Beatriz Alves Barros de Sousa

Formalização em Coq de execução de testes para *ioco*

Recife

2022

Beatriz Alves Barros de Sousa

Formalização em Coq de execução de testes para *ioco*

Trabalho apresentado como requisito parcial para a conclusão da graduação em Engenharia da Computação do Centro de Informática da Universidade Federal de Pernambuco.

Universidade Federal de Pernambuco – UFPE

Centro de Informática – CIn

Graduação em Engenharia da Computação

Orientador: Gustavo Henrique Porto de Carvalho

Recife

2022

Agradecimentos

Agradeço aos meus pais, Selma e Waldeni, e a minha irmã Bianca, por sempre terem acreditado em mim, me apoiarem e me incentivarem nos estudos. Agradeço ao professor Gustavo, por ter me passado tanto conhecimento sobre a área e pelo acompanhamento e dedicação durante as orientações deste trabalho. Agradeço aos meus amigos que fiz durante a graduação e a Nathália, por todos os conhecimentos compartilhados e momentos divertidos que deixavam meu dia a dia mais leve. Por fim, agradeço ao meu marido Lucas, não há palavras que expressem o quão grata sou por sua ajuda e suporte durante todos os momentos difíceis.

Abstract

The presence of software is increasing in the society, and, because of that, the concern about its quality and reliability has increased too, especially when it comes to critical systems. Although software testing is a crucial step in software production, it is typically associated with some risks and elevated costs, particularly, when carried out manually. Model-based testing (MBT) tools can change this scenario, enabling automatic test design and execution. However, assessing the correctness of these tools, which are software developed by humans, is still a valid concern. To alleviate this concern, formal testing approaches have been proposed. Continuing the researches of Sobral (2019) and Santana (2020), which formalised in Coq some concepts of the ioco formal testing theory, this work mechanises the test execution process proposed by Tretmans (2008).

Keywords: MBT, LTS, ioco, test case, test execution, Coq.

Resumo

Por softwares estarem cada vez mais presentes na sociedade, a preocupação acerca de sua qualidade e confiabilidade tem aumentado, principalmente quando se trata de sistemas críticos. Apesar de testes terem demonstrado ser uma etapa crucial da produção de um software, a forma como são criados e executados traz riscos e custos, em particular, por serem realizados de maneira manual. Técnicas de model-based testing (MBT) aparecem como uma forma de alterar esse contexto, proporcionando a criação e a execução de testes de forma automatizada. Porém, averiguar a corretude destas ferramentas, que são softwares desenvolvidos por humanos, também é um motivo de preocupação. Com a motivação de aliviar essa preocupação, surgem abordagens formais de geração e execução automática de testes, onde é possível provar a sua corretude e completude. Continuando os trabalhos de Sobral (2019) e Santana (2020), os quais formalizaram em Coq conceitos associados à teoria formal de testes *ioco*, este trabalho mecaniza o processo de execução de testes proposto por Tretmans (2008).

Palavras-chave: MBT, LTS, ioco, caso de teste, execução de testes, Coq.

Lista de ilustrações

Sumário

1 Introdução

Na sociedade atual, é nítida a influência de softwares na vida humana, estando presentes desde simples atividades cotidianas, até em atividades mais críticas, como sistemas hospitalares e de transporte. Nesse cenário, a realização de testes para encontrar possíveis falhas tem ganhado cada vez mais importância. Além de aumentar a qualidade do software, encontrar uma falha nas fases iniciais de desenvolvimento pode reduzir consideravelmente os custos de sua correção (RTI, 2002). Entretanto, testes, em sua maior parte, costumam ser realizados de forma manual.

Testes criados e executados de forma manual estão sujeitos a erros humanos. Para garantir uma maior qualidade e maior confiabilidade do software, ferramentas capazes de gerar e executar automaticamente testes surgem como uma opção mais efetiva para a etapa de verificação de software.

Nesse contexto, técnicas de Model-based Testing (MBT) vêm ganhando atenção. Esta técnica permite a criação de grandes quantidades de casos de teste de maneira automatizada utilizando como partida um modelo de especificação do sistema a ser testado. Contudo, a qualidade dessas ferramentas, que também são softwares feitos por humanos, vira um motivo de preocupação, uma vez que estão, similarmente, sujeitas a falhas que podem ocasionar a identificação de um erro inexistente (falso-positivo) ou a não identificação de um erro presente (falso-negativo).

Gaudel (1995) define, então, a noção de estratégias formais de testes, com o propósito de demonstrar que a geração e a execução de testes de maneira automatizada pode ser provada correta (*sound*). Desta forma, tem-se a garantia de que qualquer implementação que não passe nos testes de fato não está em conformidade com a especificação do sistema.

Em seu trabalho, Tretmans (2008) define de forma teórica uma estratégia formal de testes fundamentada na relação de conformidade *ioco*. Esta relação é capaz de distinguir ações de entrada de ações de saída, o que permite considerar testes de sistemas reativos de um forma mais realista. Além de ser considerada no próprio trabalho de Tretmans (2008), a relação *ioco* influenciou a definição de outras teorias formais de testes, como a proposta por Cavalcanti, Hierons e Nogueira (2020).

Com o intuito de obter uma formalização executável, Sobral (2019) iniciou a formalização da teoria de testes de Tretmans (2008), no assistente de provas Coq (BERTOT; CASTRAN, 2010), definindo sistemas de transições rotuladas (*Labelled Transition Systems* - LTS) e a partir disso a relação de conformidade *ioco*. Santana (2020), dando continuidade ao trabalho, definiu a linguagem de processos introduzida por Tretmans (2008), permitindo a obtenção automática de LTSs a partir de especificações textuais do comportamento

esperado do sistema em Coq.

Apesar dos resultados obtidos nos trabalhos anteriores caminharem na direção de criar uma formalização executável da teoria de Tretmans (2008), o processo de execução de testes ainda não foi contemplado. Logo, a pergunta de pesquisa deste trabalho é: como formalizar, em Coq, o processo de execução de testes definido por Tretmans (2008), integrando também os resultados obtidos previamente pelos trabalhos de Sobral (2019) e Santana (2020)?

1.1 Objetivos

O objetivo **geral** deste trabalho é dar continuidade ao trabalho de Sobral (2019) e Santana (2020), formalizando em Coq o processo de execução de testes baseado na relação de conformidade *ioco*. Para isso, serão consideradas as definições descritas por Tretmans (2008, p. 30-31).

1.2 Contribuições

Na teoria de testes proposta por Tretmans (2008), o comportamento esperado de um sistema é descrito de forma abstrata, utilizando a linguagem *Behaviour Expressions*. A formalização em Coq dessa linguagem foi realizada por Santana (2020).

Por exemplo, considere um sistema cuja especificação descreve o funcionamento de uma máquina de venda automática (VEND_MACHINE), onde, após apertar um botão (but), o usuário pode optar por comprar um chocolate (choc), um biscoito (bisc) ou um refrigerante (soda). Caso a opção seja um refrigerante, o usuário deve escolher entre um refrigerante comum (reg) ou um diet (diet). A seguir, tem-se a descrição (em Coq) deste comportamento na linguagem *Behaviour Expressions*.

```
Definition VEND_MACHINE :=
  "VEND_MACHINE"::=
    "but";; ("choc";; STOP [] "bisc";; STOP [] "soda";; ("diet";; STOP [] "reg";; STOP)).
```

Inicialmente, acontece o evento *but*, simbolizando o botão sendo apertado pelo usuário, seguido de possíveis escolhas de eventos, separadas pelo operador de escolha []. As escolhas podem ser *choc*, representando a compra de um chocolate, *bisc*, representando a compra de um biscoito, ou *soda*, representando a compra de um refrigerante. Após a escolha de *choc* ou de *bisc*, o sistema se comporta como o processo STOP, que caracteriza a sua finalização. Após a escolha de uma *soda*, uma outra escolha deve ser efetuada entre *diet* e comum (*reg*), e somente após isto o sistema deve finalizar.

Na teoria de testes baseada em *ioco*, uma descrição em *Behaviour Expressions*

é formalizada a partir da criação do LTS correspondente. A formalização em Coq de LTSs foi realizada por Sobral (2019). O trabalho corrente integra os resultados obtidos anteriormente e, com o apoio da ferramenta Graphviz (ELLSON et al., 2001), permite a geração de visualizações gráficas de LTSs. A Figura 1 apresenta graficamente o LTS obtido para o exemplo em questão. O estado inicial do LTS está destacado em vermelho.

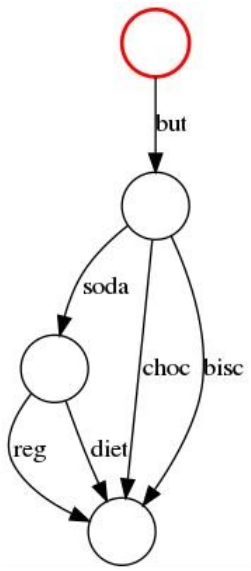


Figura 1 – Representação gráfica da especificação do sistema.

Ao integrar os trabalhos de Sobral (2019) e Santana (2020), é possível agora descrever a especificação de um sistema e uma implementação do mesmo através da linguagem *Behaviour Expressions* e, em seguida, demonstrar se o comportamento da implementação é conforme ao da especificação, considerando a relação de conformidade *ioco*. Para isto, modificações e ajustes foram realizados em definições propostas nestes trabalhos anteriores, com o objetivo de conseguir integrar os resultados previamente obtidos.

Adicionalmente, o trabalho corrente vai além e formaliza em Coq o conceito de casos de teste, assim como as regras de execução de um caso de teste. Informalmente, um caso de teste é a descrição do comportamento esperado para um cenário de uso do sistema a ser analisado. Executar este caso de teste no sistema gerará um veredito: se o sistema passa neste caso de teste ou se o sistema falha.

No contexto do exemplo anterior, um possível caso de teste seria verificar se, após a escolha do refrigerante, o usuário poderá escolher entre um refrigerante *diet* e um refrigerante comum (*reg*). Para isto, este caso de teste deve executar o sistema apertando o botão, escolhendo um refrigerante, e somente após isto, verificar se o sistema oferece as opções de escolha *diet* e *reg*. Se a implementação não oferecer estas duas opções, diz-se que a execução deste caso de teste falhou.

Casos de teste também são formalizados como LTSs, onde as transições com rótulos

de entrada representam escolhas enviadas do teste para a implementação e as transições com rótulos de saída representam o que o teste deve fazer quando é observada tal saída da implementação. A execução de um caso de teste é formalizada através da definição de regras para a execução em paralelo da implementação do sistema com o caso de teste.

A Figura 2 mostra, de maneira simplificada, como seria um caso de teste que captura o comportamento descrito anteriormente. Nela, o símbolo θ representa a situação em que a implementação não gerou uma resposta.

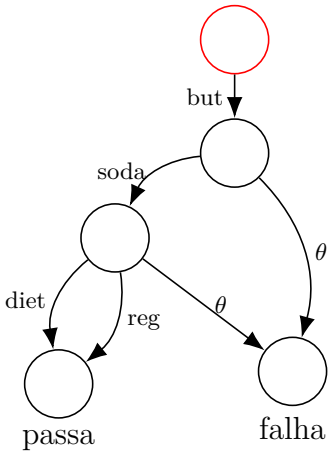


Figura 2 – Exemplo simplificado de um caso de teste.

Logo, as principais contribuições deste trabalho são:

- Reformulação de formalizações feitas por Sobral (2019) que estavam incompletas ou que foram impactadas pelas definições criadas no trabalho de Santana (2020);
- Formalização em Coq de casos de teste;
- Formalização em Coq das regras de execução de um caso de teste;
- Definição de táticas que auxiliam nas provas efetuadas durante esta pesquisa;
- Geração de visualizações gráficas de LTSs com o apoio do Graphviz.

1.3 Estrutura do texto

Este trabalho possui três capítulos, além deste introdutório. O Capítulo 2 apresenta o assistente de provas Coq, abordando os principais mecanismos utilizados neste trabalho. Adicionalmente, este capítulo descreve sucintamente os conceitos e estruturas definidos no trabalho de Tretmans (2008), necessários por serem o ponto de partida desta pesquisa. O Capítulo 3 contempla o processo de integração dos trabalhos anteriores de Sobral (2019) e Santana (2020), a formalização de casos de teste e das regras de execução de um caso de teste. Adicionalmente, o capítulo detalha a utilização do Graphviz como suporte para

visualização gráfica das estruturas definidas. Por fim, o Capítulo 4 apresenta as conclusões deste trabalho e aborda possíveis pontos para trabalhos futuros.

2 Conceitos Básicos

Este capítulo cobre conceitos necessários ao entendimento do trabalho desenvolvido nesta pesquisa. Enquanto a Seção 2.1 apresenta os fundamentos do assistente de provas Coq, a Seção 2.2 descreve os principais conceitos associados à teoria formal de testes proposta por Tretmans (2008).

2.1 Coq

Coq é um assistente de provas desenvolvido com o objetivo de auxiliar na criação de definições formais, algoritmos e asserções matemáticas. Ele foi produzido baseado na teoria de Cálculo de Construções Indutivas (COQUAND; HUET, 1988) e fornece Gallina como uma linguagem de programação funcional nativa. Um fator importante é que toda definição computável válida em Gallina precisa terminar. Desta forma, a linguagem evita laços infinitos e o problema da parada. Além de Gallina, Coq possui um conjunto de táticas pré-definidas que auxiliam na construção de provas, além de uma linguagem para a definição de novas táticas (Ltac). As próximas seções apresentam mais detalhes de Coq e suas possibilidades de uso.

2.1.1 Tipos, funções e registros

Gallina oferece em sua biblioteca padrão definições de tipos, como booleanos, números e outras estruturas de dados comuns. Além disso, nessa linguagem é possível definir novos tipos. Para definir um novo tipo em Coq, utiliza-se a palavra-chave *Inductive*. Os tipos são caracterizados através de construtores, estes podendo ou não receber argumentos. O exemplo abaixo mostra a definição do tipo *rgb*, que descreve as cores primárias. O novo tipo possui três construtores que não recebem argumentos.

```
Inductive rgb : Type :=
  | red
  | green
  | blue.
```

O exemplo a seguir mostra a definição de uma lista de naturais com dois construtores: *nil*, representando uma lista vazia, e *cons*, representando uma lista que possui como primeiro elemento o natural n seguido de uma lista de naturais l.

```
Inductive nat_list : Type :=
  | nil
  | cons (n : nat) (l : nat_list).
```

Há duas formas de definir funções em Coq. Funções não recursivas são definidas com a palavra-chave *Definition* e podem ter aridade maior ou igual a zero. Casamento de padrões é frequentemente utilizado para descrever o corpo de uma função. Este é introduzido com a palavra-chave *match* e, quando utilizado, Gallina exige que todos os padrões possíveis sejam representados. O trecho de código a seguir exemplifica o uso de casamento de padrões na função não recursiva *tl*. Esta função computa o *tail* de uma lista, ou seja, a lista sem seu primeiro elemento.

```
Definition tl (l : nat_list) : nat_list :=
  match l with
  | nil ⇒ nil
  | cons h t ⇒ t
  end.
```

Funções recursivas são normalmente definidas com a palavra-chave *Fixpoint*. Coq exige que as funções recursivas possuam algum argumento que sempre decresça a cada recursão, de forma a garantir que, para todas as entradas possíveis, a função sempre termine. A função *app* descrita abaixo é um exemplo de uma função recursiva que computa a concatenação de duas listas de naturais. Nota-se que o primeiro argumento sempre decresce em tamanho a cada chamada recursiva da função.

```
Fixpoint app (l1 l2 : nat_list) : nat_list :=
  match l1 with
  | nil ⇒ l2
  | cons h t ⇒ cons h (app t l2)
  end.
```

É possível também descrever proposições lógicas em Coq. Essas proposições são representadas pelo tipo *Prop* e podem ser construídas usando os operadores lógicos clássicos, já definidos na biblioteca padrão. Os operadores lógicos de conjunção (representando por $\wedge$), disjunção (representado por $\vee$), implicação (representado por $\rightarrow$) e negação (representado por $\sim$) são alguns dos operadores disponíveis.

