causado pela sistemática em geral, ou ainda, aumentar a taxa de rejeição relacionada aos usuários da sistemática. Assim sendo, utilizaremos a ferramenta de (Rocha & Staa, 2014) para a identificação de anomalias com base em contratos.

5.8.1 Mecanismo de verificação de contratos baseado em logs estruturados

O trabalho de (Rocha & Staa, 2014) apresenta um mecanismo para verificação de contratos em sistemas distribuídos. Este trabalho supõe logs estruturados conforme a estrutura apresentada no trabalho de Araújo (2014), logs estes, que contêm informações contextuais sobre a execução do sistema.

O log utilizado pela ferramenta de verificação de contratos é centralizado, ou seja, todos os processos enviam os eventos para esse log centralizado que então ordena os eventos pelo timestamp. Na recepção de um pacote de eventos é feita a correção de eventuais diferenças entre os relógios dos processos geradores. Uma restrição inerente a esse mecanismo é a precisão dos diferentes relógios, restrição esta, que não influencia a sistemática de monitoramento apresentada, já que não prevê verificação de eventos temporais.

O mecanismo de Rocha (Rocha & Staa, 2014) promove a verificação de contratos descritos segundo uma gramática própria, executando rotinas de notificação sempre que um dos contratos não é obedecido. Este mecanismo possibilita tanto a identificação automática de falhas quanto a identificação manual, realizada por um mantenedor do sistema. A seguir, uma ilustração do mecanismo proposto por Rocha (Rocha & Staa, 2014):

