Pontifícia Universidade Católica
do Rio de Janeiro

Pedro de Goes Carnaval Rocha

Um mecanismo baseado em logs com meta-informações para a verificação de contratos 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

Agosto de 2014

PONTIFÍCIA UNIVERSIDADE CATÓLICA
DO RIO DE JANEIRO

Pedro de Goes Carnaval Rocha

Um mecanismo baseado em logs com meta-informações para a verificação de contratos 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 Científico da PUC-Rio. Aprovada pela Comissão Examinadora abaixo assinada.

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

Prof. Alessandro Fabricio Garcia
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, 21 de agosto de 2014

Pedro de Goes Carnaval Rocha

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

Ficha Catalográfica

Rocha, Pedro

Um mecanismo baseado em logs com meta-informações para a verificação de contratos em sistemas distribuídos / Pedro de Goes Carnaval Rocha; orientador: Arndt von Staa. — Rio de Janeiro: PUC–Rio, Departamento de Informática, 2014.

v., 64 f.: il. ; 29,7 cm

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

Inclui referências bibliográficas.

1. Engenharia de software. 2. Sistemas distribuídos. 3. Qualidade de software. 4. Detecção de falha. I. Staa, Arndt von. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

À minha mãe, Alice

.

Agradecimentos

Ao meu orientador Prof Arndt von Staa, pelo apoio, incentivo e profissionalismo presentes durante o desenvolvimento da pesquisa.

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

Aos professores que participaram da Comissão examinadora.

A todos os professores e funcionários do Departamento pelos ensinamentos e pela ajuda.

À AevoTech e seus funcionários pelo apoio e por tornar possível a aplicação deste trabalho ao sistema de software do Projeto de um Robô para Monitoramento Ambiental (RoMA).

Ao meu grande amigo, Thiago Araújo, por me manter no rumo em momentos de tormenta, incentivar e motivar nos momentos mais difíceis.

À minha mãe e minha avó, Alice e Nadir, pelo amor, dedicação e confiança sem os quais não seria possível vencer mais este desafio.

Ao meu pai, irmão, irmã e avós, Alexandre, Vinicius, Daniela, Judith e Valdir, por terem acreditado na minha capacidade.

Aos amigos, Alexandre Duarte, Caio Dias, Carla Galdino, Daiane Andrade, Eliana Goldner, Fernando Cézar, Fernando Perrotti, Flavio Timbó, Gustavo Martins, Lohanna Cardoso, Lucas Gonçalves, Luiz Guilherme Coelho, Luiza Seabra, Marcela Fiad, Pedro Grojsgold, Rodrigo Freitas, Vitor Villela, Órion Montuano, Roberto Maia, Walther Maciel, Wellington Silva, pelo apoio, estímulo

e por estarem sempre por perto nas horas difíceis.

Resumo

Rocha, Pedro de Goes Carnaval; Staa, Arndt von. **Um mecanismo baseado em logs com meta-informações para a verificação de contratos em sistemas distribuídos.** Rio de Janeiro, 2014. 64p. Dissertação de Mestrado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Contratos de software podem ser escritos como expressões lógicas capazes de identificar falhas que ocorrem durante a utilização de um software. É possível implementar a verificação de um contrato em um software através de assertivas executáveis. No entanto, a forma como assertivas convencionais são implementadas não é diretamente aplicável a sistemas distribuídos, uma vez que apresentam dificuldades para avaliar expressões temporais, tampouco as expressões podem envolver propriedades de diferentes processos. Este trabalho propõe um mecanismo baseado em logs com meta-informações para a verificação de contratos em sistemas distribuídos. Uma gramática para redigir contratos possibilita operações temporais, ou seja, permite a especificação de condições entre eventos, em diferentes instantes de tempo, ou mesmo garante uma sequência de eventos, durante um período de tempo. O fluxo de eventos gerado é avaliado assincronamente em relação à utilização do sistema, pela comparação com contratos, previamente escritos de acordo com a gramática, que representam as expectativas sobre o comportamento normal do sistema.

Palavras-chave

Engenharia de software; Sistemas distribuídos; Qualidade de software; Detecção de falha.

Abstract

Rocha, Pedro de Goes Carnaval; Staa, Arndt von (Advisor). **A mechanism based on logs with meta-information for the verification of contracts in distributed systems.** Rio de Janeiro, 2014. 64p. MSc. Dissertation - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Software contracts can be written as assertions that identify failures observed while using the software. Software contracts can be implemented through executable assertions. However, conventional assertions are not directly applicable in distributed systems, as they present difficulties to evaluate temporal expressions, as well as expressions involving properties of different processes. This work proposes a mechanism based on logs with meta-information to evaluate contracts in distributed systems. A grammar to write contracts enable temporal operations, e.g., allows specifying conditions between events at different timestamps, or even guaranteeing a sequence of events over a period of time. The flow of events is evaluated asynchronously in relation to the system execution, by comparison with contracts, previously written according to the grammar, representing the expectations on the behavior of the system.

Keywords

Software engineering; Distributed systems; Software quality; Failure detection.

Sumário

Lista de figuras

Lista de tabelas

1
Introdução

A partir de evidências históricas podemos constatar que todo software produzido pode falhar (Brown & Patterson, 2001; Fox, 2002). A ocorrência de falhas, ainda que esporádica, pode resultar em insatisfações do usuário ou mesmo em danos substanciais, sejam estes financeiros ou acidentes envolvendo vidas humanas. Mecanismos para o tratamento de falhas têm sido desenvolvidos com o objetivo de identificá-las e recuperar o sistema para um estado correto. Acredita-se que para alcançar este objetivo, a aplicação deva ser construída sobre uma infraestrutura que apresente meios para analisar o seu comportamento, identificar anomalias e promover ações para tratar suas consequências (Murch, 2004).

O comportamento de um sistema pode ser especificado e avaliado através de contratos de software (Meyer, 1992), técnica que estabelece um conjunto de regras lógicas que especificam o seu correto funcionamento. Uma forma de detectar falhas em tempo de execução consiste em escrever contratos na forma de assertivas executáveis (Andrews, 1979) que, quando não satisfeitas, notificam o erro ocorrido ou mesmo cancelam a execução do sistema, seguindo uma abordagem de programação defensiva (Phillips, 2012). Essa abordagem produz bons resultados em sistemas locais, em que o estado pode ser acessado através dos meios tradicionais de programação. No entanto, contratos convencionais não são diretamente aplicáveis a sistemas distribuídos, uma vez que apresentam dificuldades para avaliar expressões temporais, tampouco as expressões podem envolver propriedades de diferentes processos, pois não é possível acessar um estado global instantâneo e consistente da aplicação. Nesses casos, a avaliação de um contrato tradicional obrigaria que todo o sistema suspendesse a operação até o término da verificação e, ainda assim, o estado do sistema poderia não ser consistente. Isso ocorre devido à característica distribuída, pois não é possível obter o estado de cada componente, com exatidão, para um determinado momento. No caso específico de sistemas distribuídos desenvolvidos em Java, a verificação de contratos orientados a aspectos utilizando a linguagem AWED (Navarro et al., 2006a, b,

2008) é possível devido à replicação e sincronização de *cache* implementadas por JBoss Cache (Ban & Wang, 2005).

Contratos de software necessitam de informações da execução do sistema para identificar a ocorrência de falhas. Técnicas de instrumentação de software tais como *trace* e *log*, podem ser utilizadas para extrair informações do fluxo de execução, com o objetivo de entender o comportamento do sistema.

A técnica de *trace* gera eventos com a sequência dos estados de um processo no decorrer da execução da aplicação. Essa técnica tende a produzir históricos de acompanhamento muito extensos, o que em geral é eficaz durante os testes, pois fornece informações sobre a execução do sistema. Entretanto, durante o uso, apresenta um impacto considerável de desempenho (Mirgorodskiy et al., 2006; Horwitz et al., 2010) e gera um volume de dados muito grande, sendo poucos destes de interesse para fins de avaliação, pois dados são gerados, sistematicamente, mesmo se nada tiverem a ver com as falhas observadas. Isso decorre do fato que, em tempo de uso, a instrumentação deve gerar dados a priori para potenciais falhas sempre desconhecidas. Evidentemente, se fossem conhecidas deveriam ser removidas ou encapsuladas, antes do sistema ser liberado para o uso.

Um *log* é formado por um conjunto de eventos registrados no decorrer da execução da aplicação. A técnica de *logging* é utilizada diretamente por desenvolvedores, provendo informações de alto nível de abstração, no formato de mensagem de texto livre. Uma mensagem clara e inteligível, possuindo suficiente informação pode contribuir para a identificação do que provocou a falha. Entretanto, mensagens de *log* costumam não seguir um padrão que facilite a análise mecânica, tampouco tendem a conter dados relativos ao estado de execução no momento do registro, tornando difícil a extração de propriedades que auxiliem a avaliação do comportamento do sistema. Esta limitação motivou a criação de técnicas de extração de propriedades, como o algoritmo apresentado por Vaarandi (2003), capaz de identificar padrões nas mensagens e, a partir deles, extrair a parte variável que, usualmente, expõe o valor de uma propriedade do contexto de execução. Por outro lado, as mensagens podem não conter informações suficientes, reduzindo a eficácia da verificação, uma vez que existem dificuldades para expor um contexto relevante da aplicação durante a notificação de um evento como, por exemplo, o esforço empregado pelo desenvolvedor na identificação das propriedades relevantes ou pelo próprio encapsulamento da modularização, dificultando o acesso às propriedades de contextos superiores (Araújo et al., 2014).

A técnica híbrida para geração de logs desenvolvida por Araújo et al. (2014), que consiste na emissão de eventos com informações de contexto em que foram gerados. Um evento é composto por uma mensagem, um identificador temporal e um conjunto variável de propriedades. Cada propriedade inserida em um evento é chamada de *tag*, sendo esta composta por um par nome-valor. Assim como o *log*, a técnica híbrida provê uma mensagem de texto livre, que auxilia o entender o comportamento do sistema. Contudo, as informações de contexto são indexadas através das *tags* inseridas nos eventos, o que torna desnecessária a utilização de *data mining* (Vaarandi, 2003). Esta solução é implementada através de uma pilha de *tags* que armazena as *tags* dos escopos superiores e as adiciona, automaticamente, em eventos criados nos escopos inferiores, resolvendo, assim, o problema de visibilidade das informações entre escopos. Desta forma, o *log* estruturado se assemelha ao *trace*, possivelmente provendo informações suficientes para a verificação de contratos.

Uma série de trabalhos tais como os de Hellerstein et al. (2002) e Hendrickson et al. (2003), utilizam técnicas de *tracing* e *logging* para detectar falhas em sistemas distribuídos. Entretanto, suas abordagens possuem um custo adicional devido ao processo de extração das propriedades que externalizam o estado do sistema.

Araújo et al. (2014), por sua vez, desenvolveram uma abordagem de instrumentação capaz de anotar eventos com meta-informações, representando o valor das propriedades de contexto relevantes, solucionando as dificuldades de coleta, organização e relacionamento de informações sobre a sua execução, através da forma como os desenvolvedores escrevem a notificação de eventos. Esta abordagem é baseada em um repositório central, para armazenar todos os eventos registrados e ordenados temporalmente, independentemente do processo e máquina em que originaram. Esses eventos possuem dados relativos a propriedades de contexto, notificados por cada entidade geradora de *log* (processo, *thread* e máquina). Uma ferramenta de inspeção, denominada Lynx, permite os desenvolvedores e mantenedores visualizar somente os eventos que forem do contexto de interesse. Esta técnica de extração resolve o problema de avaliação do estado em sistemas distribuídos, uma vez que possibilita um contrato de software comparar informações de eventos gerados por processos em máquinas diferentes, até mesmo quando estes são gerados em momentos distintos da execução.

A motivação para este trabalho de pesquisa é a complementação e o aprimoramento da abordagem de Araújo et al. (2014), pela adição de um

processo de identificação automática de falhas, previamente diagnosticadas, ocorridas em tempo de utilização do sistema, que reduza o trabalho do mantenedor do sistema e diminua o tempo para a verificação de contratos inválidos. No entanto, existe o seguinte problema: Como avaliar contratos em sistemas distribuídos utilizando *logs* estruturados?

Para a verificação de falhas em tempo de uso do sistema, ou seja, após ser instalado e estar disponível para os usuários finais, a abordagem de extração de informações deve externalizar as propriedades necessárias para verificar todos os contratos definidos, sem impactar significativamente o desempenho do sistema. Entretanto, a falta de eventos contendo informações relevantes constitui uma ameaça, pois impossibilita a identificação das falhas.

O objetivo deste trabalho de pesquisa consiste no desenvolvimento de um mecanismo baseado em logs com meta-informações para a verificação de contratos em sistemas distribuídos. É proposta uma gramática para redigir contratos que possibilita operações temporais, ou seja, permite a especificação de condições entre eventos, em diferentes instantes de tempo, ou mesmo garante uma sequência de eventos, durante um período de tempo. O fluxo de eventos gerado é avaliado assincronamente em relação à utilização do sistema, pela comparação com contratos, previamente escritos de acordo com a gramática, que representam as expectativas sobre o comportamento normal do sistema.

Na solução proposta, a ameaça de falta de eventos contendo informações relevantes é amenizada, uma vez que as *tags* de cada contrato são conhecidas e documentadas em um dicionário de *tags* do projeto. Esse problema ocorrerá apenas no caso de erro humano, ou seja, se o desenvolvedor não gerar a *tag* necessária para o contrato ou se a política de instrumentação não previr a sua geração. Cabe ressaltar que, apesar da política de instrumentação ser um fator que impacta a eficácia do mecanismo proposto, a verificação da sua adequação não faz parte do escopo deste trabalho. Por outro lado, o excesso de eventos contendo informações detalhadas, ou não relacionadas às falhas, pode gerar um volume excessivo de *log.* Questões relacionadas ao volume do *log* também não fazem parte do escopo deste trabalho. Além disso, a solução não pretende identificar a ocorrência de todas as falhas possíveis em um sistema. Contudo, o sistema pode ser melhorado mesmo que apenas algumas falhas sejam identificadas.

As perguntas de pesquisa que se deseja responder são:

(1) Como transcrever contratos para mecanismos capazes de avaliar o comportamento do sistema?
(2) Como tratar contratos que dependem de relacionamentos temporais entre eventos?
(3) Como comparar propriedades em eventos diferentes?
(4) Como tratar contratos que relacionem interações entre componentes diferentes, considerando concorrência, em threads, processos e máquinas diferentes?
(5) Como implementar uma solução de forma que o impacto no desempenho do sistema seja pequeno?

Para a avaliação da viabilidade e adequação da solução proposta, foi selecionado um conjunto de falhas de um sistema real na área da robótica. Esse sistema possui componentes para interação com *hardware*, específicos para controlar ou monitorar equipamentos dispostos na estrutura física de um robô, tais como: motores, sondas, sensores de proximidade, visão estereoscópica, entre outros A avaliação envolveu a discussão sobre a aplicação do mecanismo em cada uma das falhas selecionadas e a medição do impacto no desempenho do sistema.

Finalmente, este trabalho está organizado da seguinte forma: no primeiro capítulo é introduzido o problema; no segundo capítulo é apresentada a solução; no terceiro capítulo é descrita a arquitetura da solução, no quarto capítulo é discutida a avaliação, no quinto capítulo são apresentados os trabalhos relacionados e no sexto capítulo é concluído o trabalho.

2
Solução

A solução para a identificação, de forma automática, de falhas ocorridas em tempo de utilização de um sistema distribuído consiste em um mecanismo de verificação de contratos que avalia o comportamento do sistema, comparando-o com contratos escritos de acordo com uma gramática para redigir contratos. Essa gramática viabiliza a utilização das meta-informações presentes em eventos, localizados em uma base de dados, possibilitando que os contratos utilizem as *tags* dos eventos como variáveis, o que auxilia a comparação das condições de execução do sistema com as condições definidas, como esperadas, nos contratos. A gramática também possibilita operações temporais, ou seja, pode permitir a especificação de condições entre eventos, em diferentes instantes de tempo, ou mesmo garantir uma sequência de eventos, durante um período de tempo. Pressupõe-se que o mantenedor define, previamente, os contratos para avaliar o fluxo de eventos gerado. Cabe ressaltar que, a base de dados para o fluxo de eventos deve ser obtida pela técnica de extração de propriedades contextuais de Araújo et al (2014). Uma visão geral do mecanismo de verificação é ilustrada na Figura 1.