O trecho de código a seguir mostra a função *nat_is_even*, que retorna uma proposição lógica caracterizando a paridade de um número n a partir da existência de um natural k tal que $n = 2 * k$.

```
Definition nat_is_even (n : nat) : Prop :=
  ∃ (k : nat), n = 2 × k.
```

Proposições lógicas também podem ser definidas de maneira indutiva, utilizando a palavra-chave *Inductive*. Neste caso, os construtores determinam regras para a construção de proposições válidas. Por exemplo, o código a seguir mostra a definição *Forall*, que relaciona uma lista de naturais (*nat_list*) e uma propriedade sobre naturais (*nat* → `Prop`).

Dada uma proposição P e uma lista l, a proposição *Forall P l* é verdadeira caso todos os elementos da lista l satisfaçam a propriedade P.

```
Inductive Forall : (nat → Prop) → nat_list → Prop :=
  | Forall_nil (P : nat → Prop) : Forall P nil
  | Forall_cons (P : nat → Prop) (n : nat) (l : nat_list) :
        P n →
        Forall P l →
        Forall P (cons n l).
```

O construtor *Forall_nil* define o caso base de *Forall*, onde esta é verdade para uma lista vazia, independende da propriedade P. O construtor *Forall_cons* define o caso indutivo. Ele propõe que, se a propriedade P é válida para um natural n, e *Forall P l* é verdade para uma lista de naturais l, então *Forall P (cons n l)* também será verdade.

Coq também permite a criação de registros. A palavra-chave *Record* permite a criação de tipos que agrupam dados (variáveis/atributos) e propriedades. Para criar instâncias dos tipos definidos com esta palavra-chave, é preciso prover valores para as variáveis em conjunto com as provas das propriedades definidas. O trecho de código a seguir mostra um exemplo de uso de *Record*, com o tipo *even_nat_list*.

```
Record even_nat_list : Type := mkEvenNatList {
    l : nat_list ;
    all_nat_in_l_are_even : Forall nat_is_even l
}.
```

A definição *even_nat_list* possui a lista de naturais (l) como seu único atributo e possui uma propriedade (*all_nat_in_l_are_even*) que determina que a lista l possui apenas naturais pares. Um exemplo de como criar uma instância deste tipo será apresentado na Seção 2.1.2.

2.1.2 Teoremas e provas

Sendo um assistente de provas, Coq permite a criação de teoremas e lemas para enunciar proposições e dispõe de táticas que auxiliam na prova da veracidade das mesmas. Um teorema é introduzido através da palavra-chave *Theorem*, seguido de um nome e de uma *Prop* que o caracteriza. Logo após a descrição do teorema, a palavra-chave *Proof* demarca o início da prova. O final da prova é normalmente demarcado pela palavra-chave *Qed*. Algumas das táticas disponibilizadas pela biblioteca padrão para ajudar na construção de provas são listadas a seguir.

- **reflexivity**: utilizada quando há uma igualdade no objetivo de prova e é possível provar automaticamente que ambos os lados são iguais.

- **simpl**: simplifica uma expressão, reescrevendo a aplicação de funções pelo respectivo resultado, quando possível.
- **intros**: move variáveis quantificadas universalmente (instanciação universal) e hipóteses do objetivo de prova para a área de suposições do contexto de prova atual.
- **apply**: pode ser utilizada para aplicar uma hipótese ou teorema tanto no objetivo de prova quanto em outra hipótese. Quando utilizada em hipóteses, a tática aplicará a regra de *modus ponens*. Quando utilizada no objetivo de prova, a tática criará novos objetivos para cada uma das premissas da hipótese que está sendo aplicada, concluindo a prova caso não existam premissas.
- **exists**: usado em casos onde o objetivo de prova possui o quantificador existencial. Substitui a variável que está quantificada existencialmente por valor específico estabelecido pelo usuário.
- **rewrite**: usado para reescrever expressões de acordo com alguma hipótese do contexto ou teorema.
- **destruct**: destrói um termo elencando todas as formas de construir o termo destruído. Para cada possível construtor, um novo subobjetivo é criado.
- **induction**: utilizado em tipos indutivos para destruir um termo de maneira semelhante ao *destruct*, mas introduzindo uma hipótese indutiva para cada referência recursiva feita pelo construtor elencado no subobjetivo.

O exemplo abaixo mostra o teorema *app_nil_r*, onde é proposto que, para toda lista de naturais *l*, a concatenação pela direita com uma lista vazia resulta na própria lista *l*. A prova deste teorema é feita por indução e utiliza algumas das táticas descritas acima. Símbolos como -, + e × são utilizados para delimitar o aninhamento de objetivos de prova. Ferramentas como CoqIDE[1], permitem visualizar interativamente como cada tática modifica o objetivo de prova.

```
Theorem app_nil_r :
  ∀ (l: nat_list), app l nil = l.
Proof.
  intros l. induction l as [| h t IH].
  - reflexivity.
  - simpl. rewrite IH. reflexivity.
Qed.
```

Ao declarar instâncias de elementos de tipos criados com a palavra-chave *Record*, normalmente é preciso apresentar provas de que as propriedades descritas são válidas para

[1] https://coq.inria.fr/refman/practical-tools/coqide.html

os valores passados. Por exemplo, o trecho de código a seguir mostra a instanciação do tipo *even_nat_list*. É importante notar que neste exemplo foi preciso provar que todos os elementos da lista de naturais passada são pares.

```
Definition even_nat_list_ex : even_nat_list.
Proof.
  apply (mkEvenNatList (cons 2 (cons 4 (cons 6 nil)))).
  apply Forall_cons.
  - exists 1. reflexivity.
  - apply Forall_cons.
    + exists 2. reflexivity.
    + apply Forall_cons.
      × exists 3. reflexivity.
      × apply Forall_nil.
Defined.
```

Observe que a prova anterior termina com `Defined` e não `Qed`. Deve-se usar a palavra-chave `Defined` quando se deseja que a prova não seja opaca. No caso acima, isto é necessário para que seja possível, posteriormente, acessar os valores do registro em questão. Como esses valores (2, 4 e 6) são providos dentro do corpo da prova, caso a prova fosse opaca, não seria possível acessá-los, após a conclusão da prova.

2.1.3 Automação de provas e Ltac

Além das táticas anteriormente apresentadas, Coq também possui recursos de automação de provas. Esses recursos, apesar de se assemelharem com as táticas comuns, podem ajudar a encurtar o tamanho de uma prova. Por exemplo, a tática de automação *auto* resolve objetivos que são solucionáveis a partir de qualquer combinação de *intros* e *apply*, limitada a uma quantidade pré-definida ou informada pelo usuário de *apply*.

O trecho de código a seguir prova o lema *transitivity* de duas maneiras diferentes. O primeiro utiliza as táticas comuns enquanto o segundo utiliza apenas a tática *auto*.

```
Lemma transitivity :
  ∀ (P Q R : Prop), (P → Q) → (Q → R) → P → R.
Proof.
  intros P Q R PQ QR PT.
  apply QR. apply PQ. apply PT.
Qed.

Lemma transitivity' :
  ∀ P Q R : Prop, (P → Q) → (Q → R) → P → R.
Proof.
```

```
  auto.
Qed.
```

Um fator interessante é que esta tática não gera uma falha, caso não seja possível terminar o objetivo de prova, apenas nada se altera.

Outro artifício utilizado em provas é escrever uma sequência de táticas separadas por ponto e vírgula (;). Neste caso, se uma das táticas gera mais que um subobjetivo de prova, as táticas subsequentes serão aplicadas em todos os subobjetivos. Se a tática resolve um subobjetivo, este é automaticamente removido. O trecho de código a seguir exemplifica o uso deste operador.

```
Lemma app_suffix_eq:
  ∀ (a b c : nat_list), app a b = app a c → b = c.
Proof.
  intros a b c H.
  induction a as [| n a' IHa]; simpl in H; inversion H; auto.
Qed.
```

Neste exemplo a tática *induction* gera dois subobjetivos de prova, ambos são resolvidos através da mesma sequência (*simpl in H; inversion H; auto.*). O uso do operador ';' permite que a prova seja concluída sem a necessidade de repetir esta sequência de comandos duas vezes.

Uma forma ainda mais poderosa no auxílio de provas de teoremas é criando uma tática própria e personalizada. Coq oferece esse artifício através de uma linguagem integrada chamada *Ltac*. Usar os recursos de automação, juntamente a táticas definidas pelo usuário em *Ltac*, pode deixar as provas ainda mais curtas. Outra vantagem da combinação desses mecanismos é que, se usados corretamente, podem tornar as provas mais robustas (gerais) e resistentes a mudanças nas definições subjacentes.

Para criar uma tática personalizada, utiliza-se a palavra-chave *Ltac*, seguido de um nome e um conjunto de argumentos necessários para a tática. O corpo da nova tática pode ser construído a partir de combinações de táticas existentes separadas por ';'. Outra possibilidade é usar casamento de padrões. O exemplo abaixo mostra a definição de *split_Forall*, que possui o objetivo de ajudar na prova de proposições do tipo *Forall P l*. Esta tática cria um objetivo de prova para cada elemento da lista *l*.

```
Ltac split_Forall :=
  try (apply Forall_nil); apply Forall_cons; try split_Forall.
```

O comando *try*, utilizado nesta tática, tenta executar a tática passada como argumento (neste caso, *apply Forall_nil*). Caso falhe, o resultado é ignorado e a prova volta ao estado original (antes da aplicação da tática).

A construção desta nova tática auxilia, por exemplo, na prova do teorema *all_ones* mostrado a seguir. Ele postula que todos os elementos da lista possuem valor igual a 1. Com a aplicação de *split_Forall*, um subobjetivo para cada elemento da lista é criado, e como todos os elementos de fato são 1, reflexivity conclui a prova.

```
Theorem all_ones :
  Forall (fun x ⇒ x = 1) (cons 1 (cons 1 (cons 1 nil))).
Proof.
  split_Forall; reflexivity.
Qed.
```

Como outro exemplo de uso desta tática, o trecho de código a seguir exemplifica a instanciação de um elemento do tipo *even_nat_list*, descrito no final da Seção 2.1.1. Pode-se notar que esta nova definição se torna mais clara e sucinta, quando comparada à definição semelhante feita no final da Seção 2.1.2.

```
Definition even_nat_list_ex' : even_nat_list.
Proof.
  apply (mkEvenNatList (cons 2 (cons 4 (cons 6 nil)))).
  split_Forall.
  - exists 1. reflexivity.
  - exists 2. reflexivity.
  - exists 3. reflexivity.
Defined.
```

2.2 Testes baseados em modelos

Testes baseados em modelos é uma técnica de teste de software em que testes são gerados a partir de modelos que descrevem o comportamento esperado do sistema. Estes testes são então utilizados para verificar se uma implementação em teste (em inglês *implementation under test* – IUT) satisfaz o comportamento esperado. Esses comportamentos podem ser descritos em termos de uma sequência de entradas, ações, saídas, estados e outros aspectos.

Essa técnica tem ganhado destaque por proporcionar automação da geração e da execução de testes. Contudo, como explicado anteriormente, mesmo se o modelo for criado corretamente, expressando precisamente o comportamento esperado do sistema, tipicamente, não existem garantias de que o processo de geração e de execução são corretos. Teorias formais de teste buscam aliviar este problema, propondo estratégias de MBT onde se busca provar a corretude da geração e da execução de testes.

Há diversos tipos de MBT, a depender de alguns fatores, como o tipo de modelo

usado e o grau de acessibilidade e observabilidade do sistema a ser testado. Este trabalho segue o modelo descrito por Tretmans (2008), e considera-se a estratégia de MBT como:

- **Formal**: o modelo que descreve o comportamento da IUT é dado em alguma linguagem com sintaxe e semântica formalmente definidas.
- **Ativa**: o teste observa de forma ativa a execução, criando estímulos para a IUT e analisando as respostas recebidas.
- **Caixa-preta**: a IUT é vista como uma caixa preta. Os detalhes internos não são observados, apenas as interfaces externas.
- **Teste de funcionalidade**: envolve a verificação de propriedades relacionadas a funcionalidades da IUT, examinando se o sistema faz corretamente o que deveria ser feito.

2.2.1 Labelled transition systems

Em seu artigo, Tretmans (2008) usa *Labelled Transition Systems* (LTSs) para descrever os modelos e as IUTs. Um LTS é definido como uma estrutura formada por estados e transições rotuladas. Os estados representam os possíveis estados do sistema e as transições rotuladas descrevem ações que o sistema pode executar a partir daquele estado.

Uma quádrupla (Q, L, T, q_0) é usada para representar um LTS; Q representa o conjunto enumerável e não nulo de estados do sistema; L é o conjunto enumerável de rótulos; T é o conjunto de transições dado por $T \subseteq Q \times (L \cup \{\tau\}) \times Q$, onde $\tau \notin L$; e q_0 é o estado inicial, onde $q_0 \in Q$. É importante destacar que τ é um rótulo especial para caracterizar as ações internas do sistema, ou seja, ações que não são observáveis a partir de uma interface externa.

A Figura 3 mostra um exemplo de LTS, nomeado de r, representado através de um grafo. Neste exemplo, uma seta destaca o estado inicial r_0. O conjunto de estados Q é $\{r_0, r_1, r_2, r_3, r_4, r_5\}$, o conjunto de rótulos L é $\{but, liq, choc\}$ e o conjunto de transições T é $\{(r_0, but, r_1), (r_0, but, r_2), (r_1, liq, r_3), (r_2, but, r_4), (r_4, choc, r_5)\}$.

Ainda sobre LTSs, Tretmans (2008) apresenta algumas definições necessárias à definição da sua teoria de testes. Dentre estas, destaca-se o conceito de alcançabilidade a partir de uma transição e de uma sequência de transições.

(1) $q \xrightarrow{\mu} q' \Leftrightarrow_{def} (q, \mu, q') \in T$

(2) $q \xrightarrow{\mu_1 \dots \mu_n} q' \Leftrightarrow_{def} \exists q_0 \dots q_n : q = q_0 \xrightarrow{\mu_1} q_1 \xrightarrow{\mu_2} \dots \xrightarrow{\mu_n} q_n = q'$

Ao escrever $q \xrightarrow{\mu} q'$, tem-se que o estado q' é alcançável a partir de q realizando a transição rotulada por μ. De forma análoga, $q \xrightarrow{\mu_1 \dots \mu_n} q'$ denota que o estado q' é alcançável

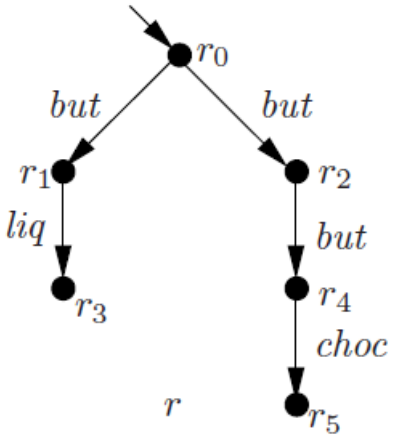


Figura 3 – LTS r definido por Tretmans (2008, Fig. 2, p. 8).

a partir de q considerando uma certa sequência de rótulos. Por exemplo, considerando o LTS da Figura 3, pode-se dizer que $r_0 \xrightarrow{but} r_1$ e que $r_0 \xrightarrow{but \cdot liq} r_3$.

É importante ressaltar também o conceito de alcançabilidade ignorando as transições internas (τ) do sistema, denotado por $q \Rightarrow$. Este conceito é particularmente importante pois, quando se faz a análise do sistema pela perspectiva de testes, não é possível observar as transições internas.