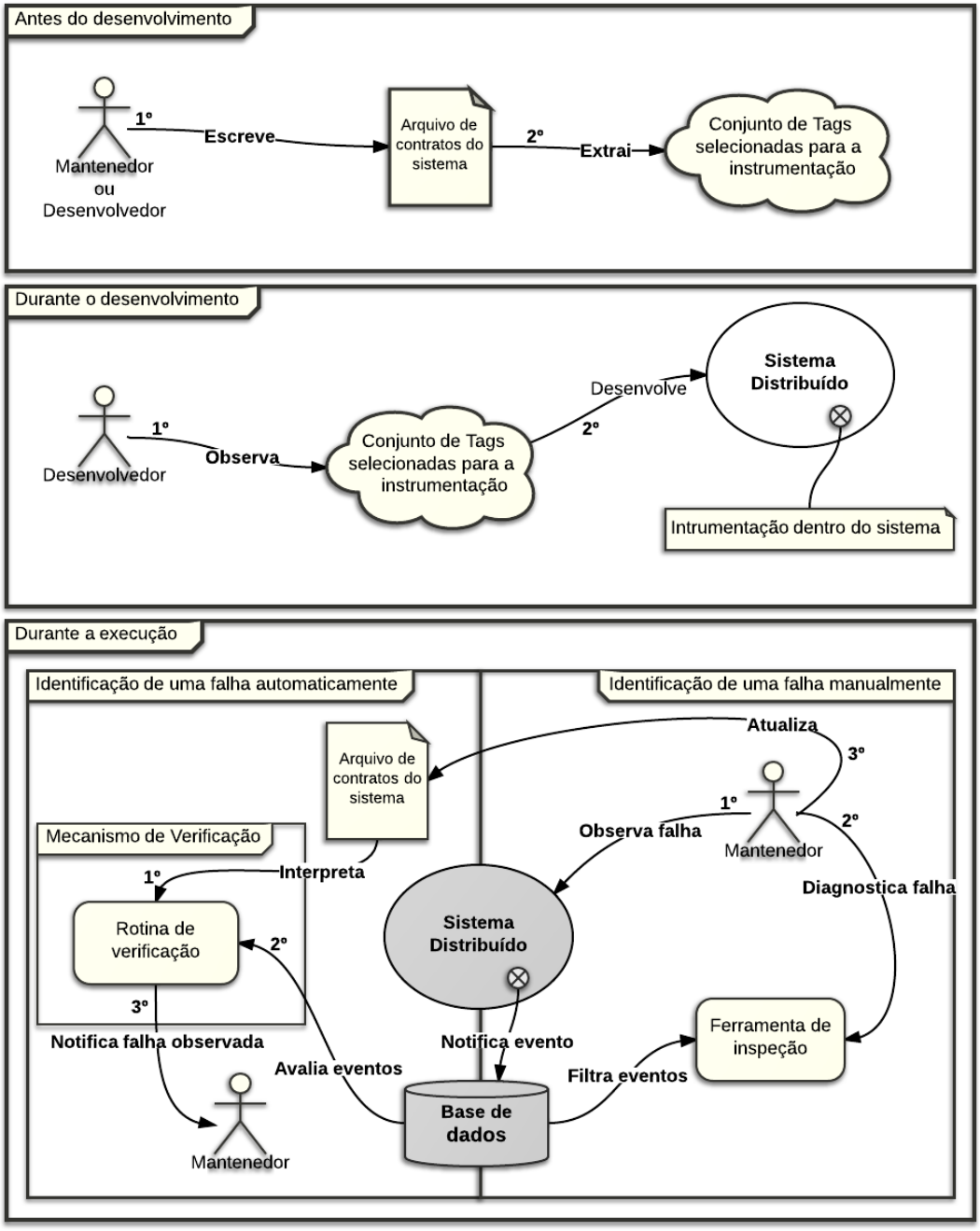

Figura 11 - Mecanismo de verificação proposto por (Rocha & Staa, 2014)

Ainda que o mecanismo apresentado tenha sido apresentado com o escopo de utilização em sistemas distribuídos, não há indícios de que sua utilização esteja restrita a esta classe de sistemas, uma vez que a abordagem apresentada não

assume características de sistemas massivamente distribuídos para sua aplicação. Além disso, o mecanismo apresentado inclui uma gramática suficientemente abrangente para suportar a verificação de contratos em logs estruturados conforme os eventos de Araújo et al, sem que haja a necessidade de que estes logs contenham informações do contexto de execução do sistema-alvo.

Portanto, utilizamos este mecanismo como base para a identificação de possíveis falhas de inconsistência de tipos semânticos e posterior notificação ao responsável pela manutenção do sistema.

5.8.1.1 Política de alimentação de contratos

A etapa de identificação desta classe de falhas consiste na criação de contratos que verifiquem a igualdade entre duplas de tags que representam os tipos semânticos de campos de registro de cada instância de comunicação entre componentes. O processo de identificação de falhas poderia ser realizado com a verificação de apenas um contrato que verificasse pares de tags em um mesmo evento, porém a gramática proposta por (Rocha & Staa, 2014) não prevê tal caso.

Sendo assim, adotamos uma política de alimentação de contratos que insere um novo contrato sempre que a instância de comunicação utilizar uma mensagem com número de campos ainda não suportado. A política de alimentação de contratos está representada na figura a seguir:

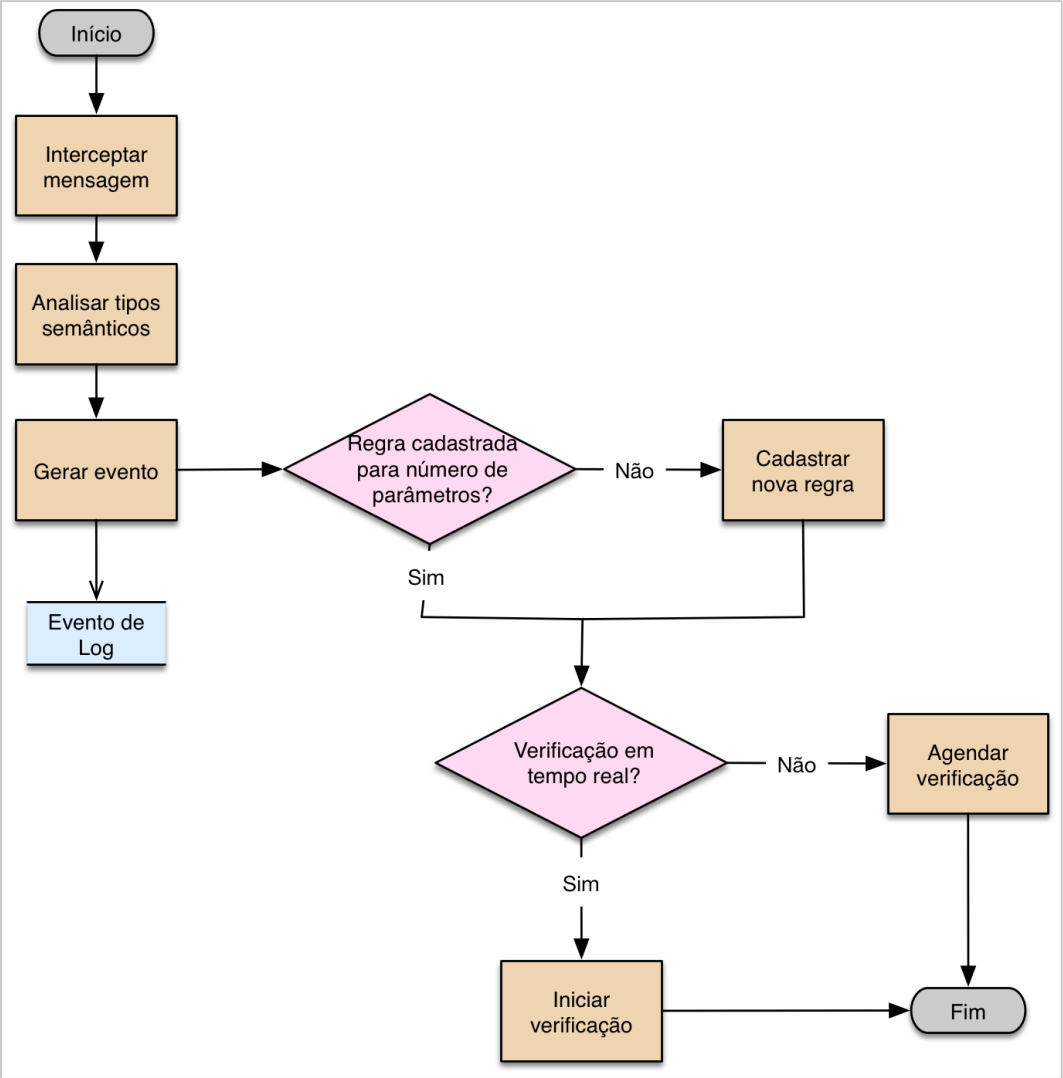

Figura 12 - Política de alimentação de contratos

Um exemplo para o caso de publicação de uma mensagem com apenas 1 parâmetro. O contrato adicionado:

```
SINGLE_EVENT_EXPRESSION ->
        {{$exists: 'f1_type_t'},
        {$exists: 'f1_type_r'},
        {$if: {$compare: 'f1_type_t', ==, 'f1_type_r'}}}
```

Figura 13 - Contrato que verifica a publicação de mensagem com 1 parâmetro

Consecutivamente, caso a mensagem publicada possua 2 parâmetros, os seguinte conjunto de contratos será inserido:

```
SINGLE_EVENT_EXPRESSION ->
        {{$exists: 'f1_type_t'},
        {$exists: 'f1_type_r'},
        {$exists: 'f2_type_t'},
        {$exists: 'f2_type_r'},
        {$if: {$and:
                {$compare: 'f1_type_t', ==, 'f1_type_r'},
                {$compare: 'f2_type_t', ==, 'f2_type_r'}}}}
```

Figura 14 - Contrato que verifica a publicação de mensagens com 2 parâmetros

5.8.1.2 Rotina de notificação

A identificação automática de falhas prevê a execução de uma rotina de notificação sempre que um contrato não é satisfeito. Conforme mencionado, a ausência de um padrão de nomeação de tipos semânticos pode gerar falsos positivos, uma vez que um mesmo tipo semântico pode assumir diversas nomenclaturas. Portanto, é necessário promover formas de diminuir a incidência de falsos positivos, já que em excesso, podem inviabilizar a utilização da sistemática apresentada em sistemas reais.

Assim sendo, a rotina de notificação de falhas de inconsistência semântica de tipos foi projetada segundo três objetivos:

1. Atestar a ocorrência de uma anomalia de inconsistência semântica decorrente da comunicação entre componentes.
2. Prover informações de auxílio ao diagnóstico de sua causa.
3. Possibilitar que o mantenedor do sistema reporte falsos positivos para manutenção de conhecimento.

Dessa forma, criamos uma rotina de notificação que gera um relatório que contém:

- Nome do tópico em que a inconsistência ocorreu.
- Nomes dos nós que participaram da comunicação que gerou a inconsistência semântica.
- Nomes de tipos semânticos que originaram a inconsistência.

O atestado de ocorrência de falha é provido através do relatório apresentado e os nomes de instâncias fornecidos auxiliam o diagnóstico da causa da falha, pois identificam os componentes envolvidos.