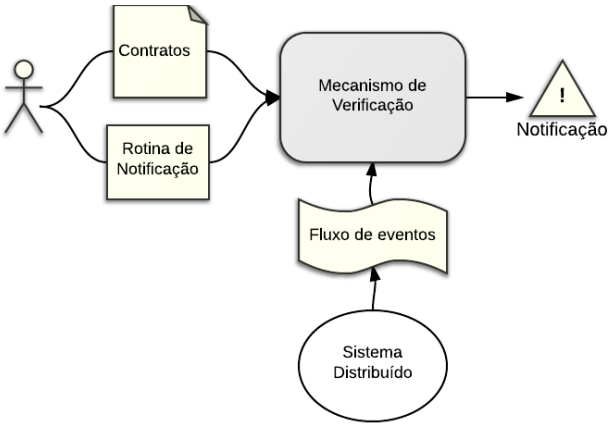

Figura 1 – Visão geral do mecanismo de verificação

Para um melhor entendimento da geração do fluxo de eventos a serem avaliados, é apresentada, na seção 2.1, a técnica de extração de propriedades contextuais de Araújo et al (2014). O mecanismo de verificação de contratos e a

gramática para redigir contratos são descritos nas seções 2.2 e 2.3, respectivamente.

2.1. A técnica de extração de propriedades contextuais

A técnica de extração de propriedades contextuais de Araújo et al (2014) é uma técnica híbrida de geração de *logs,* em que eventos são anotados com meta-informações do contexto em que foram gerados. Uma biblioteca de instrumentação possibilita que técnicas tradicionais de programação sejam utilizadas para inserir notificações de evento no código. As meta-informações são representadas por um conjunto de propriedades, chamadas *tags*, compostas por um par <nome, valor>, em que o valor é opcional. Uma *tag* pode ser tanto utilizada para representar uma variável, informando o seu nome e o seu valor, quanto para representar um estado como, por exemplo, um erro. Um exemplo de instrumentação é apresentado a seguir.

Exemplo de instrumentação em Python:

```
def is_user_legal_age(user_age):
    if user_age > 18:
        logger.notify('User is legal_age', {'age': user_age})
        return True
    logger.notify('User isn´t legal_age', {'age': user_age})
    return False

name = 'Pedro'
age = 25
logger.notify('Validate user', {'user': name, 'age': age})  # Notifica evento 1

logger.push_tag({'user': name})  # Empilha tag
logger.push_tag({'action': is_user_legal_age})  # Empilha tag
result = is_user_legal_age(age)    # Notifica evento 2
logger.pop_tag()  # Desempilha tag

logger.notify('Validation result', {'result': result})  # Notifica evento 3
logger.pop_tag()  Desempilha tag
```

Eventos notificados:

```
[user: Pedro][age: 25][message: Validate user]
[user: Pedro][action: is_user_legal_age][age: 25][message: User is legal_age]
[user: Pedro][result: True][message: Validation result]
```

O contexto do sistema é definido pelas propriedades em todos os escopos da execução de uma *thread*. A técnica de extração utiliza uma pilha de *tags* que possibilita ao desenvolvedor inserir *tags* relevantes em cada escopo e, a cada

notificação de evento, a biblioteca de instrumentação insere, automaticamente, todas as propriedades contidas na pilha no evento. Desta forma, em uma notificação de eventos, o desenvolvedor precisa apenas escrever as propriedades que são específicas a esse evento, e deixar que as propriedades do contexto sejam inseridas pela pilha de *tags*, o que evita a quebra de encapsulamento.

Todo componente gerador de *log* notifica seus eventos enviando-os para um repositório central, que armazena as informações, ordenadas temporalmente, em um banco de dados não estruturado. Uma ferramenta de inspeção, chamada Lynx, auxilia o diagnóstico de falhas através de um serviço de pesquisa de cenários, os quais são definidos por restrições, tais como um filtro temporal (datas inicial e final) ou filtros baseados em *tags* que devam ou não existir no evento de log, para que este seja selecionado. O mantenedor informa o contexto de interesse com base nas propriedades conhecidas da falha e a interface exibe os eventos possivelmente relacionados a ela. O resultado de uma pesquisa é uma sequencia de eventos que passam por esses filtros, ordenados temporalmente. Esta ferramenta de inspeção permite ao mantenedor aprender sobre o contexto das falhas, através do estudo dos eventos resultantes, e aprimorar os filtros, a fim de compreender a causa que levou às falhas. O mantenedor também pode selecionar *tags* para que estas não sejam exibidas, reduzindo, assim, o volume de informação visualizada.

O aprendizado da equipe de manutenção, com as falhas identificadas durante o tempo de execução do software contribui para a elaboração de contratos que possam detectar as falhas conhecidas e, consequentemente, para o sucesso da solução proposta neste capítulo.

2.2. Mecanismo de verificação

Inicialmente, antes do desenvolvimento do sistema, o desenvolvedor ou mantenedor deve estudar a sua arquitetura e, com base em conhecimento prévio, escrever, sob a forma de um ou mais contratos, o comportamento esperado para o sistema, assim como definir as respectivas rotinas de notificação. Cada um dos contratos deve ser escrito utilizando a gramática para redigir contratos. Assim, durante o desenvolvimento do sistema, o desenvolvedor saberá quais *tags* deverão ser usadas na etapa de instrumentação, que deve ser realizada de forma a auxiliar a verificação dos contratos em tempo de execução. Caso o *log* gerado não possua eventos suficientes para a identificação de falhas,

o mecanismo de verificação não será eficaz. Por outro lado, um excesso de eventos pode tornar o processo de verificação mais lento e, portanto, ineficiente.

Durante a execução do sistema, a rotina de verificação do mecanismo interpreta os contratos, previamente definidos, e avalia o fluxo de eventos extraídos de uma base de dados. Na medida em que as falhas são observadas, o mantenedor é notificado seguindo as rotinas definidas previamente.

Cabe ressaltar que, o conjunto de contratos deve ser atualizado sempre que uma falha for identificada manualmente. Assim, falhas similares poderão ser identificadas, automaticamente, pelo mecanismo de verificação, caso essas falhas ocorram novamente.

O mecanismo de verificação proposto é ilustrado na Figura 2.

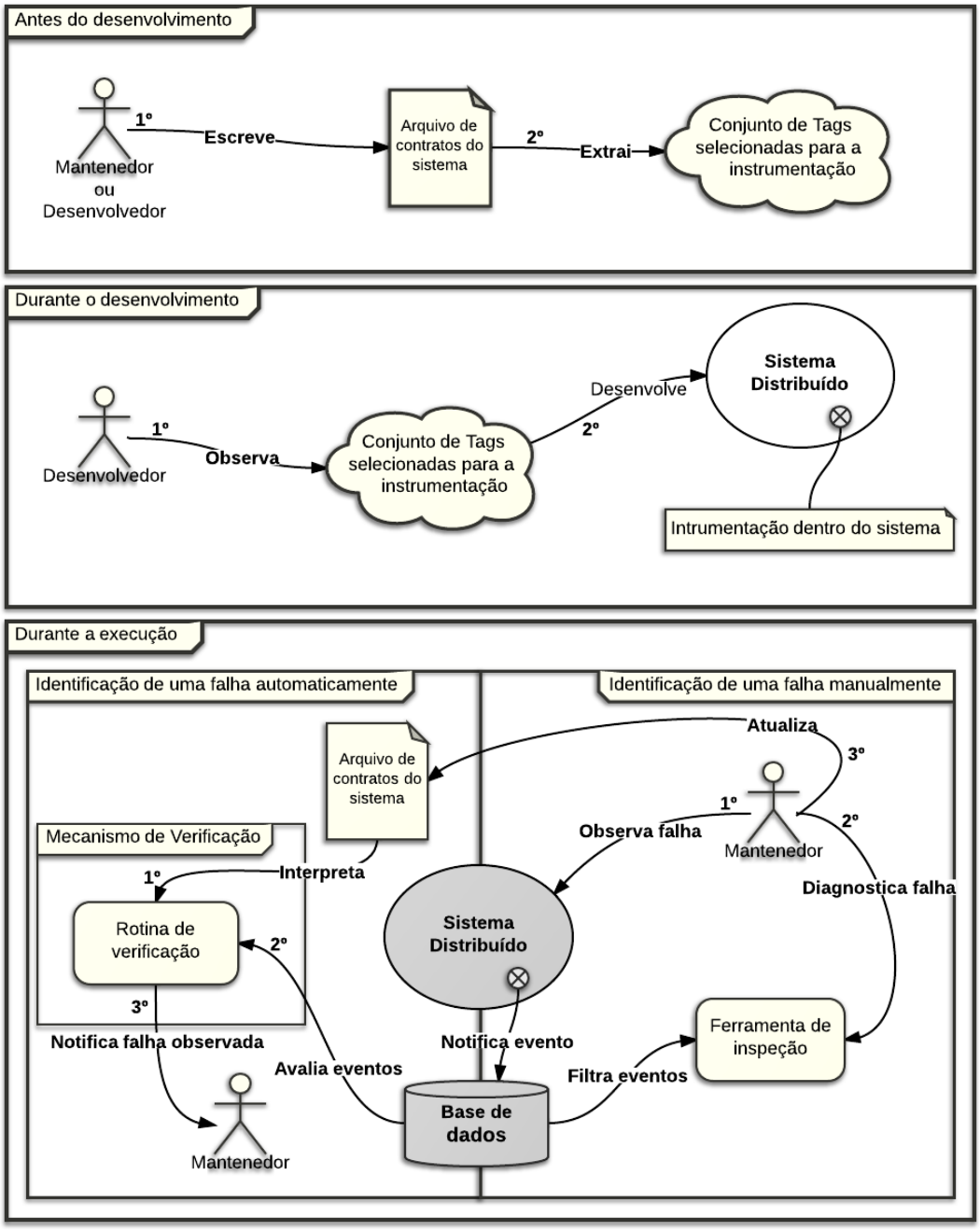

Figura 2– Mecanismo de verificação proposto

A arquitetura do mecanismo de verificação, ilustrada na Figura 3, é constituída pelos seguintes componentes:

- *Contrato Textual* – representa os contratos escritos pelo mantenedor ou desenvolvedor, sendo específico do sistema em avaliação.
- *Rotina de Notificação* – representa a rotina a ser chamada quando um contrato for invalidado, sendo específico do sistema em avaliação.
- *Motor de Busca* – componente da técnica de extração de propriedades contextuais de Araújo et al (2014) que extrai os dados de uma base de dados não relacional.

- *Expressão* – representa uma expressão que compõe um *Contrato,* em forma de *query*, na linguagem do *Motor de Busca*. Uma expressão pode ser do tipo *single event*, que especifica as condições envolvendo um único evento, ou *multi event*, que especifica as condições envolvendo mais de um evento, considerando o instante em que os eventos foram gerados.
- *Verificador* – executa a rotina de verificação do mecanismo, conforme descrita na seção 3.2.
- *Contrato* – composto por uma ou mais expressões, escritas de acordo com a gramática para redigir contratos, e representa uma instância do componente *Contrato Textual* para a verificação.
- *Contrato Temporário* – gerado apenas para a verificação de expressões *multi event*, descritas na seção 2.3.2. Consiste em uma cópia de um *Contrato*, em que é adicionada uma lista de eventos visitados e um prazo para expirar. Caso o prazo não seja atendido, o *Contrato Temporário* é invalidado.

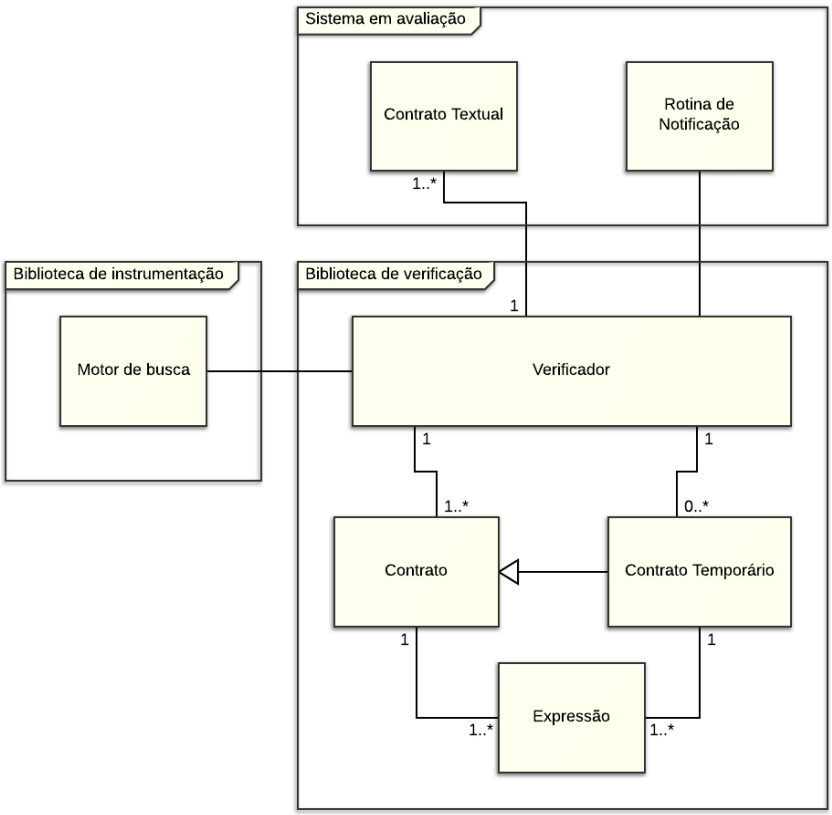

Figura 3 – Arquitetura dos componentes do mecanismo de verificação

2.3.
A gramática para redigir contratos

Durante o estágio inicial desta pesquisa, foi definida uma gramática com o objetivo de escrever contratos para mecanismos capazes de avaliar o comportamento de um sistema distribuído. O seu desenvolvimento foi iniciado durante a realização da disciplina “INF 2134 - Teste e Medição de Software”, cursada no segundo semestre de 2012. A primeira versão da gramática é ilustrada na Figura 4.

```
contract = event_exp |
           (event_exp, “in”, time) |
           (“if”, contract, “then”, (conditions | contract)) |
           (event_exp, “with”, “(“, event_exp, {“,”, event_exp}, “)”, “in”, time);

event_exp = event |
            not_event, [(“and” | “or”), event_exp];

not_event = “not”, “(“, event, “)”;

event = [“not”], tag, [[“or”], event];

tag = “[“, {character}, [“as” {character}], [“:”, {character}], “]”;

time = {digits};

condition = [“!”], ({character} | “(“, condition, “)”),
            [(“==” | “!=” | “<“ | “>” | “<=” | “>=” | “&&” | “||”), condition];
```

Figura 4 – Primeira versão da gramática para redigir contratos

Após a incorporação de uma série de melhorias, como a mudança do formato EBNF (Scowen, 1993) para uma gramática livre-do-contexto (Chomsky, 1956) com a notação JSON (ECMA, 2013), que facilita a serialização dos contratos, a inclusão de um operador que permite a verificação de uma sequencia de expressões e a adoção do padrão LALR, que possibilita a geração de analisadores sintáticos, obteve-se a gramática para redigir contratos, ilustrada na Figura 5. Cabe ressaltar que a conformidade com padrão LALR foi confirmada pelo analisador de gramáticas Grammophone (2003).

```
# Initial symbol
CONTRACT -> ANY_EXPRESSION.


# Auxiliary
ANY_EXPRESSION -> SINGLE_EVENT_EXPRESSION | MULTI_EVENT_EXPRESSION.

EXPRESSION_LIST -> SINGLE_EVENT_EXPRESSION , SINGLE_EVENT_EXPRESSION
                 | SINGLE_EVENT_EXPRESSION , EXPRESSION_LIST.

# Simple Tag
TAG -> { $exists : TAG_NAME } # Tag exists
     | { $not_exists : TAG_NAME } # Tag not exists
     | { TAG_NAME : TAG_VALUE } # Tag with value
     | { TAG_NAME : PARENT_TAG_NAME }. # Tag with parent tag value

TAG_NAME -> string.
TAG_VALUE -> string.

PARENT_TAG_NAME -> $parent.TAG_NAME | $parent.PARENT_TAG_NAME.   # $parent access the parent tag event


# Single event expression
SINGLE_EVENT_EXPRESSION -> TAG  # Statement
   | { $not: ANY_EXPRESSION }  # Not
   | { $and: [ EXPRESSION_LIST ] }  # And
   | { $or: [ EXPRESSION_LIST ] }  # Or
   | { $if: SINGLE_EVENT_EXPRESSION , $then: SINGLE_EVENT_EXPRESSION }  # Conditional
   | { $if: SINGLE_EVENT_EXPRESSION , $then: SINGLE_EVENT_EXPRESSION , $else: SINGLE_EVENT_EXPRESSION
}
   | { $compare: ( TAG_NAME , COMPARE_TYPE , COMPARE_WITH ) }.  # Comparison

COMPARE_WITH -> TAG_NAME | TAG_VALUE | PARENT_TAG_NAME.
COMPARE_TYPE -> == | != | < | > | <= | >=.


# Multi event expression
MULTI_EVENT_EXPRESSION ->
     { $occurred: SINGLE_EVENT_EXPRESSION , $other: ANY_EXPRESSION , TIME_WINDOW }
   | { $sequence: [ EXPRESSION_LIST ] , ORDER , TIME_WINDOW }
   | { $sequence: [ EXPRESSION_LIST ] , TIME_WINDOW }. # Default $exclusive_order: False

ORDER -> $exclusive_order: True  # The exact sequence order will be maintained
       | $exclusive_order: False.  # The sequence will be maintained, but not the exact order

TIME_WINDOW -> $within: time
```

Figura 5 – Gramática para redigir contratos

O símbolo inicial da gramática, a partir do qual todos os contratos são derivados, é denominado *CONTRACT*. Cada contrato pode ser definido por uma ou mais expressões do tipo evento único, denominada *single event*, ou do tipo eventos múltiplos, denominada *multi event*., que especificam, por meio de operadores lógicos, condicionais, existenciais e de comparação, as condições que devem ser atendidas por cada um dos eventos da base de dados do sistema.