(3) $q \stackrel{\epsilon}{\Rightarrow} q' \Leftrightarrow_{def} q = q' \ or \ q \xrightarrow{\tau \ldots \tau} q'$

(4) $q \stackrel{a}{\Rightarrow} q' \Leftrightarrow_{def} \exists q_1, q_2 : q \stackrel{\epsilon}{\Rightarrow} q1 \xrightarrow{a} q2 \stackrel{\epsilon}{\Rightarrow} q'$

(5) $q \stackrel{a_1 \ldots a_n}{\Longrightarrow} q' \Leftrightarrow_{def} \exists q_0 \ldots q_n : q = q_0 \stackrel{a_1}{\Longrightarrow} q_1 \stackrel{a_2}{\Longrightarrow} \ldots \stackrel{a_n}{\Longrightarrow} q_n = q'$

(6) $q \stackrel{\sigma}{\Longrightarrow} \Leftrightarrow_{def} \exists q' : q \stackrel{\sigma}{\Longrightarrow} q'$

Com estas definições, é possível formalizar os conceitos de *init*, *trace* e *after*, onde $q \in Q$ e $\sigma \in L^*$. O primeiro conceito caracteriza o conjunto de rótulos que, a partir de um dado estado q, leva a algum outro estado do LTS. O segundo descreve o caminho percorrido a partir de um estado, denotando a sequência de ações (rótulos) que foram tomadas. O último designa o conjunto de estados alcançáveis a partir de um estado (ou conjunto de estados) considerando um trace σ.

(7) $init(q) =_{def} \{ \mu \in L \cup \{\tau\} \mid \exists q' : \ q \xrightarrow{\mu} q' \}$

(8) $traces(q) =_{def} \{ \sigma \in L^* \mid q \stackrel{\sigma}{\Longrightarrow} \}$

(9) q **after** $\sigma =_{def} \{ q' \in Q \mid q \stackrel{\sigma}{\Longrightarrow} q' \}$

(10) Q' **after** $\sigma =_{def} \bigcup \{ q$ **after** $\sigma \mid \forall q \in Q' \}$, onde $Q' \subseteq Q$

Retomando o exemplo do LTS da Figura 3, todas as sequências possíveis de rótulos são $traces(r) = \{\epsilon,\ but,\ but \cdot liq,\ but \cdot but,\ but \cdot but \cdot choc\}$. Já o conjunto de rótulos que, a partir do estado r_0, alcança algum outro estado é dado por $init(r_0) = \{but\}$. O conjunto de estados alcançáveis após percorrer o *trace* but, a partir do estado r_0, é r_0 **after** $but = \{r_1, r_2\}$.

A maioria das definições apresentadas anteriormente (exceto a Definição 10), assim como outras, se aplicam tanto a um estado arbitrário q como a um LTS $p = (Q, L, T, q_0)$. Por exemplo, ao escrever $p \stackrel{\sigma}{\Rightarrow}$, entende-se $q_0 \stackrel{\sigma}{\Rightarrow}$.

Apesar dos rótulos das transições explicitarem as ações possíveis de serem realizadas no sistema, as definições acima abstraem se as ações representam entradas ou saídas do sistema. Para poder distinguir os rótulos entre entradas e saídas, Tretmans (2008) define uma nova classe de LTSs, LTSs com entradas e saídas, onde o conjunto de rótulos L é dividido em dois novos conjuntos: L_I, simbolizando as ações de entrada; e L_U, simbolizando as ações de saída. Esta nova classe é representada pela quíntupla (Q, L_I, L_U, T, q_0), em que $L_I \cap L_U = \emptyset$ e é equivalente a um LTS (Q, $L_I \cup L_U$, T, q_0). Para facilitar a visualização, os rótulos de entradas são prefixados com um "?" e os de saída com "!".

Apesar desta nomenclatura não ser utilizada no trabalho de Tretmans (2008), aqui se denomina um LTS com entradas e saídas como um IOLTS. É importante não confundir este termo com um IOTS, explicado posteriormente. Este último, é uma sub-classe de IOLTS conhecida por ser *input-enabled*.

Um exemplo de um LTS com entradas e saídas é mostrado na Figura 4. Este exemplo consiste do LTS da Figura 3, considerando $L_I = \{but\}$ (rótulos de entrada) e $L_U = \{liq, choc\}$ (rótulos de saída).

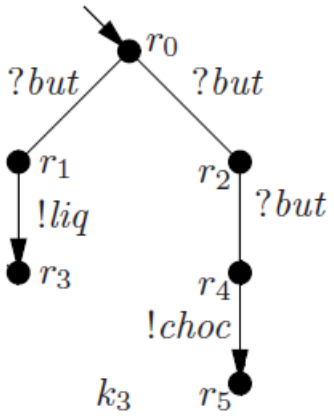


Figura 4 – Exemplo de LTS com entradas e saídas.

Para melhor representar a realidade, Tretmans (2008) define a classe *Input-Output Transition Systems* (IOTS), onde as saídas são as ações iniciadas pelo sistema e nunca

recusadas pelo ambiente. De forma análoga, as entradas são as ações iniciadas pelo ambiente e nunca recusadas pelo sistema. Consequentemente, um IOTS é um LTS com entradas e saídas, onde, para todos os estados alcançáveis do sistema, existem transições que partem destes estados com cada um dos rótulos de L_I. De maneira formal, seja $p = (Q, L_I, L_U, T, q_0)$ um LTS com entradas e saídas e $L = L_I \cup L_U$, p é um IOTS se a seguinte propriedade for satisfeita.

(11) $\forall q \in der(p),\ \forall x \in L_I : q \overset{x}{\Rightarrow}$

Considera-se *der* definido como abaixo. Lembre que $der(p)$ denota $der(q_0)$.

(12) $der(q) =_{def} \{\, q \in Q \mid \exists \sigma \in L^* : q \overset{\sigma}{\Rightarrow} q\}$

Por exemplo, o LTS com entradas e saídas da Figura 4 não é um IOTS por não ser *input-enabled*. A Figura 5 mostra a reestruturação deste LTS, adicionando as transições necessárias para que passe a ser um IOTS.

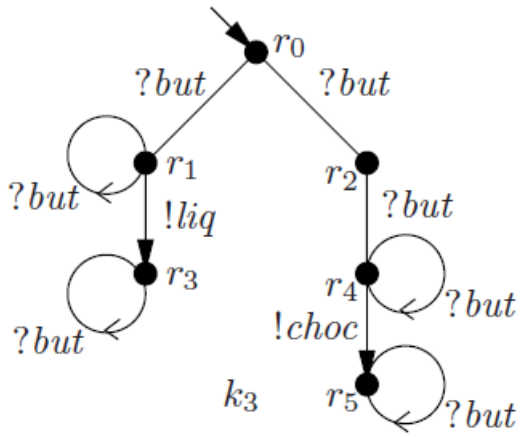


Figura 5 – IOTS $k3$ definido por Tretmans (2008, Fig. 4, p.14).

Há casos onde os estados do sistema executam nenhuma ação de saída. Nesses casos, a ausência de resposta por parte do sistema é observada pelo ambiente através de uma ação de saída especial denominada quiescência (δ). Um estado sem ações de saída é chamado de quiescente ($\delta(q)$) e é caracterizado pela definição a seguir.

(13) $\delta(q)$ se $\forall \mu \in L_U \cup \{\tau\},\ \nexists q' \in Q : q \overset{\mu}{\rightarrow} q'$

Por fim, Tretmans (2008) considera essa informação para definir uma nova classe de LTSs, na qual são adicionadas as ações que representam a observação da ausência de saídas. Estas observações são representadas por transições δ e fazem parte de T_δ. Formalmente, seja $p = (Q, L_I, L_U, T, q_0)$ um LTS com entradas e saídas, este LTS anotado com a informação de quiescência é denotado por p_δ.

(14) $p_\delta =_{def} (Q, L_I, L_U \cup \{\delta\}, T \cup T_\delta, q_0)$, onde $T_\delta =_{def} \{\, (q, \delta, q) \mid q \in Q, \delta(q)\}$

Um *trace* nesta nova classe é chamado de *Strace* e considera as transições δ. Seja $L_\delta =_{def} L_I \cup L_U \cup \{\delta\}$, $Straces(p)$ é definido da seguinte forma.

(15) $Straces(p) =_{def} \{\, \sigma \in L_\delta^* \mid p_\delta \stackrel{\sigma}{\Rightarrow} \}$

Usa-se a letra S como alusão ao termo trace "suspenso". Um estado quiescente é também conhecido como um estado onde o sistema se encontra suspenso (não é possível observar ele em operação, produzindo uma saída). A Figura 6 mostra como seria o LTS da Figura 4 introduzindo as transições δ.

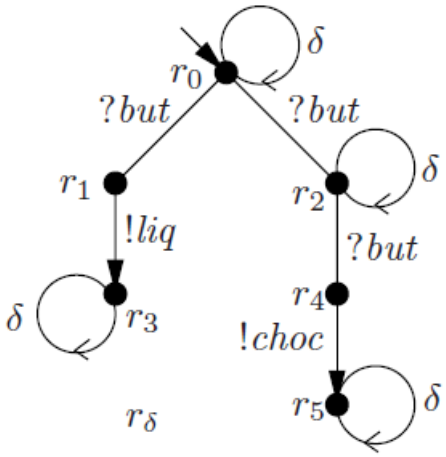


Figura 6 – IOLTS r_δ definido por Tretmans (2008, Fig. 6, p.17).

2.2.2 Representando um LTS através de uma linguagem de processos

Apesar de um LTS ser um modelo semântico poderoso para representar implementações e especificações de sistemas, sistemas reais podem conter milhares de estados, tornando inviável a sua representação gráfica ou pela quádrupla (Q, L, T, q_0). Buscando uma forma de contornar este problema, Tretmans (2008) introduz a linguagem de processos chamada *Behaviour Expressions*.

Cada expressão feita nesta linguagem caracteriza um LTS. Assim, é possível especificar sistemas mais complexos usando operadores da linguagem que combinam sistemas (expressões) mais simples.

A Figura 7 mostra a gramática desta linguagem, onde a é um rótulo tal que $a \in L$, B é uma expressão da linguagem, $\mathcal{B}$ é o conjunto enumerável de expressões da linguagem, $G \subseteq L$ é um conjunto de rótulos e P é o nome de um processo.

Na linguagem, $a; B$ caracteriza a operação prefixo. Assim, $a; B$ define um LTS que realiza a ação do rótulo a para então se comportar como o LTS definido por B. De maneira semelhante é definido $i; B$, porém com i representando o rótulo de transição interna τ.

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

Figura 7 – Sintaxe da linguagem de processos *Behaviour Expressions*.

A expressão $\Sigma\,\mathcal{B}$ caracteriza a operação de escolha, onde o sistema deve se comportar como um dos processos do conjunto $\mathcal{B}$. Outra maneira de apresentar uma escolha é através do operador $[]$, onde $B1[]B2$ é equivalente à $\sum\{B1, B2\}$. A expressão **stop** representa $\sum\emptyset$, ou seja, um processo em que nenhuma ação pode ser realizada (*deadlock*).

A expressão $B|[G]|B$ caracteriza a execução em paralelo de dois processos. Nesta execução, todas as ações contidas em G devem sincronizar, enquanto as demais ações podem ocorrer de maneira independente nos processos.

A expressão **hide** G **in** B caracteriza o LTS B quando todos os rótulos em G forem substituídos por τ. Por fim, para associar um nome a um comportamento específico, define-se um processo $P ::= B$. O nome atribuído, P no exemplo, pode ser utilizado para referenciar este processo, ao definir outros processos, com a intenção de descrever comportamentos mais complexos a partir de outros mais simples. Por exemplo, o LTS da Figura 3 pode ser descrito na linguagem *Behaviour Expressions* da seguinte forma: $r :=$ *but* ; *liq* ; **stop** [] *but* ; *but* ; *choc* ; **stop**.

A geração do LTS correspondente a uma expressão se dá a partir da semântica operacional dos operadores da linguagem *Behaviour Expressions*. Considerando que esta informação não é explorada diretamente por este trabalho, mas sim por Santana (2020), a semântica operacional dos operadores acima apresentados não será descrita aqui.

2.2.3 Relação de conformidade **ioco**

Há várias maneiras de definir que uma implementação está correta em relação a uma especificação. Tretmans (2008) baseou sua teoria na relação de conformidade **ioco** (*input-output-conformance*). O trabalho assume que as implementações a serem testadas sempre podem ser representadas por um IOTS, que possuem conjuntos de entrada e saída iguais aos da especificação.

Informalmente, para uma implementação i e uma especificação s, diz-se que i **ioco** s se, e somente se, para todo caminho possível de s, executá-lo em i gera uma saída que está prevista em s. É importante ressaltar que a inexistência de saídas é modelada como uma saída especial da implementação. Isto quer dizer que, se a quiescência é observada em i, então a quiescência deveria ser prevista em s.

Para poder definir em termos precisos a relação de conformidade **ioco**, é necessário primeiro formalizar o conceito *out*. Este representa o conjunto de rótulos de saída das transições que partem de um estado ou de um conjunto de estados.

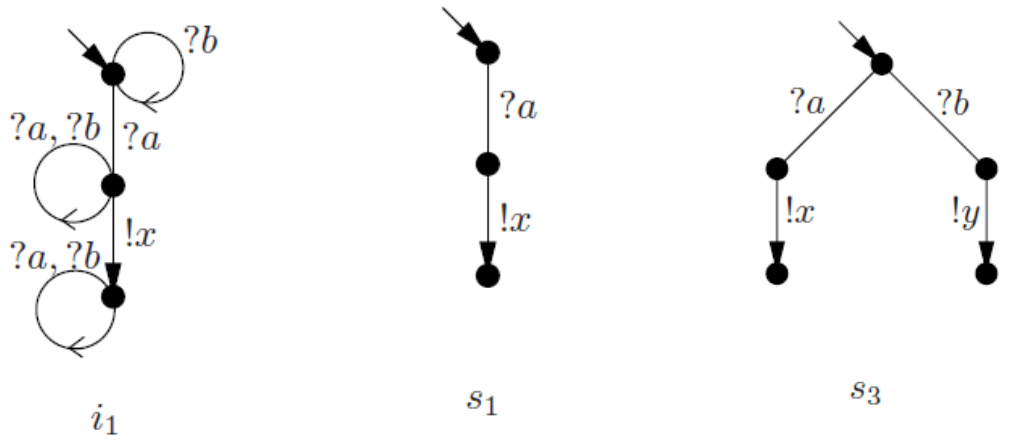
(16) $out(q) =_{def} \{\, x \in L_U \mid q \xrightarrow{x} \,\} \cup \{\, \delta \mid \delta(q) \,\}$

(17) $out(Q') =_{def} \bigcup \{\, out(q) \mid \forall q \in Q' \,\},$ onde $Q' \subseteq Q$

Desta forma, define-se que uma implementação i é conforme a uma especificação s se, e somente se, para todo trace σ de s (anotado com quiescência), as saídas da implementação i após realizar σ é um subconjunto das saídas de s após realizar σ.

(18) i **ioco** $s \Leftrightarrow_{def} \forall \sigma \in Straces(s) : out(i \textbf{ after } \sigma) \subseteq out(s \textbf{ after } \sigma)$

Para ilustrar os conceitos apresentados nesta seção, considere a implementação $i1$ e as especificações $s1$ e $s3$ mostradas na Figura 8.



(a) (b) (c)

Figura 8 – Implementação e especificações definidas por Tretmans (2008, p.22).

Seja $\delta^* = \{\epsilon, \delta, \delta \cdot \delta, \delta \cdots \delta\}$ o conjunto de todos Straces compostos apenas por δ, e, para A e B dois conjuntos compostos por Straces, $A \cdot B = \{a \cdot b \mid a \in A,\ b \in B\}$ o conjunto resultante da concatenação de todos os pares de *Straces* dos dois conjuntos. Então, o conjunto de Straces de $s1$ pode ser descrito como $Straces(s1) = \delta^* \cup \delta^* \cdot \{a\} \cup \delta^* \cdot \{a \cdot x\} \cdot \delta^*$.