A identificação de falsos positivos utiliza uma técnica de Raciocínio Baseado em Casos (Ni et al., 2003), que consiste na utilização de soluções de experiências passadas para a solução de novos problemas. Neste caso, a identificação de falsos positivos é realizada pelo mantenedor através da comparação entre os nomes de tipos semânticos fornecidos no relatório. Quando os tipos semânticos

fornecidos representam o mesmo conceito semântico, como é o caso das unidades *m_s* e *meter_per_second* apresentados anteriormente em um exemplo, o mantenedor deve alterar a extensão do arquivo de relatório para *.fp* (false-positive). Um componente do sistema é responsável por coletar periodicamente os arquivos de extensão *.fp* e inserir as duplas de tipos presentes em um dicionário de tipos semânticos. Este dicionário contem as equivalências de nomenclaturas de tipos semânticos fornecidas pelo mantenedor. Equivalências estas que são consultadas sempre que uma nova anomalia semântica é identificada, impedindo a geração de um novo relatório quando a anomalia estiver cadastrada no dicionário de tipos e consequentemente, diminuindo a incidência de falsos-positivos ao longo do tempo.

6 Avaliação

A avaliação da sistemática apresentada foi realizada através de uma prova de conceito. Tal prova de conceito consiste na aplicação da sistemática de monitoramento em um sistema de componentes e posterior validação de sua eficácia quanto à identificação de falhas decorrentes de inconsistência semântica de tipos. O sistema de aplicação desta prova de conceito tem sua arquitetura de comunicação baseada em um sistema real, o qual representa um sistema de locomoção de um robô híbrido.

6.1 Sistema de locomoção de robô híbrido

O sistema utilizado para a validação da sistemática proposta foi baseado em um sistema de componentes que promove a locomoção de um robô híbrido. Tal robô foi projetado para transitar em diferentes tipos de ambientes: terra, água e meios escorregadios, como a lama. O sistema que controla este robô possui inúmeros componentes, mas o sistema de locomoção é um dos subsistemas que apresentam um grande risco decorrente de falhas de inconsistência semântica de tipos e por consequência, foi o subsistema escolhido para a prova de conceito.

O sistema de locomoção de um robô envolve componentes de hardware e software. Quanto ao hardware, por vias de simplificação, consideraremos apenas rodas, direção e sensores de identificação de meio, como terra, água e lama. Os componentes de hardware possuem componentes de software associados, os *drivers*. Na figura a seguir, apresentamos a arquitetura de componentes do sistema de locomoção para uma das rodas do robô. As figuras circulares e cinzas representam componentes ROS, assim como as caixas brancas representam *drivers* e as caixas laranjas, tópicos ROS. Todas as mensagens e respectivos tipos sintáticos e semânticos também se encontram na figura.

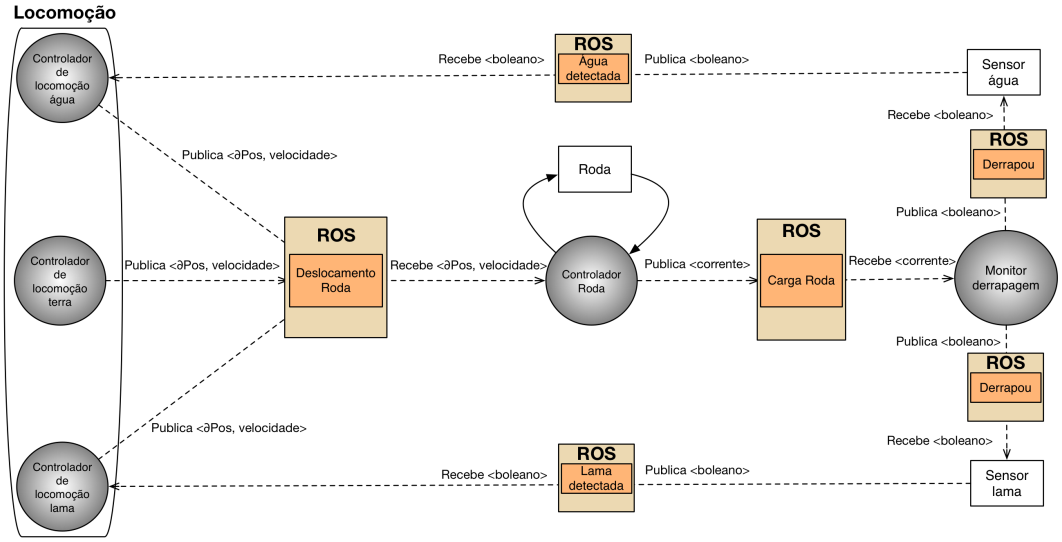

Figura 15 - Arquitetura de componentes para a locomoção de rodas

O ambiente padrão de locomoção do robô é a terra. O componente controlador de locomoção em terra, considerando a trajetória do robô, publica mensagens contendo o deslocamento que a roda deverá realizar segundo uma velocidade. Cada roda, por sua vez, é controlado por um componente, chamado controlador de roda. O controlador de roda recebe a mensagem transmitida e traduz o deslocamento para o *driver* de maneira que este deslocamento seja aplicado na roda.