Para auxiliar na construção da gramática, foram criados os símbolos a seguir:

- *SINGLE_EVENT_EXPRESSION* – deriva uma expressão *single event*;
- *MULTI_EVENT_EXPRESSION* - deriva uma expressão *multi event;*
- *ANY_EXPRESSION* - deriva qualquer tipo de expressão, isto é, do tipo *single event* ou *multi event; e*
- *EXPRESSION_LIST* - possibilita a criação de uma lista de expressões do tipo *single event*.

Os contratos devem ser escritos utilizando operadores lógicos, existenciais, condicionais e de comparação. As propriedades de um operador são representadas pelo caractere $, *token* da linguagem da gramática, seguido pelo nome da propriedade. Por exemplo, o operador lógico *Se-Então* da expressão *single event* é composto pelas propriedades *$if, $then e $else*, onde *$else* é opcional. Este operador e suas propriedades são apresentados a seguir.

```
SINGLE_EVENT_EXPRESSION ->
   { $if: SINGLE_EVENT_EXPRESSION , $then: SINGLE_EVENT_EXPRESSION }
 | { $if: SINGLE_EVENT_EXPRESSION , $then: SINGLE_EVENT_EXPRESSION , $else: SINGLE_EVENT_EXPRESSION }
```

Os eventos notificados pela técnica de extração de propriedades contextuais de Araújo et al (2014) são representados por um conjunto de propriedades, chamadas *tags*, compostas por um par <nome, valor>, em que o "valor" é opcional. O símbolo *TAG* define as suas possíveis condições, que são: a *tag* existe, não existe ou possui um valor associado. Os símbolos TAG_NAME e TAG_VALUE representam, respectivamente, o nome e o valor da *tag*, e são escritos pelo mantenedor em texto livre, ou seja, um *string*. A seguir são apresentadas as regras do símbolo TAG.

```
Regras:
       TAG -> { $exists : TAG_NAME } # Tag exists
            | { $not_exists : TAG_NAME } # Tag not exists
            | { TAG_NAME : TAG_VALUE }. # Tag with value
       TAG_NAME -> string.
       TAG_VALUE -> string.
```

Os operadores específicos das expressões *single event* e *multi event* são descritos nas seções 2.3.1 e 2.3.2, respectivamente. Os exemplos apresentados nestas seções têm como base os eventos listados a seguir:

	Eventos:	Instante de emissão:
1:	[a: 5] [b: 5] [c]	0ms
2:	[a: 3] [b] [c: 8]	1ms
3:	[a] [b: 3] [c: 9]	2ms
4:	[a: 1] [b: 2] [c: 3]	4ms
5:	[a: 2] [c: 5]	5ms
6:	[a: 3] [b: 2]	6ms
7:	[a: 4] [error]	8ms
8:	[b: 2] [c: 4] [d: 3]	9ms
9:	[b: 2]	10ms
10:	[c: 6]	11ms

2.3.1.
Expressão *single event*

Uma expressão *single event* especifica condições envolvendo um único evento, por meio de operadores existencial, lógicos e de comparação, conforme descrito nos subitens a seguir.

2.3.1.1.
Operador existencial

O operador existencial verifica as possíveis condições de uma *tag*: (i) a *tag* existe, usando a propriedade $exists; (ii) a *tag* não existe, usando a propriedade $not_exists; e (iii) a tag possui um valor associado. Uma expressão contendo esse operador é válida se a sua condição for atendida. A seguir, são apresentadas as regras e exemplos de contratos utilizando esse operador.

```
Regras:
      SINGLE_EVENT_EXPRESSION -> TAG.  # Statement
      TAG -> { $exists : TAG_NAME } # Tag exists
           | { $not_exists : TAG_NAME } # Tag not exists
           | { TAG_NAME : TAG_VALUE }. # Tag with value
      TAG_NAME -> string.
      TAG_VALUE -> string.

Exemplos de contratos e os respectivos eventos inválidos:

      {$exists: a} identifica os eventos [8, 9, 10] como inválidos.

      {$not_exists: error} identifica o evento [7] como inválido.

      {b: 2} identifica os eventos [1, 2, 3, 5, 7, 10] como inválidos.
```

2.3.1.2.
Operadores lógicos

Os operadores lógicos usados pela gramática têm como base os operados lógicos de programação, que são: de *Negação*, *E*, *Ou* e, *Se-Então*.

O operador de Negação inverte o resultado da verificação na expressão informada na propriedade *$not*, isto é, a expressão é válida se a subexpressão for inválida, e inválida se a subexpressão for válida. A seguir, são apresentadas as regras e exemplos de contratos utilizando esse operador.

```
Regra:
      SINGLE_EVENT_EXPRESSION -> { $not: SINGLE_EVENT_EXPRESSION }.  # Not
```

Exemplos de contratos e os respectivos eventos inválidos:

{$not: {$exists: d}} identifica o evento **[8]** como inválido.

{$not: {b: 2}} identifica os eventos **[4, 6, 8, 9]** como inválidos.

Uma expressão com o operador *E* é válida se, e somente se, todas as expressões da lista, informada na propriedade *$and*, forem válidas. A seguir, são apresentadas as regras e exemplos de contratos utilizando esse operador.

```
Regra:
      SINGLE_EVENT_EXPRESSION -> { $and: [ EXPRESSION_LIST ] }.  # And
      EXPRESSION_LIST -> SINGLE_EVENT_EXPRESSION , SINGLE_EVENT_EXPRESSION
                       | SINGLE_EVENT_EXPRESSION , EXPRESSION_LIST.
```

Exemplos de contratos e os respectivos resultados:

{$and: [{$exists: b}, {$exists: c}]} identifica os eventos **[5, 6, 7, 9, 10]** como inválidos.

{$and: [{a: 3}, {b: 2}]} identifica, só, o evento **[6]** como válido.

Uma expressão com o operador *Ou* é válida se, pelo menos uma das expressões da lista, informada na propriedade *$or*, for válida. A seguir, são apresentadas as regras e exemplos de contratos utilizando esse operador.

```
Regra:
      SINGLE_EVENT_EXPRESSION -> { $or: [ EXPRESSION_LIST ] }.  # Or
      EXPRESSION_LIST -> SINGLE_EVENT_EXPRESSION , SINGLE_EVENT_EXPRESSION
                       | SINGLE_EVENT_EXPRESSION , EXPRESSION_LIST.
```

Exemplos de contratos e os respectivos eventos inválidos:

{$or: [{$exists: b}, {$exists: c}]} identifica o evento **[7]** como inválido.

{$or: [{a: 3}, {b: 2}]} identifica os eventos **[1, 3, 5, 7, 10]** como inválidos.

O operador *Se-Então* é composto por expressões obrigatórias, informadas nas propriedades *$if* e *$then*, e uma expressão opcional, informada na propriedade *$else*. Uma expressão com esse operador segue as regras de implicação, isto é, a expressão é considerada inválida se, e somente se, a expressão *$if* for válida e *$then* for inválida, ou, se existir *$else*, as expressões *$if* e *$else* forem inválidas. A seguir, são apresentadas as regras e exemplos de contratos utilizando esse operador.

```
Regras:
      SINGLE_EVENT_EXPRESSION ->
         { $if: SINGLE_EVENT_EXPRESSION , $then: SINGLE_EVENT_EXPRESSION }
       | { $if: SINGLE_EVENT_EXPRESSION , $then: SINGLE_EVENT_EXPRESSION , $else: EXPRESSION }.
```

Exemplos de contratos e os respectivos eventos inválidos:

{$if: {$exists: b}, $then: {$exists: c}} identifica os eventos **[6, 9]** como inválidos.

{$if: {a: 3}, $then: {b: 2}, $else: {$exists: c}} identifica os eventos **[2, 7, 9]** como inválidos.

2.3.1.3. Operador de comparação

O operador de *Comparação* valida a comparação entre uma *tag* e um valor. A propriedade *$compare* informa os dados para comparação, nessa ordem: (i) o nome da *tag*; (ii) o tipo da comparação; e (iii) o valor que será comparado. Esse valor pode ser um número, um texto ou uma outra *tag* do evento. As comparações são dos tipos: (i) igual; (ii) diferente; (iii) menor que; (iv) maior que; (v) menor ou igual; e (vi) maior ou igual. Com base no resultado da comparação, uma expressão com esse operador é considerada válida ou inválida. Caso alguma *tag* não exista, ou não tenha valor, a expressão com esse operador é considerada inválida. A seguir são apresentadas as regras e exemplos de contratos utilizando esse operador.

```
Regras:
      SINGLE_EVENT_EXPRESSION ->
            { $compare: ( TAG_NAME , COMPARE_TYPE , COMPARE_WITH ) }.  # Comparison
      COMPARE_TYPE -> == | != | < | > | <= | >=.
      COMPARE_WITH -> TAG_NAME | TAG_VALUE.
```

Exemplos de contratos e os respectivo eventos inválidos:

{$compare: (a, >=, b)} identifica os eventos **[2, 3, 4, 5, 7, 8, 9, 10]** como inválidos.

{$if: {$and: [{$exists: a}, {$exists: b}]}, $then: {$compare: (a, >=, b)}} identifica os envetos **[2, 3, 4]** como inválidos.

{$or: [{$compare: (a, >=, b)}, {$compare: (a, <, c)}]} identifica os eventos **[3, 7, 8, 9, 10]** como inválidos.

2.3.2. Expressão *multi event*

Uma expressão *multi event* especifica condições envolvendo mais de um evento, considerando o instante em que os eventos foram gerados. Sua principal

característica é a definição da janela de tempo em que esses eventos são avaliados, sendo esta informada na propriedade *$within* do operador. A seguir são apresentadas as regras e exemplos de contratos utilizando essa propriedade.

Regras:

MULTI_EVENT_EXPRESSION -> { ... , TIME_WINDOW }

TIME_WINDOW -> $within: time

Exemplos:

{..., $within: 2ms} verifica em uma janela de tempo de 2ms.

{..., $within: 1m20s10ms} verifica em uma janela de tempo de 1m:20s:2ms.

Os operadores condicionais de Ocorrência e Sequência e uma propriedade para acessar *tags* de eventos visitados, definidos para esse tipo de expressão, são descritos nos subitens a seguir.

2.3.2.1. Operadores condicionais

O operador de Ocorrência é composto por duas subexpressões informadas nas propriedades *$occurred* e *$other*. Uma expressão com esse operador é considerada inválida se a expressão *$occurred* for válida e a expressão *$other* não for válida, dentro da janela de tempo. Ou seja, se a expressão *$occurred* for válida, a expressão *$other* deverá ser atendida dentro da janela de tempo. A seguir, são apresentadas as regras e exemplos de contratos utilizando esse operador.

Regras:

MULTI_EVENT_EXPRESSION ->

{ $occurre: SINGLE_EVENT_EXPRESSION , $other MULTI_EVENT_EXPRESSION , TIME_WINDOW }

TIME_WINDOW -> $within: time

Exemplos:

{$occurred: {a: 3}, $other: {b: 2}, $within: 4ms} é válido.

{$occurred: {a: 3}, $other: {b: 3}, $within: 3ms} é inválido no instante **9ms**. A expressão observa a ocorrência da expressão **{a: 3} aos 6m,** mas expira **3ms** depois.

O operador de *Sequência* relaciona uma lista de expressões informada na propriedade *$sequence*. Uma expressão com esse operador, inicialmente, busca a ocorrência da primeira expressão da sequência e, em seguida, verifica se as demais expressões ocorrerem, em ordem, dentro da janela de tempo. Isto é, uma expressão com esse operador é inválida se a sequência não for atendida dentro da janela de tempo. Além disso, a propriedade opcional *$exclusive_order*

define se a verificação de uma ordem diferente invalida a expressão. A seguir, são apresentadas as regras e exemplos de contratos utilizando esse operador.

```
Regras:
   MULTI_EVENT_EXPRESSION ->
        { $sequence: [ EXPRESSION_LIST ] , ORDER , TIME_WINDOW },
      | { $sequence: [ EXPRESSION_LIST ] , TIME_WINDOW }.
   ORDER -> $exclusive_order: True  # The exact sequence order will be maintained
          | $exclusive_order: False.  # The sequence will be maintained
   TIME_WINDOW -> $within: time
```

Exemplos:

{$sequence: [{a: 3}, {c: 5}, {b: 2}], $within: 7ms} é inválido no instante **13ms**. A expressão observa a ocorrência da expressão **{a: 3} em 6ms**, mas expira **7ms** depois, sem encontrar as outras expressão da sequência na ordem.

{$sequence: [{a: 3}, {b: 2}, {c: 6}], $within: 7ms, $exclusive_order: False} é inválido no instante **8ms**. A expressão observa a ocorrência das expressão **{a: 3} em 1ms** e **{b: 2} em 4ms**, mas expira **6ms** depois, sem satisfazer a expressão **{c: 6}**.

{$sequence: [{a: 3}, {c: 4}, {b: 2}], $within: 4ms, $exclusive_order: True} é inválido nos instantes **4ms** e **10ms**. Na primeira, a expressão observa a ocorrência da expressão **{a: 3} em 1ms**, mas observa a ocorrência fora de ordem da expressão **{b: 2} em 4ms.** Na segunda, a expressão observa a ocorrência da expressão **{a: 3} em 6ms** e **{c: 4} em 9ms**, mas expira no instante seguinte.

2.3.2.2. Acessando *tags* de eventos visitados

A propriedade $*parent*, existente no símbolo *PARENT_TAG_NAME*, possibilita o acesso ao valor de *tag* de um evento visitado por uma expressão *multi event*. Os eventos possuem uma relação de parentesco, onde cada evento visitado é considerado pai do evento visitado em seguida. Ou seja, se for desejado acessar um valor de *tag* de um evento visitado, basta concatenar "*$parent*." com o nome da *tag*.

As regras modificadas do símbolo TAG, para a aplicação da propriedade *$parent*, são apresentadas a seguir.

```
Regras:
      TAG -> { $exists : TAG_NAME } # Tag exists
           | { $not_exists : TAG_NAME } # Tag not exists
           | { TAG_NAME : TAG_VALUE } # Tag with value
           | { TAG_NAME : PARENT_TAG_NAME }. # Tag with parent tag value
      TAG_NAME -> string.
      TAG_VALUE -> string.
      PARENT_TAG_NAME -> $parent.TAG_NAME | $parent.PARENT_TAG_NAME.
```

As regras modificadas no operador de *Comparação*, para para a aplicação da,propriedade *$parent*, são apresentadas a seguir.

```
Regras:
      SINGLE_EVENT_EXPRESSION ->
            { $compare: ( TAG_NAME , COMPARE_TYPE , COMPARE_WITH ) }.  # Comparison
      COMPARE_TYPE -> == | != | < | > | <= | >=.
      COMPARE_WITH -> TAG_NAME | TAG_VALUE | PARENT_TAG_NAME.
```

A propriedade *$parent* só pode ser utilizadas em subexpressões dos operadores de *Ocorrência* e de *Sequência,* ou seja, só pode ser usada quando existir um evento pai (último evento visitado). A seguir, são apresentadas uma explicação mais detalhada e exemplos de uso dessa propriedade.

Explicação:

Se **Ev(X)** representa o evento que satisfaz a expressão **X**, então:

(i). Em uma expressão {$occurred: A, $other: B, $within: 1m}, **Ev(A)** é o evento pai durante da expressão **B**.

(ii). Em uma expressão {$sequence: [A, B, C], $within: 1m}, **Ev(A)** é o evento pai durante da expressão **B**, e o **Ev(B)** é o evento pai da expressão **C**.