Se σ é um *Strace* de $s1$ ($\sigma \in Straces(s1)$), é possível dizer que σ está em um dos três conjuntos que compõem *Straces(s1)*, ou seja, $\sigma \in \delta^* \ \vee\ \sigma \in \delta^* \cdot \{a\} \ \vee\ \sigma \in \delta^* \cdot \{a \cdot x\} \cdot \delta^*$. Se σ estiver presente no primeiro conjunto ($\sigma \in \delta^*$), então $out(i1 \textbf{ after } \sigma) = \{\delta\}$ e $out(s1 \textbf{ after } \sigma) = \{\delta\}$. Se σ estiver presente no segundo conjunto ($\sigma \in \delta^* \cdot \{a\}$), então $out(i1 \textbf{ after } \sigma) = \{x\}$ e $out(s1 \textbf{ after } \sigma) = \{x\}$. Por fim, se σ estiver presente no terceiro conjunto ($\sigma \in \delta^* \cdot \{a \cdot x\} \cdot \delta^*$), então $out(i1 \textbf{ after } \sigma) = \{\delta\}$ e $out(s1 \textbf{ after } \sigma) = \{\delta\}$. Portanto, em todos os casos, $out(i1 \textbf{ after } \sigma) \subseteq out(s1 \textbf{ after } \sigma)$, sendo assim é possível afirmar que $i1$ **ioco** $s1$. Em contrapartida, $i1$ não é **ioco**-comforme com $s3$ ($i1$ $\neg$**ioco** $s3$), pois, para o *Strace* composto apenas pelo rótulo b, $out(i1 \textbf{ after } b) = \{\delta\}$, enquanto $out(s3 \textbf{ after } b) = \{y\}$ e $\{\delta\} \not\subseteq \{y\}$.

2.2.4 Casos de teste

Tretmans (2008), em seu artigo, caracteriza um caso de teste como uma especificação que descreve o comportamento de um testador operando um experimento sobre uma implementação. Neste experimento, o testador representa um ambiente artificial da implementação. Considerando o que foi apresentado anteriormente, as implementações podem: aceitar qualquer entrada de L_I; produzir alguma saída L_U; ou permanecer quiescente. Desta forma, um testador deve ser capaz de prover essas entradas, observar as saídas resultantes destas entradas e observar a quiescência, caso não haja saídas. Ainda, o testador deve estar habilitado a receber como entrada qualquer saída L_U.

Assim sendo, o comportamento do testador é modelado como um IOTS, porém com as entradas e saídas contrárias a da implementação. Para denotar a observação da quiescência por parte do testador, um novo rótulo $\theta \notin L_I \cup L_U \cup \{\tau, \delta\}$ é considerado. Logo, a ocorrência de θ em um teste indica que nenhuma saída foi produzida pela implementação.

Na prática, constatar a ausência de saídas de uma implementação pode durar um tempo infinito. Diante disso, pode-se pensar em θ como sendo a expiração de um tempo limite. Este tempo limite deve ser escolhido com cautela, para que após esse período de tempo sem receber alguma saída, o sistema é de fato quiescente.

Prosseguindo com a formalização de um caso de teste, dois estados especiais **pass** e **fail** são necessários para estabelecer uma decisão (veredito). Isto quer dizer que, para conseguir estabelecer esta decisão em tempo finito, os casos de testes devem sempre conseguir alcançar o estado **pass** ou **fail** dentro de um número finito de transições. Ao alcançar algum destes estados especiais, o teste não pode mais deixar este estado.

Por a implementação sempre aceitar as entradas do ambiente, conceder mais de uma entrada simultaneamente provocaria um não determinismo desnecessário. Portanto, também se assume que o comportamento do testador é determinístico.

A partir destas considerações, Tretmans (2008) formaliza a classe *Test Transition Systems* (TTS). Esta classe enuncia que um caso de teste t, para uma implementação com entradas L_I e saídas L_U, é um IOTS (Q, L_U, $L_I \cup \{\theta\}$, T, q_0) tal que:

- t possui uma quantidade finita de estados e é determinístico;
- Q contém dois estados especiais **pass** e **fail**, onde **pass** $\neq$ **fail**;
- todas as transições que partem dos estados **pass** e **fail** voltam para si mesmos;
- t não possui ciclos, com exceção dos ciclos dos estados **pass** e **fail**;
- para qualquer estado $q \in Q$ do caso de teste, $init(q) = \{a\} \cup L_U$, para algum $a \in L_I$ ou $init(q) = \{\theta\} \cup L_U$.

A Figura 9 exemplifica o funcionamento de um caso de teste ($t1$). Este teste concede a entrada *but* para a implementação e, somente após isso, espera receber *liq* como resposta da implementação. Após receber a saída *liq*, o teste espera que nenhuma outra ação seja produzida, ou seja, que a implementação fique em estado de quiescência. Qualquer outro resultado é considerado falho pelo caso de teste.

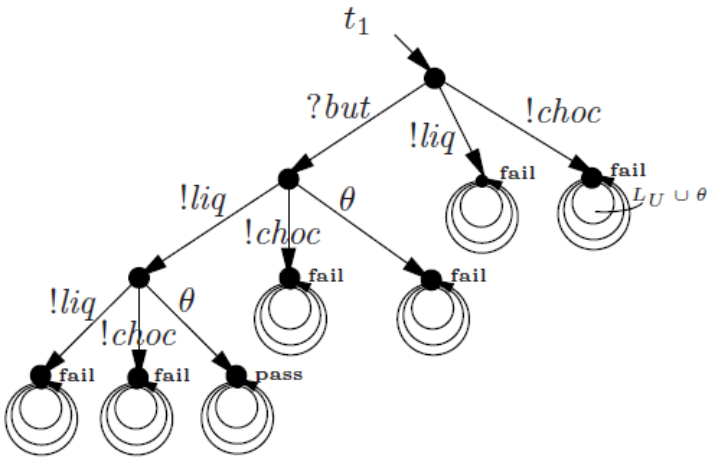


Figura 9 – Caso de teste definido por Tretmans (2008, p.20).

2.2.5 Execução de testes

Uma execução de teste é descrita por Tretmans (2008) como um experimento onde o caso de teste t fornece as entradas para a implementação i, observando as saídas ou a ausência delas.

Para representar uma execução de teste, o trabalho introduz um novo operador (na linguagem *Behaviour Expressions*) $\rceil\!|$, onde $t \rceil\!| i$ representa a execução em paralelo do teste e da implementação. Esta execução é síncrona em todos rótulos visíveis (a implementação pode evoluir individualmente através da realização de eventos internos) e leva em consideração a simetria entre os rótulos θ e δ. Desta forma, o operador pode ser descrito pelas regras de inferência (semântica operacional) apresentadas na Figura 10.

$$\frac{i \xrightarrow{\tau} i'}{t \rceil\!| i \xrightarrow{\tau} t \rceil\!| i'} \qquad \frac{t \xrightarrow{a} t',\ i \xrightarrow{a} i'}{t \rceil\!| i \xrightarrow{a} t' \rceil\!| i'}\ a \in L_I \cup L_U \qquad \frac{t \xrightarrow{\theta} t',\ i \xrightarrow{\delta}}{t \rceil\!| i \xrightarrow{\theta} t' \rceil\!| i}$$

Figura 10 – Regras de inferência do operador $\rceil\!|$ definidas por Tretmans (2008).

A execução de um teste é definida como um *trace* de $t \rceil\!| i$ que alcança o estado **fail** ou **pass** no teste. De maneira precisa, a execução de um teste é formalizada como a seguir.

(19) *σ é a execução de um teste* $t \rceil\!| i \Leftrightarrow_{def} \exists i' : t \rceil\!| i \stackrel{\sigma}{\Rightarrow} \mathbf{pass} \rceil\!| i'$ *ou* $t \rceil\!| i \stackrel{\sigma}{\Rightarrow} \mathbf{fail} \rceil\!| i'$

Diz-se que uma implementação passa um caso de teste se nenhuma execução de teste leva para o estado **fail**. Esta definição é representada pela relação **passes** e é definida formalmente da maneira abaixo. Esta relação também pode ser considerada no contexto de uma suíte (conjunto) de testes T.

(20) i **passes** $t \Leftrightarrow_{def} \forall \sigma \in L^*_\theta, \forall i' : \neg\ t \rceil\!| i \stackrel{\sigma}{\Rightarrow} \textbf{fail} \rceil\!| i'$

Por exemplo, considere o IOTS $k1$ mostrado na Figura 11 e o caso de teste $t1$ mostrado na Figura 9. O *trace* $?but \cdot !liq \cdot \theta$ é a única execução de teste de $t1 \rceil\!| k1$. Esta execução de teste leva apenas ao estado **pass** de $t1$, ou seja, $t1 \rceil\!| k1 \stackrel{?but \cdot !liq \cdot \theta}{\Longrightarrow}$ **pass** $\rceil\!| p2$. Assim, pode-se afirmar que $k1$ passa no caso de teste $t1$.

Por outro lado, considerando o mesmo caso de teste e o IOTS $k3$ da Figura 5, as execuções de teste possíveis de $t1 \rceil\!| k3$ são os *traces* $?but \cdot !liq \cdot \theta$ e $?but \cdot \theta$. Apesar de $?but \cdot !liq \cdot \theta$ levar para o estado **pass** de $t1$, ou seja, $t1 \rceil\!| k3 \stackrel{?but \cdot !liq \cdot \theta}{\Longrightarrow}$ **pass** $\rceil\!| r3$, o *trace* $?but \cdot \theta$ leva ao estado **fail**, i.e., $t1 \rceil\!| k3 \stackrel{?but \cdot \theta}{\Longrightarrow}$ **fail** $\rceil\!| r2$. Desta forma, a implementação $k3$ não passa no caso de teste $t1$.

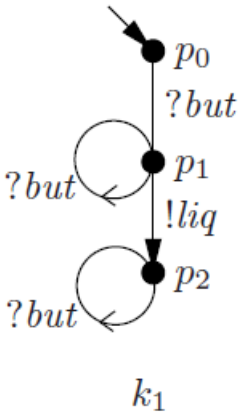


Figura 11 – IOTS $k1$ definido por Tretmans (2008, p.14).

Por fim, Tretmans (2008) define um algoritmo de geração de testes, que não é apresentado aqui dado que a sua formalização não faz parte do escopo deste trabalho, e prova que a teoria de testes proposta é correta (*sound*). Seja T o conjunto de todos os testes que podem ser gerados pelo algoritmo proposto para uma especificação s, tem-se que:

(21) $\forall i \in \text{IOTS}(L_I, L_U)\ :\ i$ **ioco** $s \implies i$ **passes** T

Da lógica clássica, sabe-se que $P \implies Q$ é logicamente equivalente a $\neg Q \implies \neg P$. Portanto, a corretude da teoria de testes significa que, se uma implementação i não passar na suíte de testes T, esta implementação é necessariamente não conforme em relação à especificação s.

3 Execução de testes para **ioco** em Coq

Em seu trabalho, Santana (2020) alterou a implementação de LTS anteriormente proposta por Sobral (2019). Consequentemente, esta mudança precisa ser propagada para todas as outras definições dependentes do tipo LTS feitas por Sobral (2019). Em um primeiro momento, esta pesquisa realiza a integração destes dois trabalhos, ajustando as definições necessárias para que fiquem de acordo com a alteração na implementação de LTS. Em seguida, são realizados ajustes em algumas definições de Sobral (2019) que estavam incompletas. Estas também são refraseadas seguindo o mesmo padrão de definições adotado por Santana (2020). A Seção 3.1 detalha o resultado destas atividades.

Posteriormente, este trabalho formaliza em Coq a noção de casos de teste, assim como a execução dos mesmos, de acordo com as definições de Tretmans (2008). A Seção 3.2 apresenta esta formalização e a ilustra com exemplos que passam e falham em um determinado teste. Por fim, na Seção 3.3, será introduzida uma forma de visualizar graficamente as estruturas definidas no trabalho, por meio da ferramenta Graphiz.

Todo o código produzido nesta pesquisa está disponível no GitHub (<https://github.com/iocoCoq/iocoCoqFormalization/>), onde também estão os resultados dos trabalhos desenvolvidos anteriormente. A Figura 12 mostra os arquivos que foram adicionados ao repositório.

Os arquivos incorporados ao diretório *src* possuem 674 linhas de código de novas definições e 324 linhas de código de scripts de prova. A lista a seguir detalha o conteúdo de cada arquivo.

- **LTS_functions.v**: engloba todas as definições funcionais elaboradas no contexto de LTS, incluindo os enunciados dos teoremas que fazem a relação destas definições com as respectivas definições indutivas mostradas na Seção 3.1.1;
- **IOTS.v**: formaliza as definições de LTS com entradas e saídas (IOLTS), IOTS e quiescência apresentados na Seção 2.2.1. Estas formalizações serão discutidas em mais detalhes na Seção 3.1.2;
- **IOCO.v**: inclui as definições necessárias para a formalização da relação de conformidade **ioco**, introduzidas na Seção 2.2.3. Estas formalizações serão apresentadas na Seção 3.1.3;
- **TTS.v**: contém as formalizações de casos de teste e execução de um caso de teste de acordo com as definições apresentadas na Seção 2.2.4. O conteúdo deste arquivo será discutido em mais detalhes na Seção 3.2;

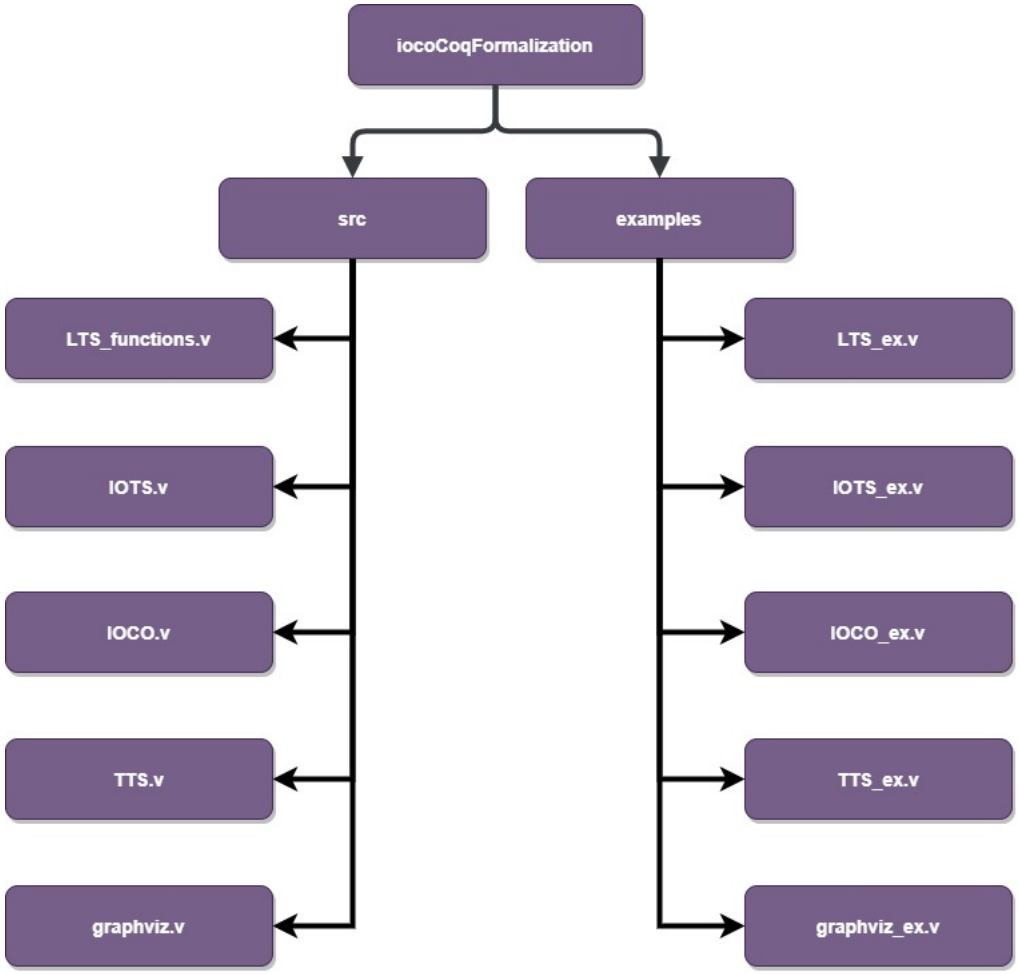


Figura 12 – Novos arquivos adicionados ao repositório.

- **graphviz.v**: reúne as funções que possibilitam a geração automática de uma representação gráfica de LTSs. Estas funções serão discutidas detalhadamente na Seção 3.3.

Os novos arquivos da pasta *examples* são listados abaixo, explicitando o conteúdo de cada um deles. Os arquivos de exemplos possuem 723 linhas de código.