Durante a movimentação da roda, o componente controlador de roda publica a corrente instantânea aplicada ao motor para realizar o movimento. Esta mensagem é constantemente analisada pelo componente monitor de derrapagem. Através do monitoramento de corrente, o monitor de derrapagem detecta um evento de derrapagem, publicando mensagens de ativação para os sensores de detecção de água e lama. Ao detectar seu respectivo meio, o sensor publica uma mensagem que resulta na ativação de um controlador de locomoção específico para o meio detectado. Este controlador assume o papel de controlador principal e inicia a publicação de mensagens no tópico referente ao deslocamento de roda, através de um cálculo específico para o meio corrente.

Após o entendimento do funcionamento de uma das rodas, apresentamos um diagrama que representa todas as rodas e componentes de direção:

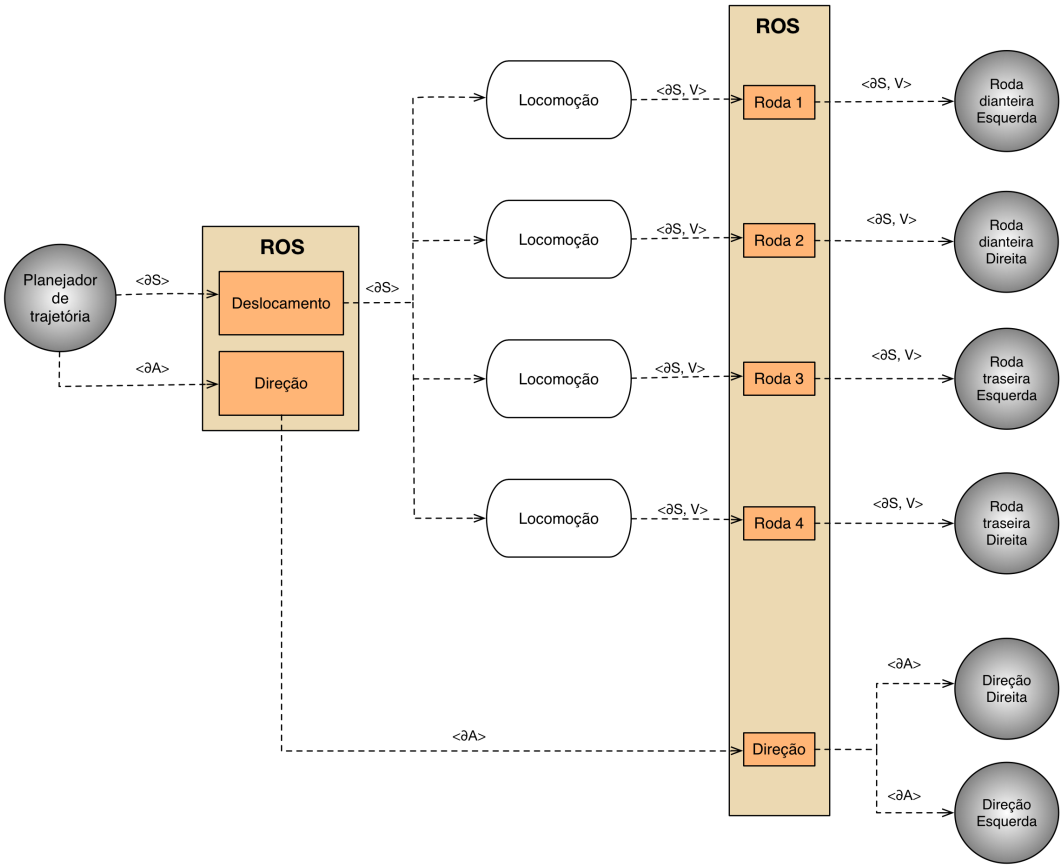

Figura 16 - Arquitetura de componentes para a locomoção de um robô híbrido

Neste caso, o planejador de trajetória publica mensagens de deslocamento e angulação em relação ao mundo. Mensagens estas que são repassadas aos componentes de controle de locomoção apresentado anteriormente e controle de direção.