Exemplos:

{$occurred: {c: 8}, $other: {d: $parent.a}, $within: 10ms} é válido. A expressão observa a ocorrência das expressão **{c: 8} em 1ms** e **{d: 3} em 9ms**.

{$sequence: [{a: 1}, {a: $parent.c}, {b: $parent.$parent.b}], $within: 10ms} é válido. A expressão observa a ocorrência das expressão **{a: 1} em 4ms**, **{a: 3} em 6ms** e **{b: 2} em 9ms**.

{$occurred: {c: 8}, $other: {$compare: (b, >, $parent.a)}, $within: 10ms}** é inválido no instante **11ms**. A expressão observa a ocorrência da expressão **{a: 3} em 1ms**, mas expira **10ms** depois, sem satisfazer a expressão **{$compare: (b, >, 3)}.

2.4. Limitações e ameaças

A principal limitação conhecida para a solução proposta está no fato do mecanismo apenas poder ser utilizado para eventos anotados com meta-informações e, portanto, haver a necessidade de se obter a base de dados para o fluxo de eventos, por meio da técnica de extração de propriedades contextuais de Araújo et al (2014). Consequentemente, as limitações provenientes dessa técnica são herdadas diretamente para o mecanismo de verificação. São elas:

- o aprendizado e utilização da biblioteca de instrumentação;
- o esforço de implementação da instrumentação;
- o pequeno erro de relógio, *timestamp,* devido ao tempo de transmissão dos eventos; e

- a necessidade de um banco de dados disponível para todos os componentes emissores de log.

A falta de eventos contendo informações relevantes constitui uma ameaça, pois impossibilita a identificação das falhas e torna o mecanismo de verificação ineficaz. Essa ameaça pode ser amenizada se as *tags* de cada contrato são conhecidas e documentadas em um dicionário de *tags* do projeto. Esse problema ocorrerá apenas no caso de erro humano, ou seja, se o desenvolvedor não gerar a *tag* necessária para o contrato ou se a política de instrumentação não previr a sua geração. Cabe ressaltar que, apesar da política de instrumentação ser um fator que impacta a eficácia do mecanismo proposto, a verificação da sua adequação não faz parte do escopo deste trabalho. Por outro lado, o excesso de eventos contendo informações detalhadas, ou não relacionadas às falhas, pode gerar um volume excessivo de *log* e tornar o processo de verificação mais lento e, portanto, ineficiente. Questões relacionadas ao volume do *log* também não fazem parte do escopo deste trabalho.

A falta de precisão do *timestamp* é uma ameaça para a ordenação dos eventos, caso esses tenham sido gerados em um intervalo de tempo muito pequeno. Isso pode inabilitar um contrato que verifique as relações entre mais de um evento.

A principal limitação da gramática é a necessidade de uma *tag timestamp* em todos os eventos do *log*. As demais limitações da gramática estão listadas a seguir:

- apesar do operador existencial possuir propriedades para verificar se uma *tag* existe, não existe ou possui um valor associado, não foram implementadas propriedades para verificar o valor de uma *tag*, por exemplo, a *tag* possui um dado valor ou não possui valor (*$has_value e $no_value, respectivamente)*.
- não foram implementadas propriedades para utilizar expressões regulares para verificar padrões de caracteres nos valores de uma *tag*, como, por exemplo, *$regexp.*
- o operador de comparação não permite a comparação entre dois valores textuais, ou de um valor textual com valor numérico;
- o caractere “$”, por ser específico da gramática, não permite redigir contratos que verifiquem *tags* ou valores que iniciem por esse caractere;

- apesar do operador lógico XOR não ter sido implementado, é possível escrever “{$xor: [A, B]}” como “{$not: {$and: [{$if: A, $then: B}, {$if: B, $then: A}]}}”;
- não foram implementadas propriedades para verificar a não ocorrência de um evento dentro de uma janela de tempo. Por exemplo: ocorrer um evento A, não pode ocorrer o evento B dentro de uma janela de tempo T.

As sugestões para o operador existencial são apresentadas a seguir:

```
Regras:
      TAG -> { $exists : TAG_NAME } # Tag exists
           | { $not_exists : TAG_NAME } # Tag not exists
           | { TAG_NAME : TAG_VALUE } # Tag specific value
           | { TAG_NAME : $no_value } # Tag has no value
           | { TAG_NAME : $has_value } # Tag with value
           | { TAG_NAME : REGEXP }. # Tag with pattern of value to be compared
      TAG_NAME -> string.
      TAG_VALUE -> string.
      REGEXP -> { $regexp: string }.
```

3
Arquitetura da solução

Com o objetivo de implementar a solução proposta, foi desenvolvida uma ferramenta que avalia o comportamento de um sistema distribuído em utilização, comparando-o com contratos escritos de acordo com a gramática para redigir contratos. O diagrama de caso de uso, a rotina de verificação e o diagrama de classe do mecanismo de verificação e detalhes técnicos da implementação da ferramenta são descritos nas seções 3.1, 3.2, 3.3 e 3.4, respectivamente.

3.1.
Diagrama de caso de uso do mecanismo de verificação

O diagrama de caso de uso, ilustrado na Figuras 6, descreve as funcionalidades propostas para o mecanismo de verificação de contratos. As operações básicas são descritas a seguir:

- *Adicionar arquivos com contratos* – o operador informa ao mecanismo, o endereço do arquivo que contém os contratos. Essa operação requer que os contratos, armazenados no arquivo informado, estejam escritos de acordo com a gramática para redigir contratos;
- *Definir a Rotina de Notificação* – o operador informa ao mecanismo, o nome da rotina desejada. Essa operação requer que as possíveis rotinas estejam codificadas em um arquivo. O escopo dessa rotina deve receber os seguintes argumentos: (i) o contrato invalidado; (ii) a lista de eventos inválidos; e (iii) o momento da notificação;
- *Iniciar a verificação* – o operador informa ao mecanismo, o instante para o início da verificação. Em seguida, é iniciada a rotina de verificação, conforme descrito na seção 3.2. Os contratos invalidados durante a verificação são notificados pela *Rotina de Notificação e* armazenados em uma lista, a qual é visualizada na operação *Listar contratos invalidados; e*
- *Listar contratos invalidados* – o operador solicita ao mecanismo, que liste os contratos que foram invalidados.

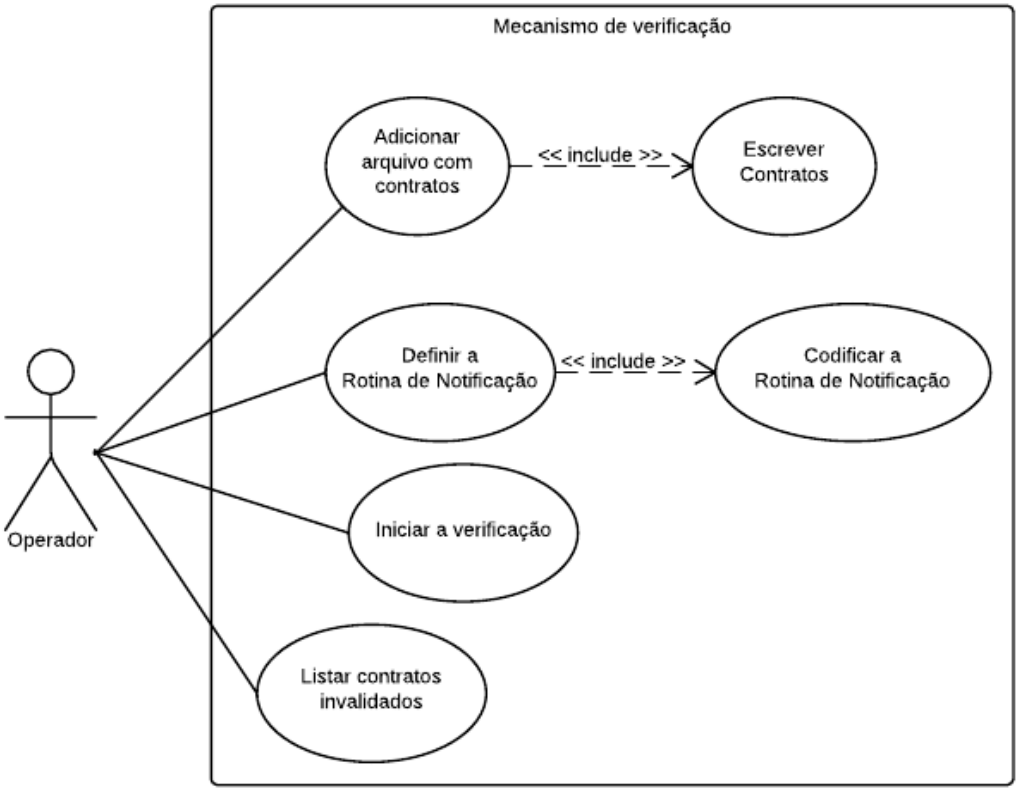

Figura 6 – Digrama de caso de uso do mecanismo

3.2. Rotina de verificação

A rotina de verificação é realizada pelo componente *Verificador.* Essa rotina executa as seguintes etapas:

(i) conexão com o *Motor de Busca*;

(ii) interpretação do arquivo de contratos;

(iii) configuração da *Rotina de Notificação*; e

(iv) avaliação dos eventos.

A etapa de conexão com o *Motor de Busca* é inicializada através da criação de um descritor para acessar o banco de dados. A etapa de interpretação converte cada um dos contratos, contidos no arquivo, em uma lista de expressões na forma de *query*. Estas expressões são criadas com a finalidade de acelerar a verificação do contrato, uma vez que a *query* é escrita de acordo com a linguagem utilizada pelo *Motor de Busca.* Cabe ressaltar que, cada *Contrato Textual* é representado por uma instância de *Contrato*, composto pelas expressões criadas. Esta instância é adicionada à uma lista de contratos. A seguir, são apresentados exemplos de conversão de *Contrato Textual* para uma *query* de MongoDB (2014):

```
Expressão não contém a tag error:
  Contrato Textual: { $not_exists: error }
  MongoDB: { error: { $exists: false } }

Se contém a tag A, então deve conter a tag B:
  Contrato Textual: { $if: { $exists: A }, $then: { $exists: B } }
  MongoDB: { $or: [ { $nor: [ { b: { $exists: true } } ] }, { c : { $exists: true } } ] }
```

A etapa da configuração da Rotina de Notificação consiste na importação da rotina informada na operação *Definir a Rotina de Notificação*. A seguir é apresentado um exemplo, em Python, de codificação da rotina, sua importação e sua inicialização.

```
# Notification Routine example
from pprint import pprint
def notification_routine(contract, events, timestamp):
    print “Timestamp: {0}”.format(timestamp)
    print “Invalid contract: {0}”.format(contract)
    print “Invalid events:”
    pprint(events)


# Importing defined Notification Routine
exec "from notification_routine import {0} as notification_function"
     .format(defined_notification_routine)

# Starting Notification Routine
notification_function(invalid_contract, invalid_events, current_timestamp)
```

A etapa de avaliação dos eventos inicia no instante definido na operação *Iniciar a verificação.* Inicialmente, é delimitado o trecho de eventos que será avaliado. Esse trecho é atualizado sempre que terminada a sua avaliação com todos os contratos nas listas de contratos e contratos temporários. Cabe ressaltar que, a base de dados pode apresentar um volume considerável de eventos a ser processado e, portanto, na avaliação deve-se, primeiro, verificar todos os contratos no trecho de eventos, antes de atualizá-lo. Desta forma, é possível determinar quais eventos representam um estado correto para um determinado instante de tempo, uma vez que estes tenham sido verificados por contratos que possuem apenas expressões *single event*. Entretanto, no caso dos eventos que tenham sido verificados por contratos que possuem expressões *multi event*, é necessário que estes contratos tenham terminado a sua verificação, na janela de tempo.

Os diagramas de fluxo para a avaliação das listas de contratos e contratos temporários são apresentados nas Figuras 7 e 8, respectivamente.

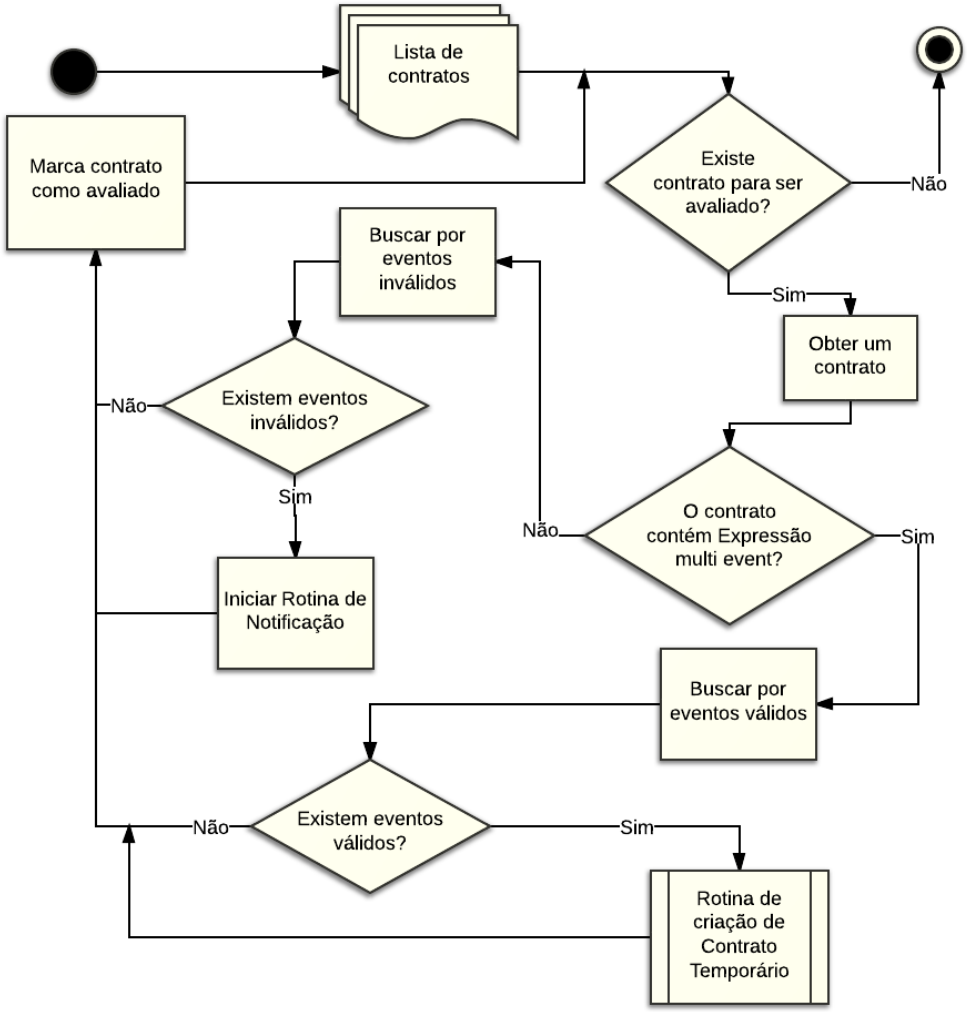

Figura 7 – Diagrama de fluxo para a avaliação da lista de contratos

A rotina de avaliação da lista de contratos verifica se os eventos do instante corrente invalidam o Contrato. Essa rotina possui como entrada, o conjunto de eventos do instante a ser verificado e, como saída, a notificação em caso de identificação de uma falha. O pseudo-código da rotina de avaliação da lista de contratos é apresentado a seguir.

1 Obter um contrato.

2 Se o contrato contém uma expressão *multi event*:

2.1 Como contratos compostos por expressões *multi event* dependem de mais de um evento durante sua verificação, nesta etapa, deve-se **buscar por eventos válidos** para a primeira expressão escrita no contrato.

2.2 Executar a Rotina de criação de Contrato Temporário.

3 Caso contrário:

3.1 Como contratos compostos por expressões single event dependem, apenas, de um evento durante sua verificação, nesta etapa, deve-se **busca por eventos que inválidos** para a expressão escrita no contrato.