- **LTS_ex.v**: poussui exemplos de LTSs, incluindo o exemplo da Figura 2 do artigo de Tretmans (2008), além de exemplos das definições contidas no arquivo LTS.v;
- **IOTS_ex.v**: exemplifica as definições do arquivo IOTS.v, descrevendo as estruturas das Figuras 4 e 6 do artigo de Tretmans (2008);
- **IOCO_ex.v**: traz exemplos de *out* e de verificação de conformidade em **ioco**.

Adicionalmente, os seguintes arquivos, já existentes em *src*, foram modificados.

- **LTS.v**: ajustes nas definições propostas anteriormente por Sobral (2019) para que ficassem consistentes com o trabalho de Santana (2020), mais detalhes na Seção 3.1.1;

- **list_helper.v**: novas funções auxiliares, no contexto de listas, foram adicionadas;
- **ltacs.v**: para ajudar em provas envolvendo LTSs e novas estruturas, novas táticas de automação foram criadas usando *Ltac*.

3.1 Integração dos trabalhos anteriores

Enquanto a Seção 3.1.1 descreve as alterações feitas no contexto do tipo LTS, a Seção 3.1.2 considera o tipo IOTS. Em seguida, a Seção 3.1.3 apresenta as alterações feitas nas definições associadas à relação de conformidade **ioco**.

3.1.1 LTS.v e LTS_functions.v

Em seu trabalho, Sobral (2019) formaliza LTS como o *Record* abaixo.

```
Record LTS : Type := mkLTS {
      Q : list state
    ; L : list label
    ; T : list (state × label × state)
    ; q0 : state
    ; Q_non_empty : Q ≠ []
    ; tau_not_in_L : ¬ (In tau L)
    ; q0_in_Q : In q0 Q
    ; valid_transitions : each_transition_is_valid T Q L
    ; no_repetition_Q : list_with_unique_elements Q
    ; no_repetition_L : list_with_unique_elements L
    ; no_repetition_T : list_with_unique_elements T
  }.
```

Nesta definição, os rótulos (*label*) das transições do LTS são do tipo *string*, onde a *string* especial "tau" é usada para caracterizar o rótulo da transição interna (τ). Por este motivo, tem-se a regra *tau_not_in_L* que garante que "tau" não pode ser uma *string* em *L*. Santana (2020) considerou uma definições diferente de LTS, apresentada no trecho de código seguinte.

```
Record LTS : Type := mkLTS {
      Q : set state
    ; L : set label
    ; T : set transition
    ; q0 : state

    ; Q_non_empty : Q ≠ []
```

```
  ; q0_in_Q : In q0 Q
  ; valid_transitions : each_transition_is_valid T Q L
  ; no_repetition_Q : NoDup Q
  ; no_repetition_L : NoDup L
  ; no_repetition_T : NoDup T
}.
```

Nesta nova formalização, um novo tipo *transition_label* foi criado, em que a transição interna (τ) passou a ser representada pelo construtor *tau* enquanto as demais transições continuam sendo caracterizadas pelo tipo *string* e representadas pelo construtor *event*. Esta reformulação, impede, por construção, o uso do rótulo associado à τ como um elemento de L.

Ademais, Santana (2020) substituiu a definição *list_with_unique_elements*, que garante que não há elementos repetidos em uma lista, pela definição semelhante *NoDup*, disponível na biblioteca padrão de *Coq*. Em Coq, um *set* é implementado como uma lista. Portanto, exceto quando se tem a garantia de que o conjunto é criado usando exclusivamente a função *set_add*, é aconselhado verificar a ausência de repetições em elementos do tipo *set*.

Dada estas alterações, neste trabalho foi desenvolvida uma tática em Coq que, dado o conjunto de estado (Q), o conjunto de rótulos (L), o conjunto de transições (T) e o estado inicial, cria uma instância de LTS provando automaticamente todas as propriedades do *Record*. O trecho de código a seguir mostra a definição desta tática chamada de *solve_LTS_rules*. Esta tática, após informar os dados da instância do *Record* através do comando *apply*, faz uso de casamento de padrões para chamar outras táticas auxiliares. Por exemplo, *list_has_no_dup* prova a ausência de elementos repetidos em uma lista.

```
Ltac solve_LTS_rules Q L T q0 := apply (mkLTS Q L T q0) ;
  repeat (
    match goal with
    | ⊢ In _ _ ⇒ elem_in_list
    | ⊢ _ ≠ nil ⇒ solve_list_not_empty
    | ⊢ each_transition_is_valid _ _ _ ⇒ solve_transition_valid
    | ⊢ NoDup _ ⇒ list_has_no_dup
    end
  ) ; fail "One or more contextual rules were not fulfilled".
```

Com esta tática (*solve_LTS_rules*), é possível definir um LTS de maneira simples, sem precisar interagir com o assistente de provas para provar as propriedades de boa formação de um LTS. Por exemplo, o LTS mostrado anteriormente na Figura 3, é definido em Coq da seguinte maneira.

```
Definition fig2_r' : LTS.
Proof.
  solve_LTS_rules
            [0;1;2;3;4;5]
            ["but";"liq";"choc"]
            [(0, event "but", 1);(1, event "liq", 3);
             (0, event "but", 2);(2, event "but", 4);(4, event "choc", 5)]
            0.
Defined.
```

Como a descrição de um LTS também pode ser feita na linguagem *Behaviour Expressions*, a tática *create_LTS_from_BE* foi criada. Abaixo, é mostrado o corpo desta tática que internamente utiliza as funções e teoremas definidos por Santana (2020).

```
Ltac create_LTS_from_BE ctx start i :=
  let H := fresh "H"in
  let t := fresh "t"in
    destruct (createBehaviourTransSet ctx start i) as [t|] eqn:H;
    [inversion H |
     inversion H; fail "createBehaviourTransSet didn't generate valid trans set"];
    apply createBehaviourTransSet_BehaviourTransSetR in H;
    apply (createLtsFromBehaviourTransSet ctx t start H).
```

Esta tática tem como objetivo transformar a definição de um LTS feita na linguagem *Behaviour Expressions* para uma instância do *Record* LTS. Neste caso, Santana (2020) prova que o LTS gerado a partir de uma descrição textual satisfaz por construção as regras de boa formação de um LTS. A definição equivalente do LTS *fig_2_r* nesta linguagem é mostrada a seguir.

```
Definition fig2_r_BE : BehaviourExpressions.
Proof.
  create_behaviour_expressions
    ["fig2_r"::= "but";; "liq";; STOP [] "but";; "but";; "choc";; STOP].
Defined.

Definition fig2_r : LTS.
Proof. create_LTS_from_BE fig2_r_BE "fig2_r"6. Defined.
```

Devido às modificações introduzidas por Santana (2020), foi necessário ajustar outras definições formalizadas por Sobral (2019) para considerar o novo tipo *transition_label*. Como exemplo, pode-se destacar a formalização da Definição 1 da Seção 2.2.1, que em Coq tem o nome *ind_transition*. Antes, este conceito era definido como mostrado abaixo.

```
Inductive ind_transition : state → label → state → LTS → Prop :=
```

```
| transition_r1 : ∀ (s s' : state) (l : label) (p : LTS),
                    In s p.(Q) ∧ In s' p.(Q) ∧ (In l p.(L) ∨ l = tau)
                      ∧ In (s, l, s') p.(T) → ind_transition s l s' p.
```

Esta definição foi refatorada com o intuito de criar regras distintas para transições internas e externas. Ainda, foram removidas as verificações de que os estados e o rótulo estão, respectivamente, no conjunto Q e L. Isto foi feito sob a justificativa de que, segundo a regra *valid_transitions* do LTS, se uma transição (s, l, s') está em T, os estados (s e s') e o rótulo (l) são válidos, i.e. os estados estão em Q e o rótulo está em L ou é *tau*. Por fim, usou-se também uma sintaxe para criar definições indutivas menos verbosa, que elimina a necessidade de explicitar os quantificadores universais. A nova definição de *ind_transition*, após as modificações descritas, é mostrada a seguir.

```
Inductive ind_transition : state → transition_label → state → LTS → Prop :=
  | transition_r1 (s s' : state) (l : label) (p : LTS):
      In (s, event l, s') p.(T) →
      ind_transition s (event l) s' p
  | transition_r2 (s s' : state) (p : LTS) :
      In (s, tau, s') p.(T) →
      ind_transition s tau s' p.
```

Observe que a nova definição é mais legível e se aproxima mais da definição original, apresentada no trabalho de Tretmans (2008). Se a tripla $(s, event\ l, s')$ está nas transições do LTS p, então pode-se dizer que $s \xrightarrow{l} s'$ (regra *transition_r1*). A regra *transition_r2* é similar à anterior, porém considerando o evento interno τ.

A definição *ind_transition_seq* modela a composição de transições apresentada na Definição 2 da Seção 2.2.1. A forma como esta havia sido formalizada previamente é mostrada abaixo.

```
Inductive ind_transition_seq : state → list label → state → LTS → Prop :=
| transition_seq_r1 : ∀ (s s' : state) (ll : list label) (p : LTS),
      ((length ll = 1) ∧ (ind_transition s (hd tau ll) s' p))
      ∨
      ((length ll > 1) ∧ (∃ (si : state), ind_transition s (hd tau ll) si p ∧
            ind_transition_seq si (tl ll) s' p))
      → ind_transition_seq s ll s' p.
```

Além de considerar o novo tipo *transition_label*, a definição foi reestruturada em duas regras: uma regra base para listas de tamanho um; e uma regra indutiva para listas de tamanho maior do que um. As estruturas das regras também foram modificadas, com a finalidade de conseguir explorar indutivamente a lista de maneira mais simples.

```
Inductive ind_transition_seq : state → list transition_label → state → LTS → Prop :=
```

```
| transition_seq_r1 (s s' : state) (l : transition_label) (p : LTS) :
    ind_transition s l s' p →
    ind_transition_seq s [l] s' p
| transition_seq_r2 (s s' si : state) (l1 l2 : transition_label)
      (ll : list transition_label) (p : LTS) :
    ind_transition s l1 si p →
    ind_transition_seq si (l2 :: ll) s' p →
    ind_transition_seq s (l1 :: l2 :: ll) s' p.
```

A regra *transition_seq_r1* captura a situação em que, se $s \xrightarrow{l} s'$, então $s \xrightarrow{\mu_1 \ldots \mu_n} s'$, onde a lista $\mu_1 ... \mu_n$ é unitária e possui somente o elemento l. A regra *transition_seq_r2* formaliza a situação em que é possível concluir $s \xrightarrow{\mu_1 \cdot \mu_2 \cdot \mu_3 \ldots \mu_n} s'$, dado que $s \xrightarrow{\mu_1} si$ e $si \xrightarrow{\mu_2 \cdot \mu 3 \ldots \mu_n} s'$, onde si representa um estado intermediário. Na definição em Coq, os rótulos $l1$ e $l2$ representam μ_1 e μ_2, respectivamente, enquanto que ll denota a lista $\mu_3 ... \mu_n$.

Outras definições do arquivo LTS.v foram alteradas de maneira semelhante, modificadas de acordo com o novo tipo *transition_label* e subdividindo regras, quando pertinente. Em particular, esta refatoração permitiu encontrar um erro na definição *ind_seq_reachability*, que formaliza a Definição 5 da Seção 2.2.1. Abaixo, é apresentada a definição como havia sido formalizada por Sobral (2019).

```
Inductive ind_seq_reachability : state → list label → state → LTS → Prop :=
| seq_reachability_r1 :
    ∀ (s s' : state) (ll : list label) (p : LTS),
      (ll = [] ∧ ind_empty_reachability s s p)
          ∨
      ((ll ≠ [] ∧ tl ll = []) ∧ ind_one_step_reachability s (hd tau ll) s' p)
          ∨
      ((tl ll ≠ []) ∧ (∃ (si : state),
          ind_one_step_reachability s (hd tau ll) si p ∧
          ind_seq_reachability si (tl ll) s' p))
    → ind_seq_reachability s ll s' p.
```

O único construtor desta definição possui disjunções criando três casos possíveis: se a lista de rótulos (ll) é vazia, se ll contém exatamente um elemento, e se ll contém mais do que um elemento. Para o primeiro caso, a formalização considera *ind_empty_reachability s s p*, que não estabelece qualquer restrição sobre s'. Portanto, é possível sempre concluir *ind_seq_reachability s* [] s' *p*, independente do valor de s', já que, por definição, *ind_empty_reachability s s p* é sempre verdade.

Durante a refatoração, constatou-se que apenas dois dos três casos eram realmente necessários de serem considerados. Também foi corrigido o enunciado do primeiro caso, alterando *ind_empty_reachability s s p* para *ind_empty_reachability s* s' *p*. O código a

seguir mostra o resultado após estas alterações.

```
Inductive ind_seq_reachability : state → list label → state → LTS → Prop :=
  | seq_reachability_r1 (s s' : state) (p : LTS) :
      ind_empty_reachability s s' p →
      ind_seq_reachability s [] s' p
  | seq_reachability_r2 (s si s' : state) (l1: label) (ll : set label) (p : LTS) :
      ind_one_step_reachability s l1 si p →
      ind_seq_reachability si ll s' p →
      ind_seq_reachability s (l1 :: ll) s' p.
```

As definições *init* e *after*, apresentadas na Seção 2.2.1, haviam sido definidas de maneira indutiva e funcional por Sobral (2019). As definições funcionais foram separadas em um novo arquivo, LTS_functions, e também sofreram reformulações. Abaixo, a antiga definição de *f_init* é apresentada.

```
Fixpoint f_init (s : state)
  (lt : list (state × label × state)) : set label :=
 match lt with
 | [] ⇒ []
 | h :: t ⇒ match h with
              | (a,l,b) ⇒ if s =? a
                          then set_add string_dec l (f_init s t)
                          else (f_init s t)
              end
 end.
```

Na refatoração, a função definida por Sobral (2019) foi renomeada para *f_init_aux* e a nova implementação de *f_init* instancia *f_init_aux* com os argumentos corretos. Esta modificação garante que *f_init_aux* sempre será aplicada considerando as transições do LTS. A definição após os ajustes pode ser observada a seguir.

```
Fixpoint f_init_aux (s : state)
  (lt : list (state × transition_label × state)) : set transition_label :=
 match lt with
 | [] ⇒ []
 | h :: t ⇒ match h with
              | (a,l,b) ⇒ if s =? a
                          then set_add transition_label_dec l (f_init_aux s t)
                          else (f_init_aux s t)
              end
 end.
```

```
Definition f_init (s : state) (p : LTS) : set transition_label :=
  f_init_aux s p.(T).
```

Um teorema que garante a equivalência entre a definição indutiva (*ind_init*) e funcional (*f_init*) foi provado e possui o cabeçalho mostrado a seguir. Ter definições funcionais e indutivas (semanticamente equivalentes) de um mesmo conceito é útil. A definição funcional é útil quando se deseja realizar computações envolvendo o conceito em questão. Já a definição indutiva é mais adequada para certos tipos de manipulações lógicas.

```
Theorem ind_init_reflect :
  ∀ (s : state) (ll : set transition_label) (p : LTS),
    ind_init s ll p ↔ f_init s p [=] ll.
```

Durante os ajustes, também foi possível notar que a definição funcional de *after* (*f_after*) estava incompleta. Uma nova lógica para o funcionamento desta função foi pensada, pois, na maneira anterior, nem todos os estados alcançáveis eram contemplados.