6.2 Falhas de inconsistência semântica de tipos decorrentes da comunicação entre componentes do sistema de locomoção

Ao analisar os componentes de controle de locomoção em diferentes meios (terra, água e lama), podemos perceber que todos publicam o mesmo tipo sintático de mensagem <∂Pos, velocidade>. Considere que cada um utilizou suas próprias unidades de medida para representação de deslocamento e velocidade. Assim sendo, a quebra de contrato semântico é estabelecida assim que um dos componentes semanticamente incompatíveis é acionado. Esta classe de falhas foi considerada a base para a validação da sistemática apresentada.

6.3 Validação de eficácia de sistemática de monitoramento de falhas

A etapa de validação da eficácia consistiu em 3 etapas:

1. Implementação do sistema de componentes de locomoção de robô híbrido.
2. Reconfiguração do sistema de componentes com o objetivo de injetar falhas do tipo descrito.
3. Comparação entre total de falhas simuladas e total de falhas identificadas.

Considerando a arquitetura de componentes apresentada para o sistema de locomoção de um robô híbrido, foram criados 3 diferentes tipos de componentes controladores de locomoção:

1. Componente controlador em ambiente aquático
2. Componente controlador em ambiente terrestre
3. Componente controlador em condição adversa (lama)

Cada componente foi implementado conforme uma unidade semântica de mensagem. Neste caso, as unidades semânticas estão relacionadas a grandezas físicas. A seguir estão os componentes e os tipos semântico das mensagens publicadas por eles:

1. Componente controlador em ambiente aquático publica a posição em metros e velocidade em metros/segundo <m, m_s>.
2. Componente controlador em ambiente terrestre publica a posição em quilômetros e velocidade em quilômetros/hora <km, km_h>.
3. Componente controlador em condição adversa (lama) publica a posição em centímetros e velocidade em centímetros/segundo <cm, cm_s>.

O componentes controlador da roda foi projetado para receber mensagens conforme os tipos semânticos: a posição em metros e velocidade em metros/segundo <m, m_s>.

A prova de conceito consistiu na configuração da arquitetura de componentes para uma das rodas do robô, promovendo a troca entre componentes controladores de cada um dos ambientes.

6.4 Resultados

Para cada um dos três cenários apresentados, os resultados obtidos foram iguais aos esperados:

Cenário 1:

Inserção de controlador em ambiente aquático no sistema de componentes.

Mensagens publicadas não geraram relatórios de falhas. Este resultado era esperado já que as mensagens publicadas obedeciam a mesma interface semântica utilizada pelo componente controlador de roda.

Cenário 2:

Inserção de controlador em ambiente terrestre no sistema de componentes.

Mensagens publicadas geraram relatórios de falhas com as seguintes informações:

topic_name: wheel_offset

publisher_node: terrestrial_environment_controller

subscriber_node: wheel_controller

published_semantic_types: [km, km_h]

subscribed_semantic_types: [m, m_s]

Este resultado era esperado já que as mensagens publicadas obedeciam uma interface semântica diferente da interface utilizada pelo componente controlador de roda.

Cenário 3:

Inserção de controlador em ambiente em condição adversa (lama) no sistema de componentes.

Mensagens publicadas geraram relatórios de falhas com as seguintes informações:

topic_name: wheel_offset

publisher_node: mud_environment_controller

subscriber_node: wheel_controller

published_semantic_types: [cm, cm_s]

subscribed_semantic_types: [m, m_s]

Assim como no cenário 2, este resultado era esperado já que as mensagens publicadas obedeciam uma interface semântica diferente da interface utilizada pelo componente controlador de roda.

Sendo assim, a prova de conceito apresentou resultados que nos possibilita verificar que a sistemática foi capaz de identificar falhas de inconsistência semântica de tipos, bem como gerou informações que possibilitaram a localização da causa da falha. Entretanto, faz-se necessário avaliar o impacto da

utilização da sistemática no desempenho de sistemas reais, etapa a ser realizada em trabalhos futuros.

7 Conclusão

Durante esse trabalho foi desenvolvida uma sistemática para identificação de falhas decorrente de uso incorreto de tipos semânticos em sistemas distribuídos. A sistemática consiste na definição de tipos semânticos para as mensagens trocadas entre componentes, interceptação da comunicação e verificação de contratos segundo o mecanismo de (Rocha & Staa, 2014).