3.2 Se algum evento retornar dessa busca, então, iniciar a Rotina de Notificação configurada.

3.3 Marca o contrato como avaliado.

4 Continuar a verificação dos contratos seguintes.

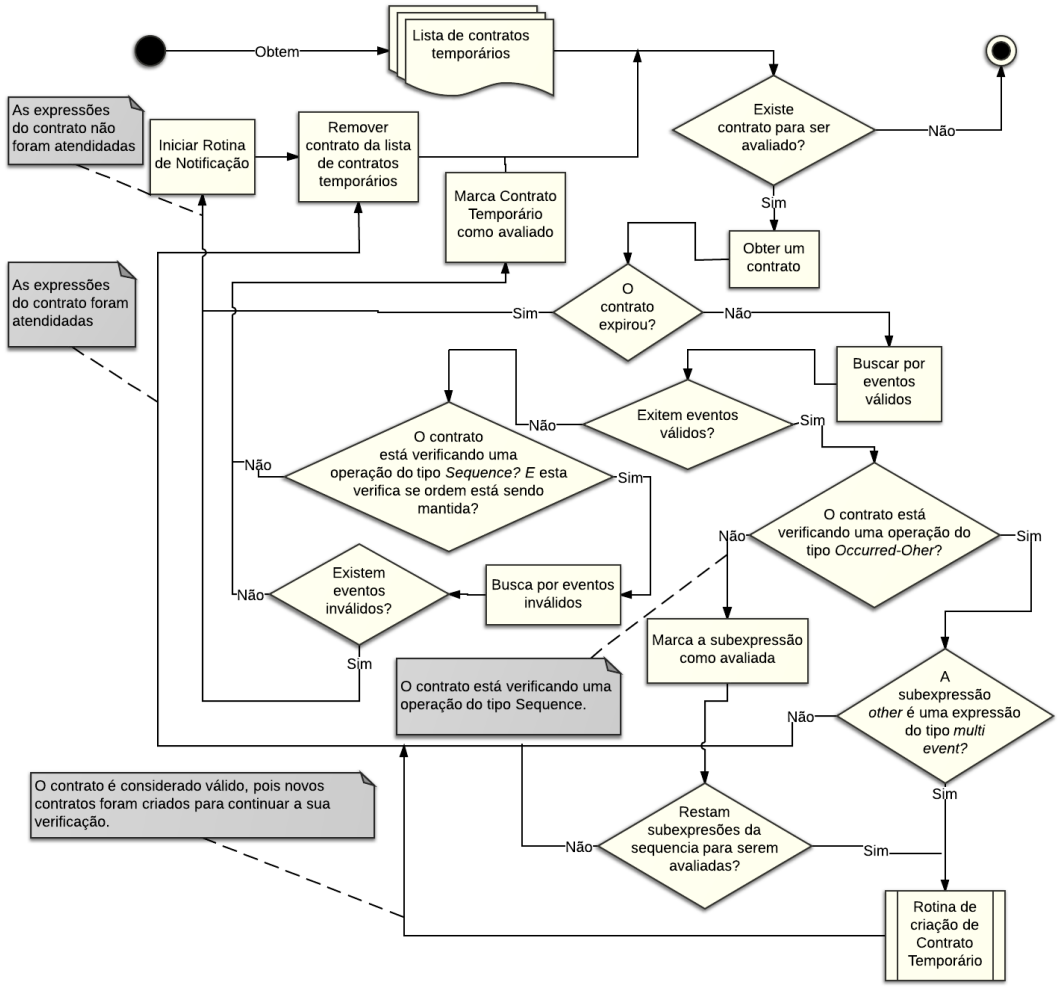

Figura 8 – Diagrama de fluxo para a avaliação da lista de contratos temporários

A rotina de avaliação da lista de contratos temporários verifica se os eventos do instante corrente invalidam o Contrato Temporário, previamente criado para satisfazer a expressão *multi event* na rotina anterior. Essa rotina possui como entrada o conjunto de eventos do instante a ser verificado e, como

saída, a notificação em caso de identificação de uma falha. O pseudo-código da rotina de avaliação da lista de contratos temporários é apresentado a seguir.

```
1    Obter um contrato.
2    Se o contrato expirou:
     2.1    Iniciar a Rotina de Notificação.
     2.2    Remover o contrato da lista de contratos temporários.
3    Caso contrário:
     3.1    Buscar por eventos válidos.
     3.2    Se existir evento válido retornado dessa busca:
            3.2.1   Se o contrato estiver verificando uma operação do tipo Occurred-
                    Other:
                    3.2.1.1   Se a subexpressão Other é do tipo multi event, então,
                              executar a Rotina de criação de Contrato Temporário.
            3.2.2   Caso contrário, o contrato estará verificando uma operação do tipo
                    Sequence:
                    3.2.2.1   Marcar a subexpressão como avaliada.
                    3.2.2.2   Se restar uma subexpressão da sequencia para ser avaliada,
                              então, executar a Rotina de criação de Contrato Temporário.
            3.2.3   Remover o contrato da lista de contratos temporários.
     3.3    Caso contrário:
            3.3.1   Se o contrato estiver verificando uma operação Sequence e esta
                    estiver verificando se a ordem está sendo mantida
                    3.3.1.1   Buscar por eventos que inválidos:
                    3.3.1.2   Se existir evento inválido retornando dessa busca:
                              3.3.1.2.1   Iniciar a Rotina de Notificação;
                              3.3.1.2.2   Remover o contrato da lista de contratos
                                          temporários;
                              3.3.1.2.3   Continuar a verificação dos contratos seguintes.
     3.4    Marca o contrato como avaliado.
4    Continuar a verificação dos contratos seguintes.
```

O diagrama de fluxo da rotina de criação de Contrato Temporário é apresentado na Figura 9. O pseudo-código dessa rotina é apresentado a seguir.

```
1    Para cada evento válido encontrado:
     1.1    Obter um evento.
     1.2    Criar um novo Contrato Temporário, adicionando o evento na sua lista de
            eventos visitados.
     1.3    Adicionar o Contrato Temporário na lista de contratos temporários.
```

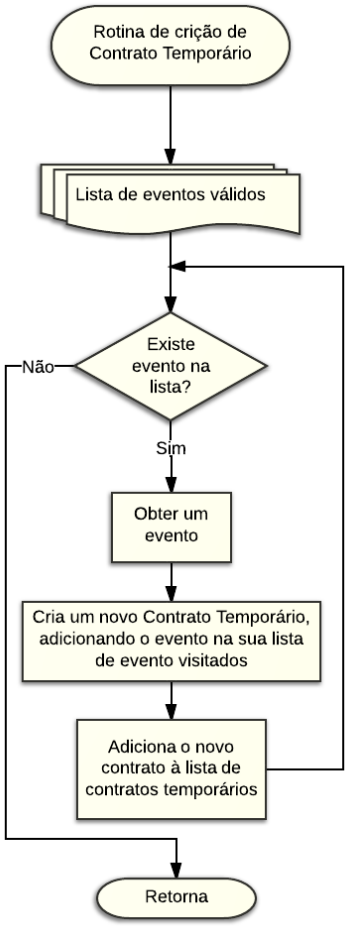

Figura 9 – Diagrama de fluxo para a Rotina de criação de Contrato Temporário

Um exemplo de funcionamento da etapa de avaliação dos eventos é ilustrado nas Figuras 10, 11, 12 e 13. Na Figura 10, são apresentados o fluxo de eventos presente da base de dados e os contratos a serem verificados. Nas demais figuras, são apresentados os instantes entre 0ms e 11ms da rotina de verificação onde, durante a avaliação dos eventos, os contratos temporários são gerados e, juntamente com os contratos, são verificados como válidos ou inválidos.