Agora, o resultado de *f_after* é definido com o apoio de várias funções auxiliares. A primeira delas é *f_after_one_transition_label'*. Esta função computa os estados que estão diretamente ligados a um dado estado inicial (*s*) a partir de um rótulo (*l*).

```
Fixpoint f_after_one_transition_label' (s : state) (l : transition_label)
  (lt : list transition) : set state :=
  match lt with
  | [] ⇒ []
  | (a, l', b) :: t ⇒
      if (s =? a) && (b_transition_label_dec l l')
      then set_add Nat.eq_dec b (f_after_one_transition_label' s l t)
      else f_after_one_transition_label' s l t
  end.
```

Em seguida, dada uma lista de estados (*ls*), a função *f_after_one_transition_label* utiliza a função anterior para computar todos os estados que estão ligados a um estado de *ls* pelo rótulo *l*.

```
Fixpoint f_after_one_transition_label (ls : set state) (l : transition_label)
    (lt : list transition) : set state :=
  match ls with
  | [] ⇒ []
  | h :: t ⇒
      set_union
        Nat.eq_dec
        (f_after_one_transition_label' h l lt)
```

```
        (f_after_one_transition_label t l lt)
  end.
```

A função *f_after_n_tau*, apresentada no trecho de código a seguir, computa o conjunto de estados alcançáveis a partir do conjunto de estados (ls), realizando de 0 até n transições internas.

```
Fixpoint f_after_n_tau (ls : set state) (n : nat) (lt : list transition) : set state :=
  match n, ls with
  | _, [] ⇒ []
  | 0, _ ⇒ ls
  | S n', _ ⇒ set_union
                  Nat.eq_dec
                  ls
                  (f_after_n_tau (f_after_one_transition_label ls tau lt) n' lt)
  end.
```

Em seguida, a função *all_reachable_by_tau* utiliza a função anterior para computar todos os estados alcançáveis a partir do conjunto de estados ls. Para isto, observa-se o fato que, em um grafo com n estados, se um estado é alcançável, existe um caminho (de estados) de tamanho menor ou igual a n que alcança o estado.

```
Definition all_reachable_by_tau (ls : set state) (p : LTS) :=
  f_after_n_tau ls (length p.(Q)) p.(T).
```

A função *f_after'* computa o conjunto de estados alcançáveis a partir de um conjunto de estados ls, seguindo uma sequência de rótulos externos, mas também permitindo a ocorrência de transições internas. Por fim, a função *f_after* utiliza *f_after'* considerando como ponto de partida um único estado inicial s.

```
Fixpoint f_after' (ls : set state) (ll : list label) (p : LTS) : set state :=
  match ll with
  | [] ⇒ all_reachable_by_tau ls p
  | h :: ll' ⇒
    let ls_tau := all_reachable_by_tau ls p in
    let ls_after_one_step := f_after_one_transition_label ls_tau (event h) p.(T) in
      f_after' ls_after_one_step ll' p
  end.
Definition f_after (s : state) (ll : list label) (p : LTS) : set state :=
  f_after' [s] ll p.
```

Além das definições mencionadas, outras também foram modificadas no arquivo LTS.v: *ind_state_reaches_some_other*, *ind_empty_reachability*, *ind_one_step_reachability*, *ind_has_reachability_to_some_other*, *ind_after*, *ind_refuses*, *ind_der* e *ind_deterministic*.

Após definir um LTS, Tretmans (2008) destaca as condições necessárias para que um LTS faça parte de uma classe mais restrita: *strongly converging*. Esta classe garante que o LTS criado não possui ciclos formados apenas por transições internas. Um *Record* que formaliza esta definição foi criado e sua implementação é mostrada a seguir. Como é possível observar, *strongly_converging* define que um LTS é fortemente convergente se, e somente se, para todo estado s e *trace* t (não vazio) formado só por eventos internos (τ), não é possível alcançar s a partir de s realizando t.

```
Definition strongly_converging (lts : LTS) : Prop :=
  ∀ (s : state) (t : list transition_label),
    t ≠ [] ∧ all_labels_tau t →
    ¬ ind_transition_seq s t s lts.

Record SC_LTS : Type := mkSC_LTS {
     lts : LTS
   ; is_strongly_converging : strongly_converging lts
  }.
```

O LTS *fig2_r*, mostrado anteriormente, não possui ciclos formados por transições internas. Então, é possível criar uma instância de *SC_LTS* passando este LTS como argumento, e assim é feito no trecho de código a seguir. Para concluir a definição, é preciso provar que este LTS é de fato fortemente convergente.

```
Definition fig2_r_SC_LTS : SC_LTS.
Proof.
  apply (mkSC_LTS fig2_r).
  unfold strongly_converging. intros s t H. destruct t.
  - destruct H. unfold not in H. exfalso. apply H. reflexivity.
  - unfold all_labels_tau in H. destruct H as [_ [H _]].
    unfold not. intros H'. inversion H'; subst.
    + proof_absurd_transition H5.
    + proof_absurd_transition H3.
Defined.
```

A tática auxiliar *proof_absurd_transition*, apresentada abaixo, foi definida para auxiliar o desenvolvimento de provas onde existe uma hipótese envolvendo *ind_transition*.

```
Ltac expand_transition Ht :=
  let H := fresh "H" in
    inversion Ht as [ ? ? ? ? H | ? ? ? H]; vm_compute in H; expand_In H.

Ltac proof_absurd_transition H :=
  expand_transition H; fail "Unable to proof invalid transition".
```

Esta tática termina a prova quando não existem transições no LTS que seguem o

padrão definido na hipótese e falha caso não consiga fazer isto.

3.1.2 IOTS.v

Este arquivo é nomeado como IOTS, uma vez que esta é uma definição chave para a da teoria de testes proposta por Tretmans (2008). A formalização prévia em Coq de um IOTS não foi modificada por este trabalho, mas sim as definições diretamente associadas a um IOLTS. É importante lembrar que um IOTS é uma sub-classe de IOLTS (LTS com entradas e saídas). Portanto, nesta seção, são descritas as alterações realizadas no contexto do tipo IOLTS.

Sobral (2019) formalizou o conceito de LTS com entradas e saídas (IOLTS) através da estrutura *IOLTS*, mostrada no trecho de código abaixo.

```
Record IOLTS : Type := mkIOLTS {
      lts : LTS
    ; L_i : list label
    ; L_u : list label
    ; disjoint_input_output_labels : is_disjoint L_i L_u
    ; is_IOLTS : is_valid_LTS (L_i ++ L_u) lts
    ; no_repetition_L_i : list_with_unique_elements L_i
    ; no_repetition_L_u : list_with_unique_elements L_u
  }.
```

Nesta definição, *is_IOLTS* garante que o LTS recebido (*lts*) é *strongly converging* e o conjunto de rótulos de entrada e saída (respectivamente L_i e L_u) compõem o conjunto de rótulos de lts. Após a restruturação, a definição passou a aceitar apenas LTSs que já são *strongly converging* (*SC_LTS*) e, consequentemente, a primeira verificação passa a ser desnecessária.

Além disso, na definição da regra *disjoint_input_output_labels*, a função *is_disjoint* foi modificada para usar a relação *Forall* provinda da biblioteca padrão de Coq, facilitando o desenvolvimento de provas, por já possuir teoremas provados envolvendo esta relação. Pelo mesmo motivo, as regras *no_repetition_L_i* e *no_repetition_L_u* passaram a utilizar a definição *NoDup*, também definida na biblioteca padrão. O código a seguir reflete as alterações comentadas.

```
Record IOLTS : Type := mkIOLTS {
      sc_lts : SC_LTS
    ; L_i : list label
    ; L_u : list label
    ; disjoint_input_output_labels : is_disjoint L_i L_u
    ; is_IOLTS : (L_i ++ L_u) [=] sc_lts.(lts).(L)
```

```
    ; no_repetition_L_i : NoDup L_i
    ; no_repetition_L_u : NoDup L_u
  }.
```

A função *create_IOLTS* foi definida com o objetivo computar um *option IOLTS* a partir de um SC_IOLTS e dos conjuntos L_I e L_U. Nela, há uma verificação automática para cada regra que caracteriza um IOLTS e, caso estas verificações sejam bem-sucedidas, a função retornará o construtor *Some* aplicado ao IOLTS gerado. Caso contrário, o retorno será *None*.

```
Definition create_IOLTS (sc_lts : SC_LTS) (Li Lu : list label) : option IOLTS :=
    match is_disjoint_dec Li Lu with
    | left h_is_disjoint ⇒
        match Equiv_dec (Li ++ Lu) sc_lts.(lts).(L) string_dec with
        | left h_is_IOLTS ⇒
            match NoDup_dec Li string_dec with
            | left h_NoDup_Li ⇒
                match NoDup_dec Lu string_dec with
                | left h_NoDup_Lu ⇒
                    Some (mkIOLTS
                            sc_lts
                            Li
                            Lu
                            h_is_disjoint
                            h_is_IOLTS
                            h_NoDup_Li
                            h_NoDup_Lu)
                | right _ ⇒ None
                end
            | right _ ⇒ None
            end
        | right _ ⇒ None
        end
    | right _ ⇒ None
    end.
```

O teorema *create_IOLTS_correct* foi enunciado para comprovar que, se um SC_LTS juntamente com listas L_I e L_U estão de acordo com todas as regras de boa formação de um IOLTS, então a função *create_IOLTS* sempre retorna um IOLTS. O enunciado do teorema é mostrado no trecho de código a seguir.

```
Theorem create_IOLTS_correct :
```

```
∀ (sc_lts : SC_LTS) (L_i L_u : list label),
  is_disjoint L_i L_u ∧ (L_i ++ L_u) [=] sc_lts.(lts).(L) ∧ NoDup L_i ∧ NoDup L_u
  → create_IOLTS sc_lts L_i L_u ≠ None.
```

Também foi desenvolvida uma tática que auxilia na criação de um IOLTS. Esta definição complementa a função definida antes (*create_IOLTS*). Enquanto a função consiste em caracterização funcional de como um IOLTS pode ser criado a partir de um LTS e dos conjuntos de rótulos de entrada e de saída, a tática traz uma perspectiva diferente (lógica). Esta tática recebe um LTS, a lista de rótulos de entrada L_i, a lista de rótulos de saída L_u, e cria uma nova instância de IOLTS provando automaticamente todas as propriedades, se possível. Abaixo, tem-se como esta tática foi definida em Coq.

```
Ltac solve_IOLTS_rules lts Li Lu := apply (mkIOLTS lts Li Lu) ;
  repeat (
    match goal with
    | ⊢ is_disjoint _ _ ⇒ disjoint_sets
    | ⊢ list_helper.Equiv _ _ ⇒ proof_Equiv
    | ⊢ NoDup _ ⇒ list_has_no_dup
    end
  ) ; fail "One or more contextual rules were not fulfilled".
```

Assim, é possível criar a instância de IOLTS *fig2_r_IOLTS*, semelhante à apresentada na Figura 4, como mostra o trecho de código a seguir. Esta instância é criada a partir de *fig2_r_SC_LTS*, definida no final da Seção 3.1.1, e os conjuntos: {*but*}, como o conjunto de rótulos de entrada; e {*liq*, *choc*}, como o conjunto de rótulos de saída.

```
Definition fig2_r_IOLTS : IOLTS.
Proof.
  solve_IOLTS_rules fig2_r_SC_LTS ["but"] ["liq"; "choc"].
Defined.
```

O conceito de IOLTS com as anotações de quiescência, apresentado na Definição 14 da Seção 2.2.1, foi formalizado por Sobral (2019) através do *Record s_IOLTS*. Esta formalização é mostrada no trecho de código a seguir.

```
Record s_IOLTS : Type := mksIOLTS {
      s_iolts : IOLTS
    ; Ts : list (state × label × state)
    ; Q_del := s_iolts.(lts).(Q)
    ; L_del := set_union string_dec s_iolts.(lts).(L) [delta]
    ; Li_del := s_iolts.(L_i)
    ; Lu_del := set_union string_dec s_iolts.(L_u) [delta]
    ; T_del := s_iolts.(lts).(T) ++ Ts
```

```
    ; delta_not_in_L : ¬ In delta s_iolts.(lts).(L)
    ; valid_del_transitions : ∀ (s s' : state) (l : label),
                              In (s, l, s') Ts → s = s' ∧ l = delta ∧ In s Q_del
    ; Ts_complete_correct :
        ∀ (s : state), In (s, delta, s) Ts ↔ In s Q_del ∧ ind_quiescent s s_iolts
}.
```

A remodulação deste *Record* é mostrada abaixo. Primeiramente, a lista com todas as transições δ (representada por Ts) passou a conter apenas todos os estados quiescentes do IOLTS. Em seguida, foram removidos os valores *Q_del, L_del, Li_del, Lu_del* e *T_del*, pois estes não são utilizados na definição. As propriedades *delta_not_in_L* e *valid_del_transitions* também foram removidas, uma vez que estas não são mais necessárias devido às modificações feitas no Ts. Por fim, a propriedade *Ts_complete_correct* foi ajustada para o novo formato de Ts. Obtém-se, assim, uma definição muito mais concisa de um IOLTS anotado com quiescência.

```
Record s_IOLTS : Type := mksIOLTS {
      iolts : IOLTS
    ; Ts : set state
    ; Ts_complete_correct :
        ∀ (s : state),
          In s Ts ↔ In s iolts.(sc_lts).(lts).(Q) ∧ ind_quiescent s iolts
}.
```

Também foi criada uma função para gerar um *s_IOLTS* a partir de um IOLTS. Para criá-la foi preciso definir a função *create_Ts*, cuja corretude foi provada no teorema *create_Ts_complete_correct*. A função *create_Ts* computa a lista de estados de um IOLTS que são quiescentes. Para isto, um filtro é realizado a partir da lista de todos os estados Q, removendo, assim, os estados que não são quiescentes. O corpo deta função é apresentado abaixo.

```
Definition create_Ts (iolts : IOLTS) : list state :=
  filter (is_quiescent iolts.(sc_lts).(lts).(T) iolts.(L_u)) iolts.(sc_lts).(lts).(Q).
```

A função *is_quiescent*, utilizada em *create_Ts*, calcula se um estado é quiescente, ou seja, se não possui nenhuma transição de saída partido deste estado. A definição *out_transitions* auxilia neste processo, verificando se na lista de transições T há alguma transição de saída partindo de um dado estado. O corpo destas funções é mostrado a seguir.

```
Definition out_transition (q : state) (Lu : set label)
    (t : state × transition_label × state) : bool :=
  match t with
```

```
  | (q1, e, q2) ⇒
      if Nat.eqb q1 q
      then
        match e with
        | tau ⇒ true
        | event e' ⇒ set_mem string_dec e' Lu
        end
      else false
  end.

Definition is_quiescent (l : list (state × transition_label × state))
    (Lu : list label) (q : state) : bool :=
  negb (existsb (out_transition q Lu) l).
```

Em *is_quiescente*, dado um tipo A, neste caso $state \times transition\_label \times state$, e uma função $f : A \to bool$, neste caso a aplicação parcial de *out_transition*, a função *existsb* retorna verdadeiro se existir um elemento da lista l que satisfaça o teste representado por f. Em outras palavras, a expressão (*existsb* (*out_transition q Lu*) *l*) retorna verdadeiro se existir em l uma transição partindo de q com um rótulo de saída.

Desta forma, é possível utilizar a função *create_s_IOLTS* em conjunto com o IOLTS *fig2_r_IOLTS*, definido anteriormente, para construir um s_IOLTS equivalente ao apresentado na Figura 6. O trecho de código a seguir mostra esta construção, atribuindo-lhe o nome *fig6_r*.

```
Definition fig6_r : s_IOLTS := create_s_IOLTS fig2_r_IOLTS.
```

A transição δ, representada no trabalho de Sobral (2019) por uma *string*, passou a ser retratada por um novo tipo indutivo *s_label*. Este tipo, semelhante ao *transition_label*, possui um construtor específico para o δ e um construtor para representar os demais rótulos das transições.

```
Inductive s_label : Type :=
  | s_event : label → s_label
  | delta : s_label.
```

Esta alteração tem impacto na definição de um *Strace* (Definição 15), pois, nesta também é necessário incluir as transições δ. Desta forma, foi necessário criar uma nova definição para *ind_seq_reachability* e posteriormente utilizá-la na formalização de um *Strace*. Estas novas definições são apresentadas a seguir.

```
Inductive ind_s_seq_reachability : state → list s_label → state → s_IOLTS
    → Prop :=
  | s_seq_reachability_r1 (s s' : state) (p : s_IOLTS) :
      ind_empty_reachability s s' p.(iolts).(sc_lts).(lts) →
```

```
      ind_s_seq_reachability s [] s' p
  | s_seq_reachability_r2 (s s' si : state) (l: label) (t : list s_label) (p : s_IOLTS) :
      ind_one_step_reachability s l si p.(iolts).(sc_lts).(lts) →
      ind_s_seq_reachability si t s' p →
      ind_s_seq_reachability s (s_event l :: t) s' p
  | s_seq_reachability_r3 (s s' : state) (t : list s_label) (p : s_IOLTS) :
      In s p.(Ts) →
      ind_s_seq_reachability s t s' p →
      ind_s_seq_reachability s (delta :: t) s' p.

Definition ind_s_traces (s : state) (t : list s_label) (p : s_IOLTS) : Prop :=
  ∃ (s' : state), ind_s_seq_reachability s t s' p.
```

A definição *ind_s_seq_reachability* adapta a definição *ind_seq_reachability*, apenas incluindo o rótulo δ. Como as transições que possuem o rótulo δ representam transições que partem de um estado voltando para ele mesmo, é possível afirmar que estas duas definições são equivalentes. Ou seja, os estados alcançáveis a partir de um *s_trace ll*, em um *s_IOLTS*, são os mesmos alcançáveis no *IOLTS* equivalente a partir do *trace* dado pela remoção dos rótulos δ de *ll*. Esta afirmação foi proposta no lema *s_seq_reachbility_eqv_seq_reachbility* e provada correta.