A sistemática foi implantada no middleware ROS e validada em um sistema de componentes que representa o controle de locomoção de um robô híbrido, cuja arquitetura de comunicação foi baseada na arquitetura de um sistema real. A eficácia da sistemática apresentada foi avaliada através de uma prova de conceito que consistiu na injeção sistemática de falhas de inconsistência semântica de tipos no sistema de locomoção mencionado e posterior validação de identificação. Os resultados obtidos demonstram que a sistemática apresentada é promissora devido à capacidade de identificação de falhas já conhecidas, conforme o esperado.

Os pontos fortes da sistemática de monitoramento apresentada são:

- A identificação precoce de erros de inconsistência semântica de tipos decorrentes da comunicação errônea entre componentes.
- Possibilidade de atuação para mitigação de riscos decorrentes de falhas deste tipo, as quais podem ser catastróficas ao considerarmos sistemas de missão crítica.
- Fornecimento de informações que indicam os componentes, tópicos e tipos de dados envolvidos, aumentando a rastreabilidade da causa da falha.
- Possibilidade de diminuição de falsos-positivos ao longo da vida do sistema, através do mecanismo apresentando que possibilita que o mantenedor identifique os falsos-positivos encontrados para o sistema.
- Reuso da infraestrutura desenvolvida no *middleware* para posterior aplicação da sistemática em outros sistemas que o utilizam para comunicação.

- Reuso do conhecimento adquirido relacionado à equivalência de nomenclatura de tipos semânticos para prevenir a ocorrência de falsos-positivos.

Já os pontos fracos da sistemática apresentada são:

- Alto esforço na implementação em um novo *middleware*, dado que deve-se considerar suas especificidades como linguagens suportadas, forma de representação de interfaces de comunicação entre componentes, entre outros pontos.
- Overhead ocasionado pela geração de logs e publicação de mensagens para o componente responsável pelo gerenciamento de tipos semânticos.
- Possibilidade de ocorrência de grande número de falsos-positivos no estágio inicial de implantação em sistemas reais.

Finalmente, alguns trabalhos futuros decorrentes deste trabalho:

- Implantação em outros sistemas de componentes com avaliação de impacto de utilização desta sistemática em um sistema real.
- Avaliação de impacto da utilização desta sistemática, como: esforço de implantação, *overhead* e número de falsos-positivos ao longo do tempo.
- Criação de mecanismo que possibilita a análise de código do componente que publica informações através da criação de árvores sintáticas abstratas para possibilitar a identificação automática de anomalias no código. Esta abordagem aplicaria técnicas relacionadas a Raciocínio Baseado em Regras (RBR) para a criação de um modelo taxonômico de tipos semânticos, o qual pode ser reutilizado independentemente do sistema em que foi aplicado.

Concluindo, a contribuição mais significativa deste trabalho foi o projeto de uma forma sistematizada de monitoramento de erros de inconsistência semântica de tipos. Tal contribuição tem sua importância devido à falhas desta classe já terem sido responsáveis por catástrofes, como a do Mars Climate Orbiter, uma sonda espacial desenvolvida pela NASA que perdeu o contato com a terra após 1 ano de operação, após uma falha de sistema decorrente da diferença entre sistemas métricos utilizados para representar grandezas físicas. Além disso, não foram encontradas na literatura abordagens semelhantes para o tratamento automático deste tipo de erro.

8 Referências bibliográficas

ABRIAL J.R., **The B-book: assigning programs to meanings**, Cambridge University Press, New York, NY, 1996.

ANDRE, R. de P.; STAA, A. v.; **Avaliação do Uso de Análise Estática na Detecção de Conflitos Semânticos em Tipos de Dados**, Rio de Janeiro, 2013, 95p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro. Disponível em: < http://www2.dbd.puc-rio.br/pergamum/tesesabertas/1021803_2013_completo.pdf>

ARAÚJO, T.; CERQUEIRA, R.; STAA, A. v. **Supporting failure diagnosis with logs containing meta-information annotations.** Rio de Janeiro, 2014, 21 p. Monografias em Ciência da Computação nº 02/14 – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro. ISSN: 0103-9741. Disponível em: < ftp://ftp.inf.puc-rio.br/pub/docs/techreports/14_02_araujo.pdf>

BABAOGLU, O.; DRUMOND, R.; **Streets of Byzantium: Network architectures for fast reliable broadcast.** IEEE Trans. Softw. Eng. SE-I1, 6, (1985).