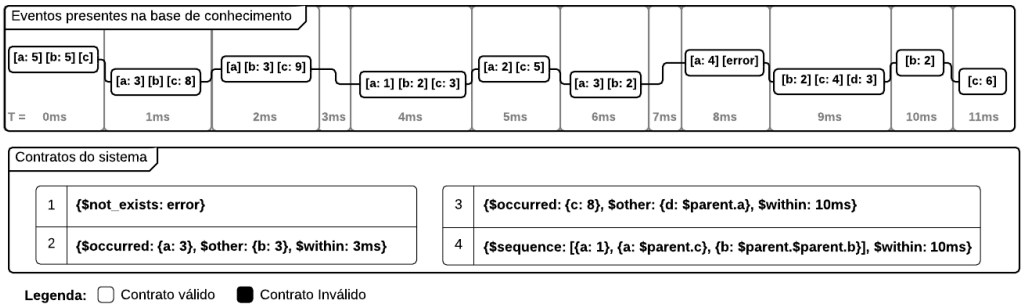

Figura 10 – Eventos presentes na base de dados e contratos do sistema

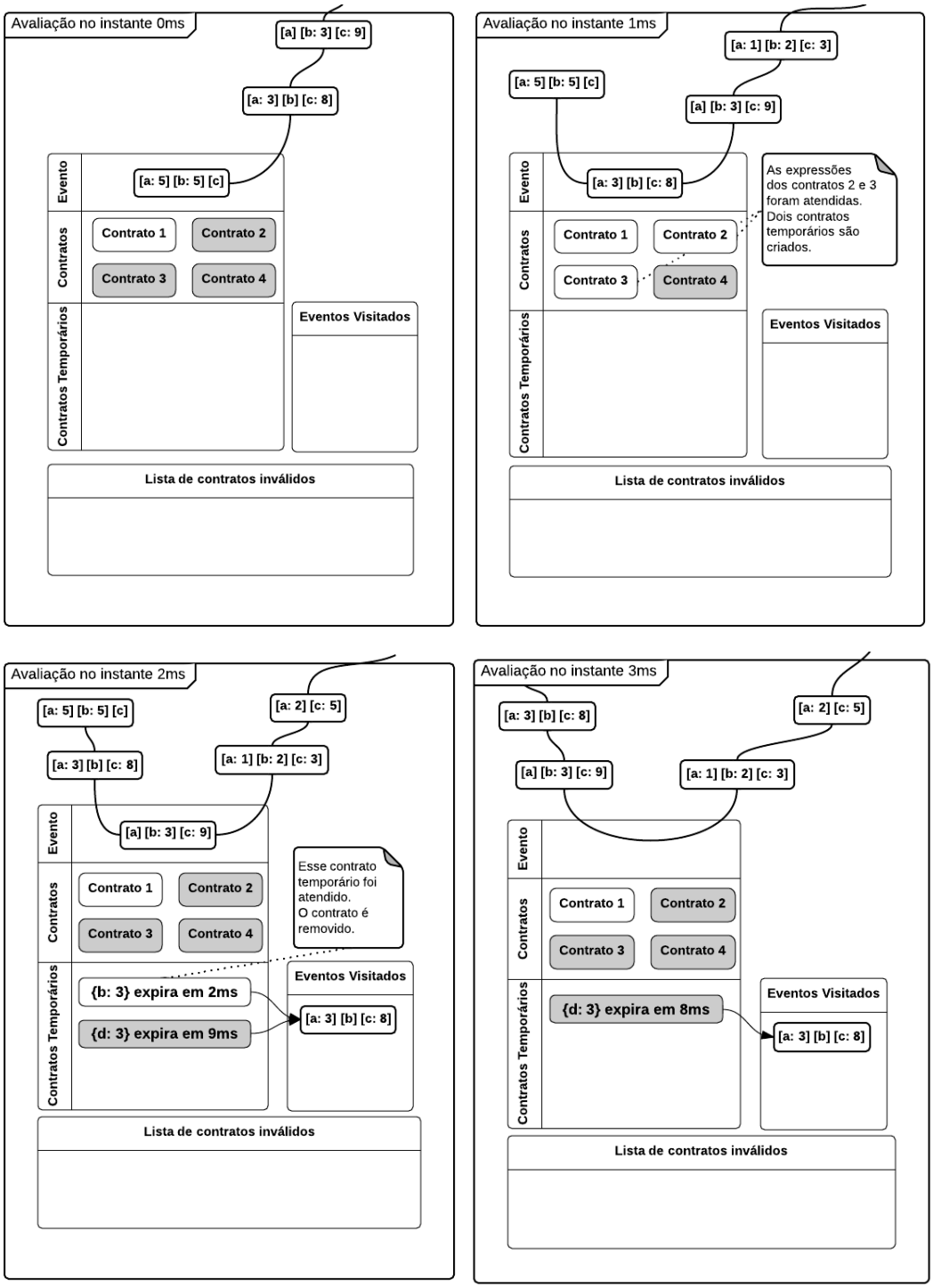

Figura 11 – Avaliação dos eventos nos instantes 0ms, 1ms, 2ms e 3ms

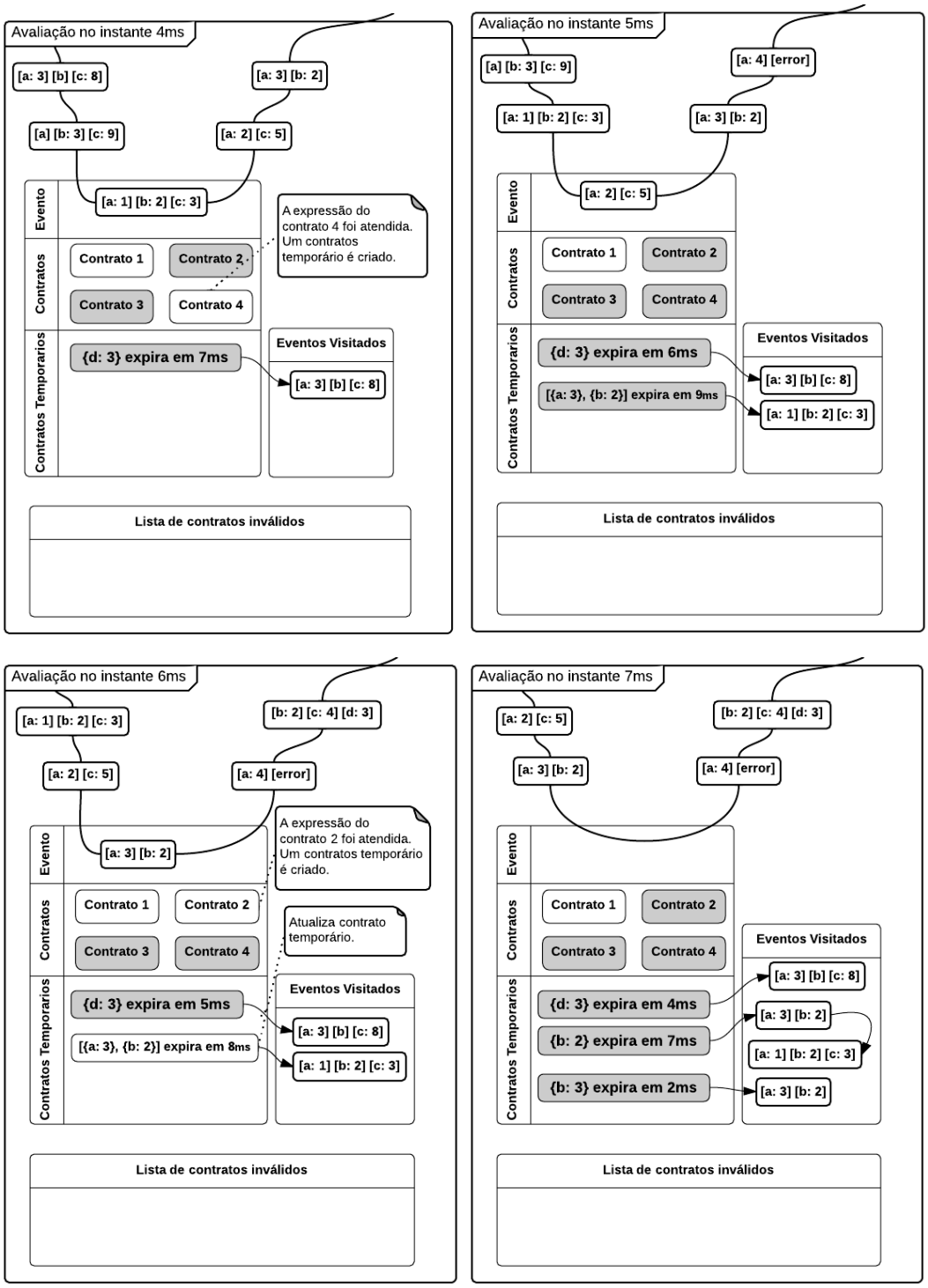

Figura 12 – Avaliação dos eventos nos instantes 4ms, 5ms, 6ms e 7ms

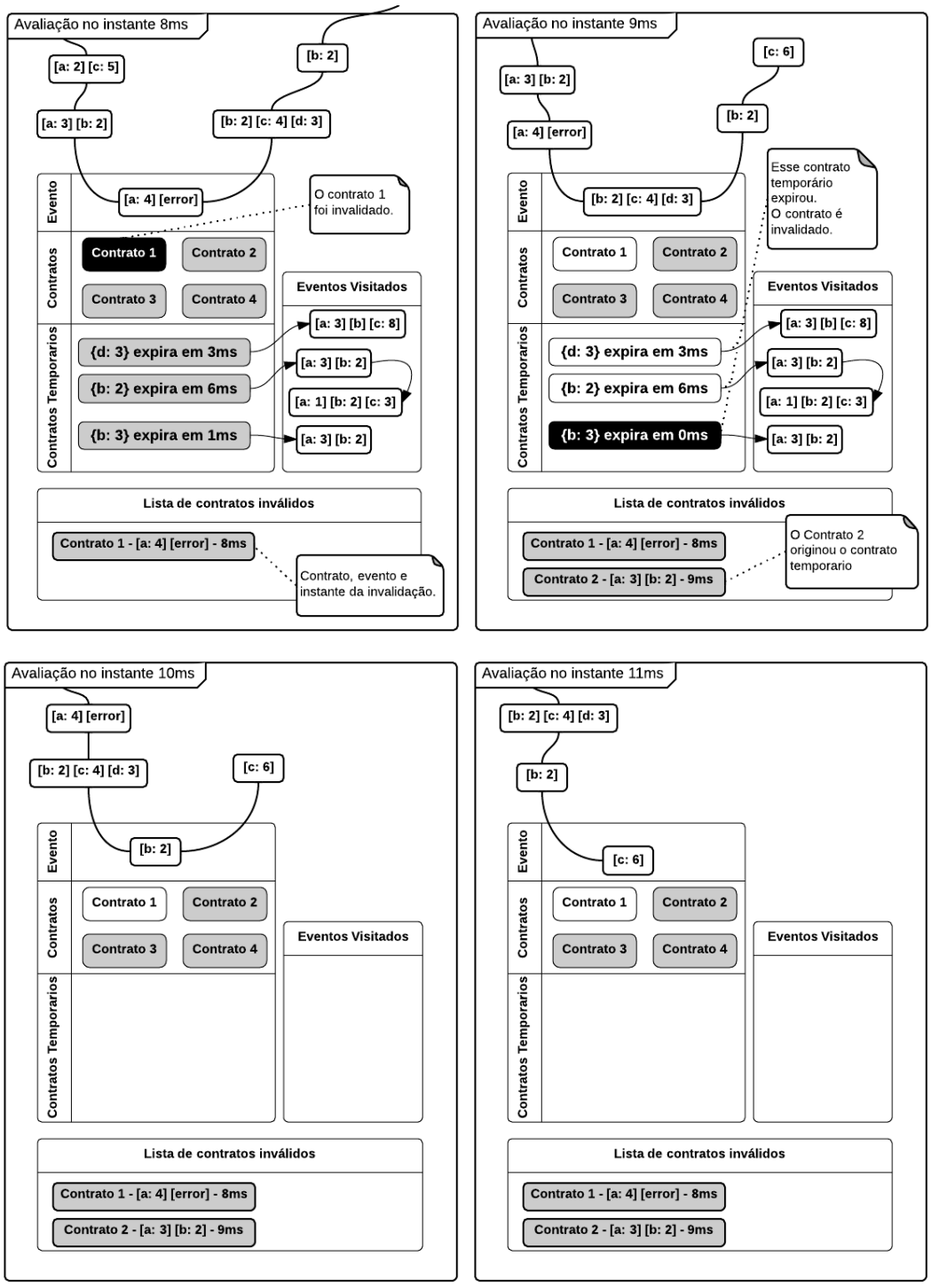

Figura 13 – Avaliação dos eventos nos instantes 8ms, 9ms, 10ms e 11ms

3.3. Diagrama de classe

O diagrama de classe da ferramenta implementada, ilustrado na Figura 14, é uma adaptação da arquitetura dos componentes do mecanismo de verificação, descrita na seção 2.2.. O detalhamento das classes é apresentado nas Figuras 15, 16 e 17.

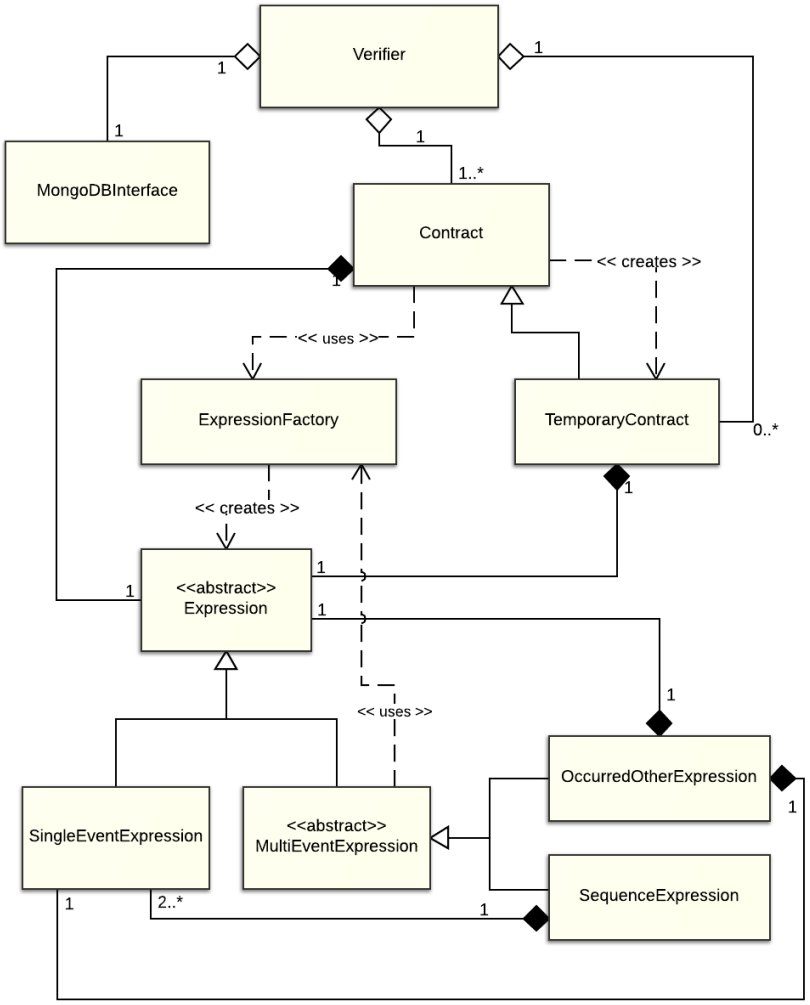

Figura 14 – Diagrama de classe da ferramenta implementada

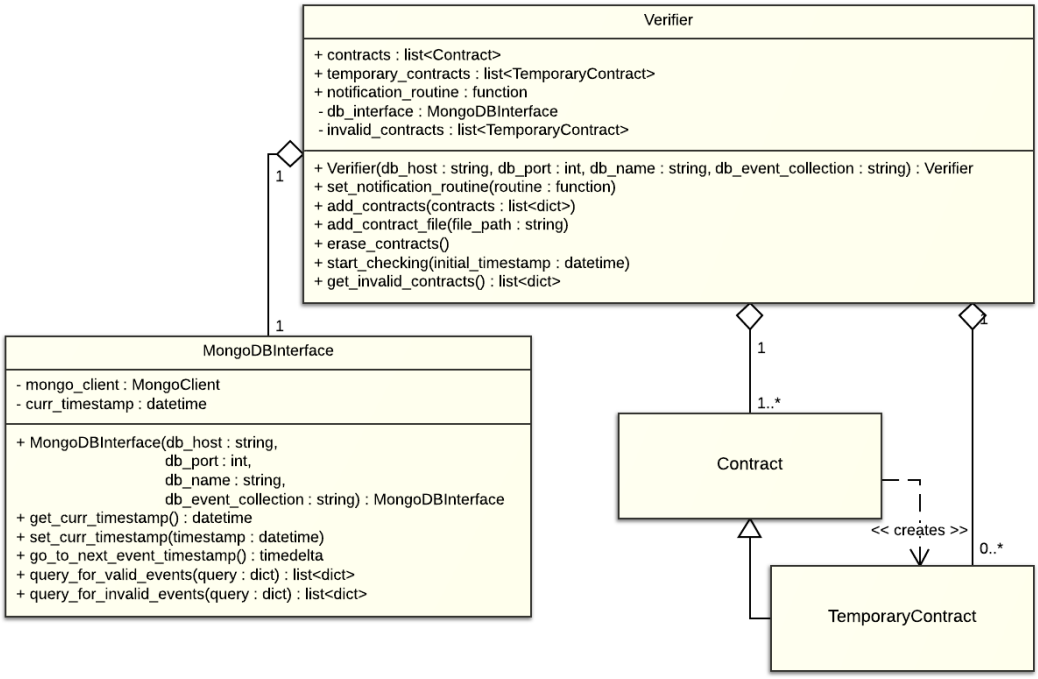

Figura 15 – Classes Verifier e MongoDBInterface

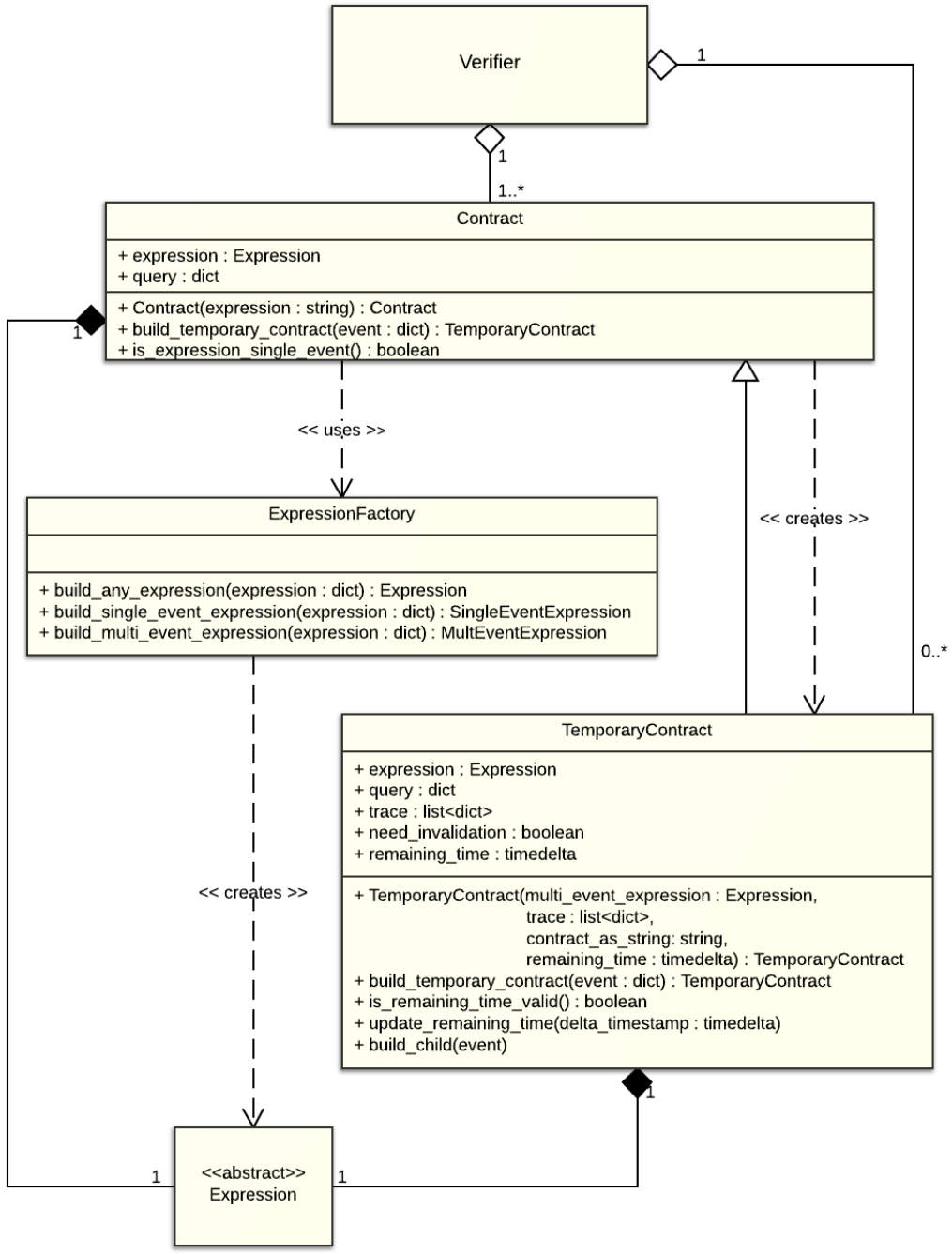

Figura 16 – Classes Contract, TemporaryContract e ExpressionFactory

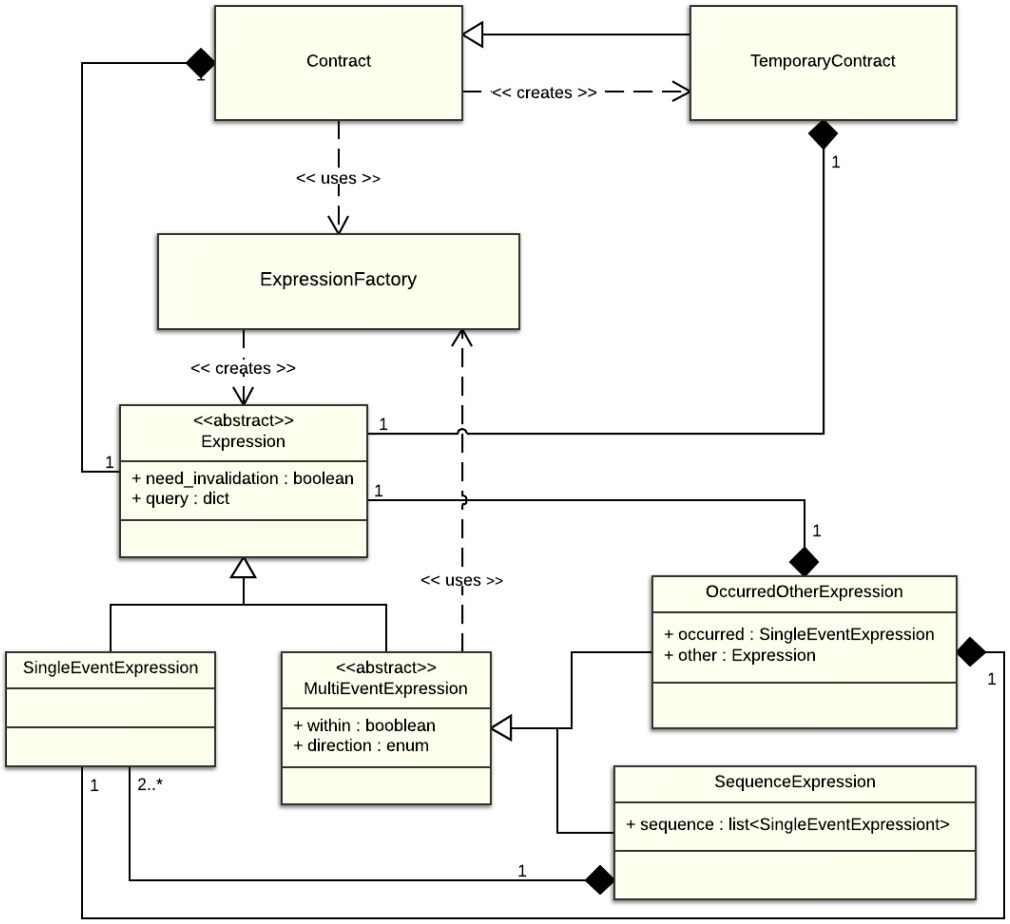

Figura 17 – Classes Expression, SingleEventExpresion, MultiEventExpresion, OccurredOtherExpresion e SequenceExpresion

O procedimento de inicialização da ferramenta instancia a classe Verifier e disponibiliza o console da ferramenta para o mantenedor. As operações “Adicionar arquivo com contratos”, “Definir a Rotina de Notificação”, “Iniciar a verificação” e “Listar contratos invalidados”, realizadas pelo console, chamam, respectivamente, os seguintes métodos da classe Verifier: *add_contract_file, set_notification_routine, start_verification* e *get_invalid_contracts*,

O Verifier é responsável por estabelecer a comunicação com a base de dados não estruturada, através de uma instancia da classe MongoDBInterface, e armazenar as listas de Contratos e Contratos Temporários, que são construídas durante a rotina de verificação. O Verifier, ao receber o arquivo de contratos escritos pelo mantenedor, inicia um processo de construção de instancias da classe Contract, isto é, desserialização dos contratos textuais. A classe Contract, por sua vez, utiliza a fábrica ExpressionFractory para converter expressões do contrato em expressões MongoDB, armazenando-as em instâncias SingleEventExpression ou MultiEventExpression, em função dos operadores utilizados.

Durante a etapa de verificação, quando um contrato composto por instância MultiEventExpression é valido, este instancia um TemporaryContract e o adiciona na lista de contratos temporários.