Do mesmo modo, pode-se afirmar que um *s_trace* é equivalente a um *trace* retirando-se as transições delta. Esta afirmação foi proposta no lema *s_trace_eqv_trace* e também provada correta. O trecho de código a seguir mostra o enunciado destes dois lemas.

```
Lemma s_seq_reachability_eqv_seq_reachability :
  ∀ (s s': state) (l: list s_label) (p : s_IOLTS),
    ind_s_seq_reachability s l s' p →
    ind_seq_reachability s (s_trace_without_delta l) s' p.(iolts).(sc_lts).(lts).

Lemma s_trace_eqv_trace :
  ∀ (s : state) (l: list s_label) (p : s_IOLTS),
    ind_s_traces s l p →
    ind_traces s (s_trace_without_delta l) p.(iolts).(sc_lts).(lts).
```

3.1.3 IOCO.v

Como visto na Seção 2.2.3, uma implementação i é dita **ioco**-conforme uma especificação s se, para todo *Strace* da especificação, o conjunto de saídas observadas em i após a execução deste mesmo *Strace* é um subconjunto das saídas previstas em s.

A definição *out*, também descrita na Seção 2.2.3, utiliza *s_traces*. Dessa forma, a formalização proposta anteriormente por Sobral (2019) precisou ser remodelada para usar o novo tipo *s_label*. A nova função *f_out* (definição funcional em Coq de *out*) usa

como funções auxiliares *f_out_one_state* e *f_out_one_state'*. Esta última retorna os rótulos de saída que partem do estado *s* passado como argumento. Já a função *f_out_one_state* chama a função anterior, passando os argumentos corretos considerando um dado *s_IOLTS* recebido como argumento.

```
Fixpoint f_out_one_state' (s : state) (lt: list transition) (Lu : set label)
    : set s_label :=
  match lt with
  | [] ⇒ []
  | (q1, e, q2) :: t ⇒
      if s =? q1
      then
        match e with
        | event e' ⇒
            if set_mem string_dec e' Lu
            then set_add s_label_dec (s_event e') (f_out_one_state' s t Lu)
            else f_out_one_state' s t Lu
        | tau ⇒ f_out_one_state' s t Lu
        end
     else f_out_one_state' s t Lu
  end.

Definition f_out_one_state (s : state) (p : s_IOLTS) : set s_label :=
  f_out_one_state' s p.(iolts).(sc_lts).(lts).(T) p.(iolts).(L_u).
```

Por fim, a função *f_out* utiliza *f_out_one_state* para calcular o conjunto de saídas de um conjunto de estados. Esta função é definida via casamento de padrões sobre a lista de estados *ls*. Se *ls* não for vazia (*ls* possui um primeiro elemento *h* seguido de uma lista *t* – $ls = h :: t$), *set_member* verifica se se o estado *h* é um elemento/membro da lista de estados quiescentes *Ts*. Sendo este o caso, a saída o estado *h* será δ. Caso contrário, a função *out* considera o conjunto de rótulos de saídas retornado por *f_out_one_state*.

```
Fixpoint f_out (ls : set state) (p : s_IOLTS) : set s_label :=
  match ls with
  | [] ⇒ []
  | h :: t ⇒
      if set_mem Nat.eq_dec h p.(Ts)
      then set_add s_label_dec delta (f_out t p)
      else set_union s_label_dec (f_out_one_state h p) (f_out t p)
  end.
```

Além da definição funcional de *out*, uma definição indutiva foi criada. Seguindo os mesmos passos da definição funcional, *ind_out_one_state* caracteriza o conjunto de saídas

para um único estado e *ind_out*, para um conjunto de estados.

```
Inductive ind_out_one_state : state → set s_label → s_IOLTS → Prop :=
  | out_one_state_r1 (s : state) (p : s_IOLTS) :
      In s p.(Ts) → ind_out_one_state s [delta] p
  | out_one_state_r2 (s : state) (p : s_IOLTS) (so : set s_label) :
      ¬ In s p.(Ts) →
      ¬ In delta so →
      (∀ (l : label),
        In (s_event l) so ↔
        ∃ (s' : state),
          In l p.(iolts).(L_u) ∧ ind_transition s (event l) s' p.(iolts).(sc_lts).(lts)) →
      ind_out_one_state s so p.
```

A definição *ind_out_one_state* possui duas regras: a primeira regra descreve o caso especial em que o estado é quiescente e não haverá rótulos de saída, apenas o rótulo *delta*; a segunda descreve os casos onde o estado possui rótulos de saída partindo dele. Sendo so o conjunto de rótulos de saída do estado s, um label l está em so se, e somente se, existe um estado s' que é alcançado a partir de s realizando a transição rotulada por l, considerando que l é um rótulo de saída.

```
Definition ind_out (Q : set state) (so : set s_label) (p : s_IOLTS) : Prop :=
  (∀ (x : s_label),
    (In x so → ∃ (s : state) (so' : set s_label),
                  (In s Q ∧ ind_out_one_state s so' p ∧ In x so')))
  ∧
  (∀ (s : state), In s Q →
    ∃ (so' : set s_label), ind_out_one_state s so' p ∧
      ∀ (o : s_label), In o so' → In o so).
```

A definição *ind_out* define uma relação entre um conjunto de estados Q e um conjuntos de rótulos so, de maneira que todo rótulo em so é rótulo de saída de ao menos um estado em Q e os rótulos de saída de todos estados de Q estão em so.

O teorema *ind_out_equiv_f_out*, o qual propõe que a definição funcional reflete (é semanticamente equivalente) a definição indutiva, foi provado. O enunciado deste teorema é mostrado a seguir.

```
Lemma ind_out_equiv_f_out:
  ∀ (Q : set state) (so : set s_label) (p : s_IOLTS),
    ind_out Q so p ↔ so [=] f_out Q p.
```

A descrição de *after* foi reelaborada para determinar os estados alcançáveis a partir de um estado s após realizar as transições denotadas por ll (considerando também transições

de rótulo δ) de um *s_IOLTS*. A nova definição funcional é chamada de *f_after_IOLTS* e a nova definição indutiva é chamada de *ind_s_after_IOLTS*.

Por fim, uma nova caracterização indutiva da relação de conformidade **ioco** (*ind_ioco*) foi desenvolvida, utilizando as definições *ind_out* e *ind_s_after_IOLTS*. Seja i um IOTS (um IOLTS *input-enabled*), seja s um IOLTS, diz-se que i é **ioco**-conforme s se, e somente se, para todo *trace* t de s (anotado com quiescência), as saídas observadas em i após t (out_i – obtidas considerando as saídas dos estados Qi alcançados em i após t) estiverem contidas nas saídas observadas em s após o mesmo t (out_s – obtidas considerando as saídas dos estados Qs alcaçados em s após t).

```
Definition ind_ioco (i : IOTS) (s : IOLTS) : Prop :=
  ∀ (Qi Qs : set state) (t : list s_label) (out_i out_s : set s_label),
    ind_s_traces_LTS t (create_s_IOLTS s) →
    ind_s_after_IOLTS t Qi (create_s_IOLTS i.(embedded_iolts)) →
    ind_s_after_IOLTS t Qs (create_s_IOLTS s) →
    ind_out Qi out_i (create_s_IOLTS i.(embedded_iolts)) →
    ind_out Qs out_s (create_s_IOLTS s) →
    incl out_i out_s.
```

Para ilustrar e validar a definição *ind_ioco*, a implementação i1, apresentada na Figura 8a, foi descrita em Coq através do tipo IOTS. As especificações $s1$ e $s3$ apresentadas, respectivamente, nas Figuras 8b e 8c foram descritas em Coq através do tipo IOLTS. Com estas descrições e a definição indutiva *ind_ioco*, é possível provar, em Coq, que $i1$ **ioco** $s1$ e que $i1$ $\neg$**ioco** $s3$, como explicado na Seção 2.2.3. Abaixo, têm-se os enunciados destes exemplos em Coq. O corpo das provas não é apresentado aqui, mas está disponível no repositório do projeto.

```
Example i1_ioco_s1 : ind_ioco imp_i1 spec_s1_IOLTS.

Example i1_not_ioco_s3 : ¬ (ind_ioco imp_i1 spec_s3_IOLTS).
```

3.2 Definição de testes para **ioco**

Para definir em Coq a noção de testes para **ioco**, é necessário formalizar dois conceitos: primeiro, o que é um caso de teste (Seção 3.2.1) e como se dá a execução de um teste (Seção 3.2.2).

3.2.1 Casos de teste

O *Record* mostrado no trecho de código a seguir formaliza a noção de um caso de teste (TTS) em Coq, seguindo as definições apresentadas na Seção 2.2.4.

```
Record TTS : Type := mkTTS {
```

```
    iots : IOTS
  ; fail_state : state
  ; pass_state : state
  ; theta : label
  ; fail_is_valid : In fail_state iots.(embedded_iolts).(sc_lts).(lts).(Q)
  ; pass_is_valid : In pass_state iots.(embedded_iolts).(sc_lts).(lts).(Q)
  ; theta_is_valid : In theta iots.(embedded_iolts).(L_u)
  ; imp_Li := set_remove string_dec theta iots.(embedded_iolts).(L_u)
  ; imp_Lu := iots.(embedded_iolts).(L_i)
  ; is_deterministic : ind_deterministic iots.(embedded_iolts).(sc_lts).(lts)
  ; pass_fail_diff : pass_state ≠ fail_state
  ; pass_cycle :
      ∀ (l : label) (s : state),
        ind_transition pass_state (event l) s iots.(embedded_iolts).(sc_lts).(lts) →
        s = pass_state ∧ (l = theta ∨ In l imp_Lu)
  ; fail_cycle :
      ∀ (l : label) (s : state),
        ind_transition fail_state (event l) s iots.(embedded_iolts).(sc_lts).(lts) →
        s = fail_state ∧ (l = theta ∨ In l imp_Lu)
  ; no_other_cycles :
      ∀ (t : list label) (s : state),
        t ≠ [] →
        ind_seq_reachability s t s iots.(embedded_iolts).(sc_lts).(lts) →
        s = fail_state ∨ s = pass_state
  ; valid_out_label :
      ∀ (q : state),
        In q iots.(embedded_iolts).(sc_lts).(lts).(Q) →
        f_init q iots.(embedded_iolts).(sc_lts).(lts) [=]
          map event (theta :: imp_Lu) ∨
        ∃ (a : label),
          In a imp_Li ∧
          f_init q iots.(embedded_iolts).(sc_lts).(lts) [=]
                map event (a :: imp_Lu)
}.
```

O *Record* TTS recebe como argumento um *iots* e quais valores representam o rótulo especial θ, o estado **fail** e e o estado **pass** neste *iots*. As propriedade *fail_is_valid*, *pass_is_valid* e *theta_is_valid* verificam se estes argumentos são válidos, i.e. se eles de fato fazem parte do *iots* recebido. Os valores *imp_Li* e *imp_Lu* representam, respectivamente, o conjunto de rótulos de entrada e de saída da implementação a ser testada. Do ponto

de vista do teste, as saídas da implementação são entradas para o teste, enquanto que as entradas da implementação são saídas do teste. Contudo, para evitar confusão, para fins de terminologia, considera-se a implementação como a referência. Portanto, assim, estes conjuntos são referenciados como entradas e saídas da implementação. As demais propriedades formalizam as restrições de um TTS, definidas por Tretmans (2008) e discutidas na Seção 2.2.4:

- um caso de teste deve ser determinístico (*is_deterministic*);
- o estado **fail** é diferente do estado **pass** (*pass_fail_diff*);
- todas as transições que partem dos estados **pass** e **fail** voltam para si mesmos (*pass_cycle* e *fail_cycle*, respectivamente);
- um caso de teste não possui outros ciclos (*no_other_cycles*);
- para todo estado q do caso de teste, os rótulos das transições partindo de q são iguais a $\{\theta\} \cup L_U$ ou a $\{a\} \cup L_U$, para um dado $a \in L_I$ (*valid_out_label*).

Uma tática que auxilia na criação de um TTS foi definida, como mostra o trecho de código abaixo. Dado o tamanho desta tática, abaixo, mostra-se apenas a assinatura da mesma. A definição completa está disponível no repositório do projeto. A partir dos argumento recebidos (o IOTS usado para a construção deste TTS, os estados *pass_state* e *fail_state* e o rótulo especial *theta*) é possível criar uma nova instância de TTS, provando todas as propriedades de boa formação automaticamente.

```
Ltac create_TTS iots pass_state fail_state theta.
```

Desta forma, criar a instância *fig7_t1_TTS* se torna algo bem mais simples. Esta instância utiliza a definição *fig7_t1_IOTS*, o estado 0 como o *pass_state*, o estado 4 como o *fail_state* e a *string* "theta" para representar o rótulo especial. A criação desta instância é mostrada abaixo, onde *fig7_t1_IOTS* é a definição em Coq correspondente ao IOTS da Figura 9.

```
Definition fig7_t1_TTS : TTS.
Proof.
  create_TTS fig7_t1_IOTS 0 4 "theta".
Defined.
```

3.2.2 Execução de testes

Para definir a execução de testes em Coq, primeiramente é preciso formalizar o operador $\rceil\!|$ e as suas regras de inferência, apresentadas na Figura 10. Estas foram caracterizadas através do tipo indutivo *ind_test_execution_transition*, apresentado no

trecho de código abaixo. Na definição, cada construtor representa uma das regras de inferência.

```
Inductive ind_test_execution_transition :
    state → state → transition_label → state → state → TTS → IOTS → Prop :=
  | test_execution_transition_r1 (t i i' : state) (test : TTS) (imp : IOTS) :
      ind_transition i tau i' imp.(embedded_iolts).(sc_lts).(lts) →
      ind_test_execution_transition t i tau t i' test imp
  | test_execution_transition_r2 (t i t' i' : state) (a : label) (test : TTS)
      (imp : IOTS) :
      In a (test.(imp_Li) ++ test.(imp_Lu)) →
      ind_transition t (event a) t' test.(iots).(embedded_iolts).(sc_lts).(lts) →
      ind_transition i (event a) i' imp.(embedded_iolts).(sc_lts).(lts) →
      ind_test_execution_transition t i (event a) t' i' test imp
  | test_execution_transition_r3 (t i t' : state) (theta_l : label) (test : TTS)
      (imp : IOTS) :
      theta_l = test.(theta) →
      ind_transition t (event theta_l) t' test.(iots).(embedded_iolts).(sc_lts).(lts) →
      In i (create_s_IOLTS imp.(embedded_iolts)).(Ts) →
      ind_test_execution_transition t i (event theta_l) t' i test imp.
```

Se existir uma transição interna (rotulada com τ) na implementação, a execução do teste evolui mantendo o estado do teste, mas atualizando o estado da implementação a partir da transição interna (*test_execution_transition_r1*). Se existir uma transição rotulada com $a \in L_I \cup L_U$ tanto na implementação como no caso de teste, a execução do teste evolui sincronizando na realização destas transições, alcançando os estados destinos destas transições (*test_execution_transition_r2*). Por fim, se a implementação estiver em um estado quiescente, o teste, ao observar esta quiescência, evolui para o estado alcançado pela sua transição rotulada com θ (*test_execution_transition_r3*).