BARBARA, D.; GARCIA-MOLINA, H.; Spauster, **A. Increasing availability under mutual exclusion constraints with dynamic vote reassignment**. ACM Trans. Comput• Syst. 7, 4 (Nov.1989)•

BERRY, G.; **Synchronous design and verification of critical embedded systems using SCADE and Esterel**, Proceedings of the 12th international conference on Formal methods for industrial critical systems, July 01-02, 2007, Berlin, Germany

CRISTIAN, F; **Probabilistic clock synchronization**. Distributed Computing 3 (1989), 146-158.

DAO D.; ALBRECHT J.; KILLIAN C.; VAHDAT A. **Live Debugging of Distributed Systems**, 2009

DELAMARO M.; MALDONADO, J.; JINO M.; **Introdução ao Teste de Software**. Rio de Janeiro, Elsevier 2007.

E. BORGER; STARK, R. F.; **Abstract State Machines: A Method for High-Level System Design and Analysis**, Springer-Verlag New York, Inc., Secaucus, NJ, 2003

FLOYD, R. 1967. **Assigning meanings to programs.** In Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics, Rhode Island. American Mathematical Society, Providence, RI, 19--32.

JONES, C. B.; **Systematic Software Development Using VDM**, 1986 :Prentice Hall

GEELS, D.; ALTEKAR, G.; MANIATIS, P.; ROSCOE, T.; STOICA, I. **Friday: Global Comprehension for Distributed Replay.** In: Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2007

GHOSH, S.; MATHUR, A. P. **Issues in Testing Distributed Component-based Systems.** In: First International ICSE Workshop on Testing Distributed Componentbased Systems, Los Angeles, 1999

KILLIAN, C.E.; ANDERSON, J.W.; JHALA, R.; VAHDAT, A. **Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code.** In: Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2007

LIU, X.; GUO, Z.; WANG, X.; CHEN, F.; LIAN, X.; TANG, J.; WU, M.; KAASHOEK, M.F.; ZHANG, Z. **D3S: Debugging Deployed Distributed Systems.** In: Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2008

MEYER, B. Applying "**Design by Contract**", Computer, v.25 n.10, p.40-51, October 1992 [doi>10.1109/2.161279]

PFISTER, C.; SZYPERSKI, C.; JELL, T.; **Why objects are not enough, Component-Based Software Engineering** (CUC',96), 1998 :Cambridge Univ. Press

PONSARD, C.; MASSONET, P.; RIFAUT, A.; MOLDEREZ, J.F.; A. van Lamsweerde and H. Tran Van, **Early Verification and Validation of Mission-Critical Systems** *Proc. Workshop Formal Methods in Critical Systems,* 2004.

PYTHON.; Programing Language, version 2.7.5, 2013. Disponível em: https://www.python.org/

REYNOLDS, P.; KILLIAN, C.E.; WIENER, J.L.; MOGUL, J.C.; SHAH, M.A.; VAHDAT, A. **Pip: Detecting the Unexpected in Distributed Systems.** In: Proceedings of the ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2006

ROS.; Site do projeto Robot Operating System. Disponível em: <http://www.ros.org>. Acesso em: janeiro de 2014.

ROCHA, P. G. C; STAA, A. V.; **Um mecanismo baseado em logs com meta-informações para a verificação de contratos em sistemas distribuídos.** Rio de Janeiro, 2014, 63p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

STAA AV. **Programação Modular.** Campus: Rio de Janeiro, RJ, 2000 (in Portuguese).

TANENBAUM A. S.; STEEN M. V. **Distributed Systems: Principles and Paradigms** (2nd Edition), Prentice-Hall, Inc., Upper Saddle River, NJ, 2006

TOROI, T.; JANTII, M.; MYKKANEN, J.; EEROLA, A.: **Testing component-based systems - the integrator viewpoint** In: IRIS 27. Information Systems Research Seminar in Scandinavia (Proceedings of the 27th IRIS), Falkenberg Sweden (2004).

WU, Y.; PAN, D.; CHEN, M.; **"Techniques for testing component-based software,"** in Proceeding of the Seventh International Conference on Engineering of Complex Computer System, vol. 00. Skovde, Sweden: IEEE, 2001, pp. 222-232.

NI, Z. W.; Yang, S. L.; Longshu L., JIA, R. W. "**Inttegrated case-based reasoning**", Proceedings of the 2nd International Conference on Machine Learning and Cybernetics, pp. 1845-1849, 2003.