3.4. Detalhes técnicos da implementação da ferramenta

A ferramenta de verificação foi implementada em Python (2013), no sistema operacional OS X Mavericks 10.9.4 (2013), utilizando:

- PyCharm 3.4.1 (JetBrains, 2013) - ferramenta de desenvolvimento de software (IDE);
- MongoDB (2014) - banco de dados não relacional para o armazenamento do *log* estruturado;
- JSON (ECMA, 2013) e YAML (Ben-Kiki et al., 2009) - bibliotecas para auxíliar a desserialização dos Contratos Textuais em Contratos;
- PyMongo (2014) - ferramentas Python de comunicação com o MongoDB;
- Mercurial Hg (2006) – ferramenta de controle de versão;
- Unittest (Hamill, 2005) - *framework* para a criação de testes de unidade automatizados,.

Cabe ressaltar que, não há restrições quanto ao uso da ferramenta em outro sistema operacional ou à sua implementação em outra linguagem, uma vez que bibliotecas semelhantes às utilizadas estão disponíveis em outras linguagens.

Os testes de unidade automatizados auxiliaram a validação da ferramenta, ao longo da evolução da sua implementação, tendo sido realizados antes da etapa de avaliação em um sistema real na área da robótica.

4
Avaliação

A solução proposta foi aplicada ao sistema de software do Projeto de um Robô para Monitoramento Ambiental (RoMA), um sistema distribuído implantado em uma arquitetura física definida por: um computador embarcado, um micro-controlador embarcado e um computador na base de controle. Cada unidade da arquitetura é suportada por um sistema de componentes capaz de trocar mensagens com as demais unidades. O robô é composto por diversos equipamentos, tais como GPS, acelerômetro, sonda, controladores de motor de roda, de suspensão, de cambagem, sensor de temperatura, ultrassom entre outros. Um dos requisitos do projeto RoMA é o grau satisfatório de flexibilidade que possibilite a alteração das características do robô através da adição ou remoção de componentes, de acordo com o objetivo da operação em campo. Outro requisito é a confiabilidade, pois devido ao difícil acesso à unidade robótica durante a operação em campo, esta deve ser tolerante a falhas. Além disso, este robô é autônomo, ou seja, ele é capaz de continuar a operação em campo, com certas limitações, mesmo que ocorra uma perda de conexão entre a unidade robótica e a unidade de controle. No entanto, cada componente possui um ciclo de vida diferente, podendo falhar, desligar e religar, com um pequeno impacto nos demais componentes. Logo, o sistema pode entrar em um estado inconsistente, o que dificulta o seu monitoramento. Devido à complexidade do sistema distribuído de componentes do projeto RoMA, é esperado que falhas sejam difíceis de serem detectadas por métodos tradicionais de verificação de contratos. Isto representou uma oportunidade para a aplicação e avaliação do mecanismo de verificação de contratos proposto na seção 2.2.

O sistema de software foi desenvolvido utilizando o *middleware* Robot Operating System (ROS) (ROS, 2014), que fornece mecanismos para a troca de mensagens entre os componentes, mesmo que estes estejam localizados em maquinas diferentes. O *middleware* implementa os padrões de mensagem *publish-subscribe* e *request-response*, denominados como tópicos e serviços, respectivamente. Durante o desenvolvimento do sistema, alguns dos equipamentos do robô não estavam disponíveis para auxiliar na tarefa de teste do seu respectivo componente de software. Portanto, componentes *mock* foram

desenvolvidos para simular a interação do sistema com esses equipamentos, de forma a viabilizar a construção dos cenários de falha necessários para testar o componente. Uma vez concluído o sistema, foi realizado um teste com a unidade robótica contendo todos os equipamentos. Cabe ressaltar que, somente um dia de trabalho foi suficiente para identificar e corrigir as falhas provenientes das características dos equipamentos reais, convergindo o sistema para uma versão estável.

Para a avaliação da viabilidade e adequação da solução proposta, seis cenários de falha interessantes e com alta probabilidade de ocorrência foram selecionados para serem avaliados. A seguir, para cada cenário, são descritos: a situação de falha correspondente; o contrato capaz de identificá-la; os eventos e *tags* necessárias e uma explicação da sua relevância para a avaliação.

4.1. Cenário de falha 1 – Valor fora do limite

O sensor de ultrassom se comunica, através de uma porta analógica, com o *micro* controlador, que realiza amostragem e a processa em um algoritmo para tratar o ruído. Sabe-se que o algoritmo deve resultar em um valor entre 0 e 1024. Apesar do *uint16* ser o tipo do campo de mensagem mais adequado para armazenar esse valor, esse tipo não garante que o valor seja menor que 1024. Através das técnicas tradicionais, essa falha pode ser facilmente tratada utilizando uma assertiva de saída. Entretanto, esse cenário será usado para exemplificar a verificação de um contrato em um caso lógico.

Inicialmente, utilizando o mecanismo de verificação de contratos proposto, o componente do *micro* controlador foi instrumentado para notificar eventos que continham o resultado da amostragem. A seguir é apresentado como um contrato foi redigido, utilizando a gramática, para verificar se o valor publicado estava entre 0 e 1024.

```
Tags necessárias: component e ultrasound_value
Contrato:
   {$if: {component: ultrasound},
    $then: {$and: [{$compare: (ultrasound_value, >=, 0)},
                   {$compare: (ultrasound_value, <=, 1024)}]}}
```

4.2. Cenário de falha 2 – Componentes param de funcionar

Conforme explicado anteriormente, cada componente possui um ciclo de vida diferente, podendo falhar, desligar e religar, com um pequeno impacto nos demais componentes. Caso um componente falhe interrompendo o seu ciclo de

execução, o sistema pode não observar que o componente falhou e prosseguir com a execução, podendo levar o robô a uma situação de risco. Uma solução para esse problema é a emissão de sinais *keep-alive* dos componentes para identificar quais componentes interromperam a sua execução.

Essa solução aplicada em métodos tradicionais resultaria na criação de um componente responsável por verificar se algum componente do sistema interrompeu sua execução, através de publicações de sinais *keep-alive* em um tópico de mensagem. Por sua vez, os demais componentes do sistema devem publicar, a cada iteração do ciclo de execução, um sinal *keep-alive*, informando que ainda estão ativos. Em seguida, caso o componente verificador observe a falta de sinal *keep-alive*, em um período muito extenso, este componente notifica a falha ao mantenedor do sistema. Apesar de solucionar o problema citado, esse método não garante que o ciclo de execução do componente verificador não seja interrompido por uma falha. Portanto, seria necessário outro componente verificador para verificar o seu ciclo de execução, e assim sucessivamente.

Utilizando o mecanismo de verificação de contratos proposto, os componentes foram instrumentados para notificar eventos com a *tag* "*keep-alive"* a cada ciclo de execução. Em seguida, um contrato foi escrito para verificar a periodicidade máxima dos eventos com *tag keep-alive*. Isto é, sempre que for observada a ocorrência de um evento com a *tag keep-alive*, deve ser aguardado, no máximo, T segundos para que ocorra outro evento com a *tag keep-alive* originado do mesmo componente. A seguir é apresentado como este contrato foi redigido utilizando a gramática.

```
Tags necessárias: keep_alive e component
Contrato:
   {$occurred: {keep_alive: $exists},
    $other: {$and: [{keep_alive: $exists},
                    {component: $parent.component}]},
    $within: 5s}
```

4.3. Cenário de falha 3 – Mensagens não chegam ao destino

A comunicação entre os componentes da unidade robótica e da unidade controle, estabelecida através de uma conexão SSH (2014), pode falhar. Nesse caso, as mensagens publicadas por componentes da unidade robótica podem não ser recebidas pelos componentes da unidade de controle. Utilizando as técnicas tradicionais não conseguimos elaborar uma solução de verificação.

Inicialmente, utilizando o mecanismo de verificação de contratos proposto, foram listados os tópicos usados na comunicação entre os componentes das unidades robótica e de controle. Em seguida, foram instrumentados os

componentes para notificar eventos, quando uma mensagem é publicada nos tópicos selecionados. Uma instrumentação análoga foi aplicada aos componentes que recebem as mensagens. Por fim, um contrato foi escrito para verificar mensagens que tenham sido publicadas, mas não recebidas. Isto é, sempre que for observada a ocorrência de um evento com uma *tag* informando a publicação, deve ser aguardado, no máximo, T segundos para que ocorra outro evento com *tags* informando o recebimento e o mesmo identificador da mensagem publicada. A seguir é apresentado como este contrato foi redigido utilizando a gramática.

```
Tags necessárias: message_id e topic_action
Contrato:
   {$occurred: {topic_action: publish},
    $other: {$and: [{topic_action: receive},
                    {message_id: $parent.message_id}]},
    $within: 1s}
```

4.4. Cenário de falha 4 – Equipamento repete amostras

O Xsens (2014) é um equipamento utilizado pelo robô e composto por acelerômetros, GPS, sensor de temperatura, barômetro e outros. Um componente foi desenvolvido para a comunicação do Xsens através de uma porta RS232. Apesar do protocolo de comunicação ter sido implementado corretamente, falhas foram observadas na comunicação com o equipamento. Com o decorrer do uso do Xsens, observou-se seguinte problema intermitente: apesar da unidade robótica estar se movendo em uma direção, a amostra do acelerômetro permanece inalterada, o que é uma situação improvável, pois mesmo quando ele está visualmente parado, ocorre uma variação nas amostras por conta de vibrações do robô. Portanto, esta é uma situação em que o equipamento, provavelmente, tenha travado em uma amostra e a está replicando. Utilizando as técnicas tradicionais não conseguimos elaborar uma solução de verificação.

O componente foi instrumentado para notificar um evento para cada amostra recebida, utilizando o mecanismo de verificação de contratos proposto. A seguir é apresentado como um contrato foi redigido para verificar se os valores das amostras estão variando.

```
Tags necessárias: component, sample, accel_x, accel_y e accel_z
Contrato:
   {$occurred: {$and: [{component: xsens},
                       {sample: accelerometer}]},
    $other: {$and: [{component: xsens},
                    {sample: accelerometer},
                    {$or: [{$compare: (accel_x, !=, $parent.accel_x)},
                           {$compare: (accel_y, !=, $parent.accel_y)},
                           {$compare: (accel_z, !=, $parent.accel_z)}]}]},
    $within: 100ms}
```

4.5.
Cenário de falha 5 – Valor imutável

Um componente responsável por ler a amostragem do *joystick* publica os valores dos seus eixos x e y, em porcentagem. Estes valores precisam passar por três filtros, que tratam o valor através de algoritmos para verificar colisão, antes de serem enviados para os quatro motores. Cabe ressaltar que, os algoritmos utilizam os valores dos eixos do *joystick*, sem alterar o seu valor, e apenas o último filtro é o responsável por definir se a mensagem deverá ser repassada para os motores, com base no cálculo dos filtros anteriores. Por isso, é necessário garantir que as futuras modificações feitas nesses algoritmos não alterem os valores dos eixos do *joystick*, durante a sua passagem pelos filtros de colisão. Utilizando as técnicas tradicionais não conseguimos elaborar uma solução de verificação.

Inicialmente, utilizando o mecanismo de verificação de contratos proposto, o primeiro e segundo filtros foram instrumentados para notificar eventos com os valores enviados para o filtro seguinte. O terceiro filtro, o último deles, foi instrumentado para notificar um evento para cada mensagem enviada para os motores. A seguir é apresentado como um contrato foi redigido para verificar se os valores do *joystick* estão se mantendo iguais, durante a sua passagem pelos filtros de colisão.

```
Tags necessárias: component, action, joy_x e joy_y
Contrato:
   {$sequence: [{$and: [{component: joystick}, {action: pub_axis}]},
                {$and: [{component: filter_A}, {action: process_joy},
                        {joy_x: $parent.joy_x}, {joy_y: $parent.joy_y}]},
                {$and: [{component: filter_B}, {action: process_joy},
                        {joy_x: $parent.joy_x}, {joy_y: $parent.joy_y}]},
                {$and: [{component: filter_C}, {action: process_joy},
                        {joy_x: $parent.joy_x}, {joy_y: $parent.joy_y}]},
    $within: 1s}

   {$occurred: {$and: [{component: filter_C}, {action: pub_joy_axis}]},
    $other: {$and: [{component: filter_C}, {action: process_joy},
                    {joy_x: $parent.joy_x}, {joy_y: $parent.joy_y}]},
    $within: 10ms}
```

4.6.
Cenário de falha 6 – Lâmpada queimada

A lanterna frontal da unidade robótica é controlada por um componente localizado no *micro*-controlador, que atende a requisições de ligar ou desligar, através de serviços chamados pela unidade de controle. Observou-se a impossibilidade de identificar, automaticamente, a partir do *micro* controlador, se

a lanterna está queimada. Entretanto, observando a câmera frontal mudar do estado noturno para diurno, é possível verificar, manualmente, se a lanterna ligou. Utilizando as técnicas tradicionais não conseguimos elaborar uma solução de verificação.

Inicialmente, utilizando o mecanismo de verificação de contratos proposto, os componentes responsáveis pela lanterna e câmera frontal foram instrumentados para notificar eventos nos instantes em que a câmera troca, automaticamente, de tipo de visão e em que a lanterna recebe a requisição para ligar. A seguir é apresentado como um contrato foi redigido para verificar se a lanterna está queimada.

```
Tags necessárias: component, change_vision_mode e request
Contrato:
   {$occurred: {$and: [{component: flash},
                       {request: turn_on}]},
    $other: {$and: [{component: camera},
                    {change_vision_mode: day}]},
    $within: 5s}
```

4.7. Resultados

Todos os contratos apresentados nos cenários de falhas, descritos nas seções 4.1 a 4.6, foram aplicados no sistema RoMA e avaliados através de operadores implementados nos *mocks*, capazes de gerar, aleatoriamente, falhas durante a execução. Os resultados da avaliação com os cenários de falha são apresentados na Tabela 1.

	Número de falhas ocasionadas	Número de falhas verificadas	Número de eventos gerados
Cenário 1	8	8	22
Cenário 2	17	17	43
Cenário 3	20	20	40
Cenário 4	15	15	45
Cenário 5	59	59	151
Cenário 6	10	10	50

Tabela 1 – Resultados da avaliação com os cenários de falha

O impacto da aplicação do mecanismo no desempenho do sistema, para cada cenário de falha selecionado, foi medido calculando-se o tempo médio do ciclo de avaliação de contratos. Os resultados são exibidos na Tabela 2.

	Tempo médio do ciclo de avaliação de contratos (ms)
Cenário 1	0.840
Cenário 2	0.464
Cenário 3	0.467
Cenário 4	0.486
Cenário 5	1.711
Cenário 6	0.588

Tabela 2 – Tempo médio de avaliação de contratos para os cenários de falha

Observou-se que a média do tempo médio de avaliação de um contrato é 0.759 milissegundos, com um desvio padrão de 0.487 milissegundos, em um computador Intel® Core i5 3.3GHz x 4.

5
Estado da Arte

Na pesquisa bibliográfica foram encontradas quatro abordagens distintas para a verificação de anomalias durante a execução de um sistema: (i) busca por padrões pré-definidos; (ii) avaliação com base na especificação formal completa; (iii) comparação de máquinas de estados; e (iv) avaliação orientada a aspectos.

A abordagem de busca por padrões pré-definidos consiste no monitoramento do *log* do sistema em busca de padrões informados pelo mantenedor, com o objetivo de identificar situações de falha. Caso seja encontrado um evento que esteja enquadrado em um dos padrões, uma notificação é emitida para o mantenedor responsável, ou mesmo é iniciada uma rotina de recuperação associada ao padrão. Swatch (Hansen & Atkins, 1993) e SEC (Vaarandi, 2002) são exemplos de ferramentas que empregam essa abordagem, utilizando técnicas de *Data Mining* para obtenção das propriedades do *log*. Por exemplo, caso o padrão "*Roteador:<router>. Interface:<interface>, mudou de estado para desligado*" seja identificado, emitir uma notificação "*<router> interface <interface> desligou*". Para a obtenção das propriedades usadas na notificação, expressões regulares podem ser utilizadas no lugar de *<router>* e *<interface>*. Essa abordagem auxilia na verificação de anomalias em um *log* tradicional, podendo ser aplicada a sistemas capazes de produzir esse tipo de *log.* Além disso, as notificações ou rotinas de recuperação podem estar associadas a padrões que observam um evento ou a relação entre eventos. Entretanto, a presença de um alto índice de falsos-positivos devido ao uso de *log* tradicional (Lou, 2010) é um ponto fraco desta abordagem. Outro ponto fraco é a limitação da quantidade de propriedades existentes no *log*, que impossibilita a verificação, de forma precisa, das falhas lógicas de baixo nível em sistemas. A nossa solução, por outro lado, é adequada para verificar, de forma precisa, as falhas na lógica dos componentes do sistema, uma vez que os eventos utilizados na verificação são gerados através de uma instrumentação capaz de extrair propriedades de contexto, sem a necessidade de resgatá-las de uma informação puramente textual, como nas abordagens previamente descritas.

A abordagem de avaliação com base na especificação formal completa consiste no monitoramento do comportamento do sistema distribuído em

execução, comparando a execução com as expectativas especificadas pelo programador em um documento formal, com o objetivo de identificar situações de falha. Todo trecho de execução é classificado como válido ou inválido, e em seguida apresentado ao mantenedor. Pip (Reynolds et al, 2006) e MTAT (Hendrickson et al., 2003) são exemplos de ferramentas que utilizam essa abordagem, por meio da observação dos eventos do sistema e mensagens trocadas entre componentes. Essa abordagem auxilia o mantenedor a descobrir ou verificar, com exatidão, o comportamento do sistema através de sua interface de comportamentos válidos e inválidos. Além disso, a especificação é escrita utilizando uma linguagem descritiva, o que possibilita aos programadores especificar expectativas acerca da estrutura do sistema, assim como o período entre dois eventos e o consumo de recursos. Entretanto, esta abordagem possui dois pontos fracos. O primeiro é o esforço necessário para construir a especificação, a qual pode ser incorreta ou imprecisa, uma vez que esta é escrita por humanos, que podem errar (Brown & Patterson, 2001). O segundo ponto fraco é o impacto no desempenho do sistema: uma verificação profunda, abrangendo toda a especificação formal, implicaria em um volume de dados elevado, logo sendo inviável para um sistema executado no ambiente de produção. Por outro lado, uma avaliação superficial, por exemplo, somente nas interfaces dos componentes, impediria a identificação de falhas na lógica interna do componente. A nossa solução é mais eficaz, no entanto menos eficiente, em que os desenvolvedores e mantenedores do sistema indicam quais situações de falha devem ser verificadas através da escrita do seu contrato. O resultado produzido é mais eficaz devido ao conhecimento prévio que os desenvolvedores e mantenedores possuem sobre o comportamento do sistema, o que os tornam capaz de identificar com precisão quais situações de falha são relevantes.

A abordagem de comparação de máquina de estados (Mariani & Pastore, 2008; Lorenzoli et al., 2008; Tan et al., 2008) consiste no monitoramento do comportamento do sistema através de uma máquina de estados. Esta máquina é gerada através de um algoritmo que analisa os *logs* gerados pelo sistema e, com base na repetição e causalidade, identifica os possíveis caminhos que a execução pode seguir. Portanto, a máquina de estados resultante é apresentada como um grafo em que os vértices são as notificações de *log*, logo, a qualidade do resultado está diretamente relacionada à granularidade destas notificações. A técnica de identificação de falhas funciona da seguinte forma: inicialmente o mantenedor coloca o sistema em uso, supervisionando a execução, em busca de uma máquina de estados isenta de falhas a partir de uma versão correta do

sistema. Em seguida, o sistema é colocado no ambiente de produção e a máquina de estados gerada inicialmente é comparada com a máquina de estados resultante da execução, em busca de arestas desconhecidas. Caso seja observada uma aresta na máquina de estados resultante que não esteja presente na máquina de estados isenta de falhas, esta aresta pode ser considerada uma falha. AVA (Babenko et al., 2009) e BTC (Mariani et al, 2011) são exemplos de ferramentas que empregam essa abordagem. O ponto forte dessa abordagem é a sua completa automatização. Por outro lado, podem ser gerados muitos falsos-positivos, ou seja, existir arestas na máquina de estados resultante que não são erros, porém também não estão presentes na máquina de estados isenta de falhas, pelo fato da funcionalidade correspondente não ter sido estimulada durante a geração da máquina de estados inicial. Portanto, maior será o esforço do mantenedor para identificar as arestas que, realmente, representam falhas. Também é possível que eventos que representam falhas estejam presentes na execução inicial, porém sem o conhecimento do mantenedor, uma vez que ele não foi capaz de observar as ocorrências destas falhas durante o treinamento do *log*. Portanto, não é possível garantir que a máquina de estados gerada a partir de uma versão correta do sistema seja, realmente, isenta de falhas, o que pode ocultar falhas durante a comparação das máquinas de estados.

A abordagem de avaliação de sistemas distribuídos orientada a (Navarro et al., 2006a, b, 2008) consiste em escrever contratos utilizando a linguagem AWED e realizar sua verificação durante a execução do sistema. Essa linguagem possibilita expressar propriedades de máquinas de estados esperadas em um sistema, onde as arestas representam os pontos de interesse, *pointcuts*, dos aspectos, para a depuração de sistemas distribuídos concorrentes A automatização da etapa de instrumentação do código é um ponto forte dessa abordagem. Contudo, poucas linguagens de programação, como Java, suportam aspectos e, além disso, o usuário deve conhecer aspectos para escrever os contratos. A linguagem AWED apesar de possuir operadores para a verificação de causalidade como, por exemplo, o operador de sequência, não realiza verificação temporal. A nossa solução, por outro lado, é mais flexível pois tem como base um log com informação contextual, que possui *tags* com informações relevantes do instante em que os eventos são notificados. Isso possibilita redigir contratos envolvendo pontos diferentes do sistema, utilizando informações que não estariam disponíveis juntas devido a modularidade.

6
Conclusão

Esse trabalho de pesquisa contribuiu com um mecanismo baseado em logs com meta-informações para a verificação de contratos em sistemas distribuídos e uma gramática para redigir contratos que possibilita operações temporais, ou seja, permite a especificação de condições entre eventos, em diferentes instantes de tempo, ou mesmo garante uma sequência de eventos, durante um período de tempo. O fluxo de eventos gerado é avaliado assincronamente em relação à utilização do sistema, pela comparação com contratos, previamente escritos de acordo com a gramática, que representam as expectativas sobre o comportamento normal do sistema.

Para a avaliação da viabilidade e adequação da solução proposta, foram realizados experimentos com um conjunto de falhas selecionado de um sistema real, na área da robótica. Os resultados obtidos nos experimentos demonstraram que o mecanismo de verificação é promissor, uma vez que foi capaz de identificar todas as falhas injetadas no sistema robótico, com um baixo impacto no seu desempenho. Entretanto, esse impacto no desempenho não pode ser comparado com o desempenho do sistema em execução, uma vez que cada componente possui um ciclo de vida diferente, requerendo um processamento variável ao longo da execução do sistema.

Por outro lado, foram encontradas algumas limitações. Além daquelas descritas na seção 2.4, não foram implementados operadores de cálculo de valores e de contagem de ocorrência de eventos, necessários para redigir contratos para, por exemplo, verificar o consumo de banda de conexão entre as unidades robótica e controle e verificar o crescimento da frequência de publicações nos tópicos, respectivamente.

São propostos os seguintes tópicos para serem desenvolvidos em trabalhos futuros:

- aprimoramento da gramática visando solucionar as suas limitações;
- adaptação da arquitetura da solução para utilizar outros banco de dados;

- aplicação do mecanismo em experimentos realizados em outros tipos de sistemas, a fim de demonstrar que o mecanismo é eficaz para um grande grupo de sistemas.

Referências bibliográficas

ANDREWS, D.M. Using executable assertions for testing and fault tolerance. In: Fault-Tolerant Computing Symposium**,** 9., Madison, 1979. **Anais...** Reprints from the collection of the University of Michigam Library, v.4, 1979.

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 n^{o} 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>

BABENKO, A.; MARIANI, L.; PASTORE, F. AVA: automated interpretation of dynamically detected anomalies. In: International Symposium on Software Testing and Analysis (ISSTA'09), 18., Chicago, 2009. **Anais eletrônicos...** New York: ACM, 2009. P. 237-248. DOI: 10.1145/1572272.1572300. Disponível em: < http://delivery.acm.org/10.1145/1580000/1572300/p237-babenko.pdf?>.

BAN, B.; WANG, B. JBossCache Reference Manual V. 1.2. JBoss Inc., 2005.

BEM-KIKI, O.; EVANS, C.; NET, I. YAML Ain´t mark-up language (YAML ^{TM}) version 1.2. 3.ed., 2009. Disponível em: <http://yaml.org/spec/1.2/spec.pdf>

BROWN, A.; PATTERSON, D. A. To Err is Human. In: Workshop on Evaluating and Architecting System Dependability (EASY '01), 1.,Gothenburg, 2001. Disponível em: < http://roc.cs.berkeley.edu/papers/easy01.pdf>

CHOMSKY, N. Three models for the description of language. IRE Transactions on Information Theory, 2 (3), p. 113-24, 1956. DOI:10.1109/TIT.1956.1056813 Disponível em: <http://www.historyofinformation.com/expanded.php?id=958>

ECMA - 404. The JSON Data Interchange Format. 1.ed., 2013. Disponível em: <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf >

FOX, A. Toward recovery-oriented computing. In: International Conference on Very Large Data Bases (VLDB), 28., Hong Kong, 2002. **Anais eletrônicos...**

VLDB Endowment, 2002. P. 873-876. Disponível em: <http://delivery.acm.org/10.1145/1290000/1287443/p873-fox.pdf?>

GRAMMOPHONE. Site do analisador de gramática. Disponível em: <http://mdaines.github.io/grammophone/>. Acesso em: junho de 2013.

HAMILL, P. Unit Test Frameworks. 1.ed.Sebastopol: O'Reilly Media Inc., 2005. 198p.

HANSEN, S. E.; ATKINS, E. T. Automated system monitoring and notification with swatch. In: Conference on System Administration, 7., Berkeley, 1993. **Anais...** Berkeley: USENIX, 1993. P. 145-152.

HELLERSTEIN, J.L.; MA, S.; PERNG, C. -S. Discovering actionable patterns in event data. **IBM Systems Journal**, v.41, n. 3, p. 475-493. 2002. Disponível em: < http://researchweb.watson.ibm.com/journal/sj41-3.html>

HENDRICKSON, S. A.; DASHOFY, E. M.; TAYLOR, R.N. An approach for tracing and understanding asynchronous architectures. In: International Conference on Automated Software Engineering, 18.,Montreal, 2003. **Anais eletrônicos...** Washington: IEEE Computer Society, 2003. P.318-322. DOI: 10.1109/ASE.2003.1240329. Disponível em: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1240329>

HORWITZ, S.; LIBLIT, B.; POLISHCHUK, M. Better debugging via output tracing and callstack-sensitive slicing. **IEEE Transactions on Software Engineering**, v.36, n. 1, p. 7-19. 2010. ISSN: 0098-5589. DOI: 10.1109/TSE.2009.66. Disponível em: <.http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5282499>

JETBRAINS. Site da ferramenta pycharm. Disponível em: <http://www.jetbrains.com/pycharm>. Acesso em: setembro de 2013.

LORENZOLI, D.; MARIANI, L.; PEZZE, M. Automatic generation of software behavioral models. In: International Conference on Software Engineering, 30., Leipzig, 2008. **Anais eletrônicos...** New York: ACM, 2008. P. 501-510. DOI: 10.1145/1368088.1368157. Disponível em: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4814161>

LOU, J. G. Mining dependency in distributed systems through unstructured logs analysis. **Operation System Review**, v. 44, n. 1, p. 91-96. 2010

MARIANI, L.; PASTORE, F. Automated Identification of Failure Causes in System Logs. In: International Symposium on Software Reliability Engineering,

19., Seattle, 2008. **Anais eletrônicos...** Washington: IEEE Computer Society, 2008. P.117-126. DOI: 10.1109/ISSRE.2008.48. Disponível em: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4700316>

MARIANI, L.; PASTORE, F.; PEZZE, M. Dynamic analysis for diagnosing integration faults. **IEEE Transactions on Software Engineering**, v.37, n. 4, p. 486-508. 2011. DOI: 10.1109/TSE.2010.93. Disponível em: <http://dx.doi.org/10.1109/TSE.2010.93>

MERCURIAL. Site oficial do projeto mercurial, 2006. Disponível em: <<http://www.selenic.com/mercurial/wiki/>. Acesso em: setembro de 2013.

MEYER, B. Applying design by contract. **Computer**, v. 25, n. 11, p 40-51. 1992. ISBN: 013144025X. DOI: 10.1109/2.161279. Disponível em: <http://dx.doi.org/10.1109/2.161279

MIRGORODSKIY, A. V.; MARUYAMA, N; MILLER, B. P. Problem diagnosis in large-scale computing environments. In: Conference on Supercomputing, Tampa, 2006. **Anais eletrônicos...** New York: ACM, Washington: IEEE Computer Society, 2006. 11 p. DOI: 10.1145/1188455.1188548. ISBN:0-7695-2700-0. Disponível em: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4090185>

MONGODB. Site oficial do projeto MongoDB. Disponível em: <http://api.mongodb.org>. Acesso em: janeiro de 2014.

MURCH, R. **Autonomic Computing**. 1. ed. IBM Press, 2004. 336 p.

OS X Mavericks, version 10.9.4: Sistema operacional. Apple Inc., 2013. Disponível em: <https://www.apple.com/osx>

NAVARRO, L. D. B et al. Modularization of distributed web services using Aspects With Explicit Distribution (AWED). In: OTM Confederated International Conferences, 28., Montpellier, 2006. **Anais eletrônicos...** Springer, New York, 2006. vol. 4276, P. 1449-1466. Disponível em: http://link.springer.com/chapter/10.1007/11914952_32#page-1.

_____.. Explicitly distributed AOP using AWED. In: International Conference on Aspect-oriented Software, 5., Bonn, 2006. **Anais eletrônicos...** ACM, New York, 2006. P. 51-62. ISBN:1-59593-300-X. Disponível em: http://dl.acm.org/citation.cfm?id=1119665.

NAVARRO, L. D. B.; DOUENCE, R.; SUDHOLT, M. Debugging and testing middleware withaspect-based control-flow and causal patterns. In:

ACM/IFIP/USENIX International Conference on, 9., Leuven, 2008. **Anais eletrônicos...** New York, Springer, 2008. P.183-202. ISBN:3-540-89855-7. Disponível em: http://dl.acm.org/citation.cfm?id=1496950&picked=prox&cfid=605823255&cftoken=98465940

PHILLIPS, A. Defensive Programming. Software Development. May, 2012. Disponível em: <http://devmethodologies.blogspot.com.br/2012/05/defensive-programming.html>

PYMONGO. version 2.7.2. Biblioteca. Disponível em: <http://api.mongodb.org/python/current/>. Acesso em: janeiro de 2014.

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

REYNOLDS, P. et al. Pip: detecting the unexpected in distributed systems. In: Conference on Networked Systems Design & Implementation, 3., San Jose, 2006. **Anais eletrônicos...** Berkeley: USENIX, 2006. P.115-128. Disponível em: <https://www.usenix.org/legacy/events/nsdi06/tech/reynolds.html>.

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

SSH. Secure Shell cryptographic network protocol, Disponível em: http://en.wikipedia.org/wiki/Secure_Shell . Acesso em: agosto de 2014

SCOWEN, R. S. Extended BNF — A generic base standard. In: Software Engineering Standards Symposium, Brighton, 1993. **Anais eletrônicos...** Washington: IEEE Computer Society,p. 24-34, 1993, DOI: 10.1109/ SESS.1993.263968. Disponível em: <ftp://avaloni.iks-jena.de/pub/mitarb/lutz/standards/iso/iso-iec-14977-EBNF-CompanionPaper.pdf>

SSH. Secure Shell cryptographic network protocol, Disponível em: http://en.wikipedia.org/wiki/Secure_Shell . Acesso em: agosto de 2014

TAN, J. et al. SALSA: analyzing logs as state machines. In: Workshop on Analysis of System Logs (WASL'08), 1., San Diego, 2008. **Anais eletrônicos...** Berkeley: USENIX, 2008. P. 1-8. Disponível em: <https://www.usenix.org/legacy/events/wasl08/tech/full_papers/tan/tan.pdf>

VAARANDI, R. SEC – a lightweight event correlation tool. In: Workshop on IP Operations & Management, 2., Dallas, 2002. **Anais eletrônicos...** Washington:

IEEE, 2002. P.111-115. DOI: 10.1109/IPOM.2002.1045765. Disponível em: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1045765>

_____. A data clustering algorithm for mining patterns from event logs. In: Workshop on,IP Operations & Management 3., Kansas, 2003. **Anais eletrônicos...** Washington: IEEE, 2003. P.119-126.
DOI: 10.1109/IPOM.2003.1251233. Disponível em: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1251233>

XSENS. Site do projeto Xsens. Disponível em: <http://www.xsens.com/>. Acesso em: janeiro de 2014.