Dando sequência, a definição *ind_test_run* formaliza o conceito de execução de um teste, apresentado na Definição 19: uma execução é caracterizada por um *trace* de $t \rceil\!| i$ que leva o teste ao estado de **pass** ou **fail**. Para isto, algumas definições auxiliares precisam ser criadas.

```
Definition ind_test_run (sigma : list label) (test : TTS) (imp : IOTS) : Prop :=
  ∃ (i' : state),
    ind_test_execution_trace sigma test.(pass_state) i' test imp ∨
    ind_test_execution_trace sigma test.(fail_state) i' test imp.
```

A definição *ind_test_execution_trace* caracteriza o que seria um *trace* válido para uma execução de teste: uma sequência de rótulos *ll* que leva o teste e a implementação, a partir dos respectivos estados iniciais $q0$, aos estados $final\_t$ e $final\_i$, respectivamente.

```
Definition ind_test_execution_trace (ll : list label) (final_t final_i : state)
    (test : TTS) (imp : IOTS) : Prop :=
  ind_test_execution_seq_reachability
    test.(iots).(embedded_iolts).(sc_lts).(lts).(q0)
    imp.(embedded_iolts).(sc_lts).(lts).(q0)
    ll final_t final_i test imp.
```

Esta última definição, por sua vez, baseia-se em *ind_test_execution_seq_reachability*, que, de maneira semelhante às definições *ind_seq_reachability* e *ind_s_seq_rechability*, define os estados que são alcançáveis a partir de um estado inicial seguindo uma dada sequência de rótulos.

```
Inductive ind_test_execution_seq_reachability :
    state → state → list label → state → state → TTS → IOTS → Prop :=
  | test_execution_seq_reachability_r1 (t1 t2 i1 i2 : state) (test : TTS) (imp : IOTS) :
      ind_test_execution_empty_reachability t1 i1 t2 i2 test imp →
      ind_test_execution_seq_reachability t1 i1 [] t2 i2 test imp
  | test_execution_seq_reachability_r2 (t1 i1 t2 i2 t3 i3 : state) (l : label)
        (ll : list label) (test : TTS) (imp : IOTS) :
      ind_test_execution_one_step_reachability t1 i1 l t2 i2 test imp →
      ind_test_execution_seq_reachability t2 i2 ll t3 i3 test imp →
      ind_test_execution_seq_reachability t1 i1 (l :: ll) t3 i3 test imp.
```

A definição *ind_test_execution_empty_reachability* caracteriza os estados que são alcançáveis a partir de um estado inicial, seguindo apenas transições de rótulo τ.

```
Inductive ind_test_execution_empty_reachability :
    state → state → state → state → TTS → IOTS → Prop :=
  | test_execution_empty_reachability_r1 (t i : state) (test: TTS) (imp : IOTS) :
      ind_test_execution_empty_reachability t i t i test imp
  | test_execution_empty_reachability_r2 (t1 t2 t3 i1 i2 i3 : state) (test : TTS)
        (imp : IOTS) :
      ind_test_execution_transition t1 i1 tau t2 i2 test imp →
      ind_test_execution_empty_reachability t2 i2 t3 i3 test imp →
      ind_test_execution_empty_reachability t1 i1 t3 i3 test imp.
```

Já a definição *ind_test_execution_one_step_reachability* caracteriza os estados que são alcançáveis a partir de um estado inicial seguindo uma transição com o rótulo dado, mas podendo também utilizar transições internas.

```
Inductive ind_test_execution_one_step_reachability :
    state → state → label → state → state → TTS → IOTS → Prop :=
  | test_execution_one_step_reachability_r1 (t1 t2 t3 i1 i2 i3 : state)
```

```
    (l : label) (test : TTS) (imp : IOTS) :
  ind_test_execution_empty_reachability t1 i1 t2 i2 test imp →
  ind_test_execution_transition t2 i2 (event l) t3 i3 test imp →
  ind_test_execution_one_step_reachability t1 i1 l t3 i3 test imp.
```

A definição *ind_passes* formaliza quando uma implementação passa um caso de teste, assim como descrito pela Definição 20: uma implementação passa um caso de teste se, e somente se, para toda execução possível do teste o estado alcançado nunca é o de falha.

```
Definition ind_passes (imp : IOTS) (test : TTS) : Prop :=
  ∀ (ll : list label) (i' : state),
    ind_test_run ll test imp →
    ¬ ind_test_execution_trace ll test.(fail_state) i' test imp.
```

Por fim, as definições *ind_passes_set* e *ind_fails_set* definem quando uma implementação passa ou falha um conjunto (suíte) de testes, respectivamente. Para passar um conjunto de testes, a implementação precisa passar todos os casos de teste que compõem a suíte. Para falhar, é suficiente existir um teste na suíte para o qual a implementação não passa.

```
Inductive ind_passes_set : IOTS → set TTS → Prop :=
  | passes_set_r1 (imp : IOTS) :
      ind_passes_set imp []
  | passes_set_r2 (test : TTS) (tests : set TTS) (imp : IOTS) :
      ind_passes imp test →
      ind_passes_set imp tests →
      ind_passes_set imp (test :: tests).

Definition ind_fails_set (imp : IOTS) (tests : set TTS) : Prop :=
  ∃ (test : TTS), In test tests ∧ ¬ ind_passes imp test.
```

As definições $k1$ e $k3$, apresentadas nas Figuras 11 e 5, respectivamente, foram descritas em Coq através do tipo IOTS. Com estas descrições e as definições apresentadas nesta seção, é possível provar, em Coq, que $k1$ **passes** $t1$ e que $k3$ $\neg$**passes** $t1$, como explicado na Seção 2.2.5. A seguir, têm-se o enunciado destes exemplos (provas). O detalhamento de cada prova está disponível no repositório do projeto.

```
Example k1_passes_t1 : ind_passes fig4_k1_IOTS fig7_t1_TTS.

Example k3_fails_t1 : ¬ ind_passes fig4_k3_IOTS fig7_t1_TTS.
```

3.3 Representação visual com Graphviz

Graphviz é um software *open source* para visualização de grafos. Com ele é possível criar diagramas em formatos úteis, como SVG e JPEG, através de uma descrição de grafos feita em uma linguagem textual. A linguagem utilizada para descrever os grafos é chamada de DOT. Ela possui vários recursos para customizações dos diagramas, como, por exemplo, opções de cores, fontes e formas personalizadas.

Com a finalidade de permitir visualizar as estruturas definidas neste trabalho (LTS, IOLTS, entre outras), criou-se uma integração com o Graphviz, descrita a seguir. O primeiro passo para representar graficamente as estruturas do trabalho é descrever cada estrutura na linguagem DOT. Para isto, foram definidas funções que auxiliam neste processo e convertem a instância de uma estrutura em uma *string* que a representa em DOT. De posse da *string*, é possível salvá-la em um arquivo, por exemplo *IOLTS.gv*, e com a ferramenta Graphviz instalada executar o comando abaixo.

```
dot -Nlabel="" -Nshape=circle -Tjpeg IOLTS.gv -o IOLTS.jpeg
```

As transições de uma estrutura são descritas em DOT de acordo com o seguinte padrão "*<P> -> <Q> [label=r];*", onde P é uma representação do estado de partida da transição, Q é uma representação do estado de destino e r é o rótulo da transição. A função genérica (polimórfica) *generate_dot*, mostrada no trecho de código a seguir, converte um lista de transições na sua representação em DOT.

```
Fixpoint generate_dot {State Event : Type} (lts : set (State × Event × State))
  (state_to_str : State → string) (event_to_str : Event → string) : string :=
  match lts with
  | nil ⇒
  | (P, e, Q) :: tl ⇒
   " <" ++ state_to_str P ++ "> -> <" ++ state_to_str Q ++ ">" ++
   " [label=<" ++ event_to_str e ++ ">];" ++ (generate_dot tl state_to_str event_to_str)
  end.
```

Nesta função os estados e rótulos são representados por tipos parametrizados *State* e *Event*, respectivamente. Isto permite aplicá-la no contexto de diferentes estruturas. Por exemplo, o rótulo δ só existe no contexto de estruturas anotadas com quiescência. Além da lista de transições a ser convertida, esta função recebe duas funções como argumento: *state_to_str* e *event_to_str* que convertem, respectivamente, um estado e um rótulo para uma *string*.

A função genérica *style_initial_state*, apresentada no trecho de código a seguir, foi criada com o intuito de dar um destaque ao estado inicial do estrutura. Nela, defini-se que o estado inicial será destacado por borda em negrito na cor vermelha.

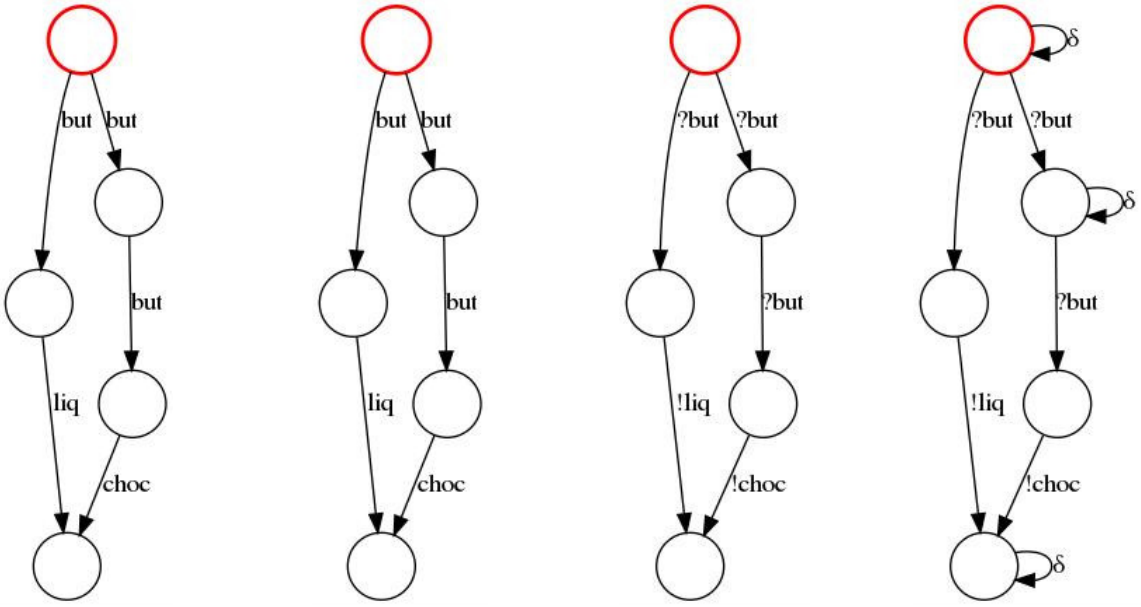
```
Definition style_initial_state {State : Type} (q0 : State)
  (state_to_str : State → string) : string :=
      "<" ++ state_to_str q0 ++ "> [style=bold, color=red];".
```

Assim como a função anterior, *style_initial_state* representa o estado através do tipo parametrizado *State* e recebe como argumento uma função que converte um estado em sua representação textual.

Adicionalmente, foram criadas funções que utilizam estas duas funções genéricas para converter cada um dos tipos descritos neste trabalho em suas representações equivalentes na linguagem DOT. As funções *generate_dot_behaviour_expressions*, *generate_dot_lts*, *generate_dot_IOLTS*, *generate_dot_s_IOLTS* e *generate_dot_TTS* foram criadas para converter instâncias do tipo *Behaviour Expression*, *LTS*, *IOLTS*, *s_IOLTS* e *TTS*, respectivamente. Destas, a primeira função tem a sua recursão limitada por um número natural, pois, a partir de uma *Behaviour Expression* é possível expressar um LTS com uma quantidade infinita de estados.

Executando os comandos abaixo, com o apoio do Graphviz, obtém-se a representação gráfica de *fig2_r_BE*, *fig2_r*, *fig2_r_IOLTS* e *fig6_r*, como apresentado na Figura 13.

```
Compute (generate_dot_behaviour_expressions "fig2_r" fig2_r_BE 6).
Compute (generate_dot_lts fig2_r).
Compute (generate_dot_IOLTS fig2_r_IOLTS).
Compute (generate_dot_s_IOLTS fig6_r).
```



(a) fig2_r_BE (b) fig2_r (c) fig2_r_IOLTS (d) fig6_r_s_IOLTS

Figura 13 – Representações gráficas obtidas via Graphviz.

De forma similar, com o resultado da chamada à função *generate_dot_TTS*, é possível usar o Graphviz para obter uma representação gráfica (ver a Figura 14) da

formalização em Coq do TTS $t1$, apresentado anteriormente na Figura 9.

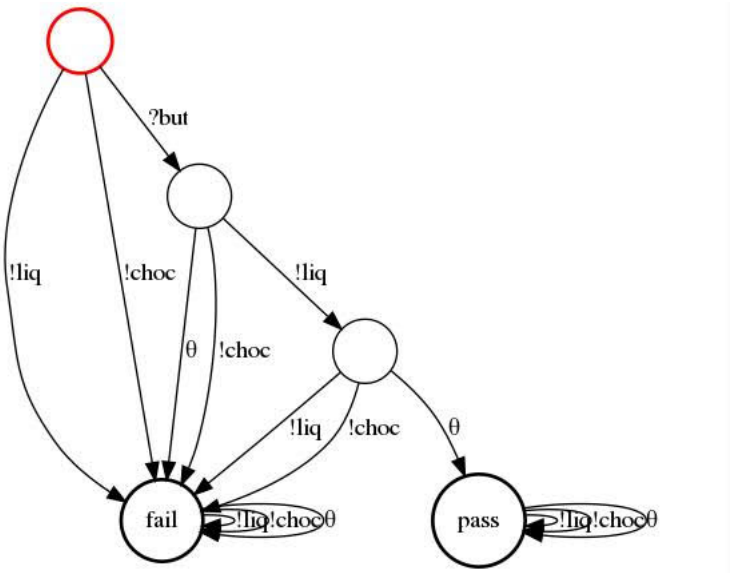


Figura 14 – Representação gráfica de $t1$ obtida via Graphviz.

É importante destacar que, diferente dos exemplos apresentados por Tretmans (2008) (Figuras 3, 6 e 9), as representações gráficas geradas convergem para menos estados (na parte inferior). Isso acontece porque a representação formal em Coq unifica estados que descrevem o mesmo comportamento; no caso da Figura 13, STOP; e, no caso da Figura 14, os estados **fail** e **pass**.

4 Conclusão

Este trabalho realizou um estudo da teoria formal de testes proposta por Tretmans (2008), além de uma análise detalhada dos resultados alcançados nos trabalhos anteriores de Sobral (2019) e Santana (2020), que formalizaram partes desta teoria em Coq.

Através desta análise, foram realizadas várias modificações em definições previamente formalizadas, além de correções, que possibilitaram a incorporação dos resultados de Sobral (2019) aos resultados de Santana (2020). Dentre estas modificações, destaca-se a reformulação das definições de um LTS com entradas e saídas e da relação de conformidade **ioco**.

Adicionalmente, este trabalho foi além e formalizou o conceito de casos de teste, juntamente com as definições das regras que caracterizam a execução de um caso de teste. Desta forma, agora é possível enunciar e provar, em Coq, se implementações passam ou falham testes, de acordo com teoria de testes baseada em **ioco**.

Por fim, foram também criadas funções que, com auxílio da ferramenta Graphviz, fornecem representações visuais de um LTS, de um LTS obtido automaticamente a partir de *Behaviour Expressions*, de um LTS com entradas e saídas (IOLTS), de um IOLTS anotado com quiescência, assim como de um TTS.

4.1 Trabalhos futuros

Outras atividades, não realizadas neste trabalho, podem trazer resultados importantes e complementar o projeto de revisitar em Coq a teoria formal de testes baseada em **ioco**. Dentre estas, destacam-se as listadas abaixo.

- Criar mais táticas de automação de prova. Táticas personalizadas são importantes pois auxiliam e facilitam a prova de teoremas. Apesar de algumas terem sido criadas neste trabalho, seria oportuno a criação de mais táticas, por exemplo, que auxiliam a demonstrar que um LTS é fortemente convergente ou que uma implementação é **ioco**-conforme uma especificação.
- Provar a equivalência das definições funcionais e indutivas. Apesar de algumas provas já terem sido realizadas, como *ind_init_reflect*, outras equivalências ainda precisam ser demonstradas. Por exemplo, este trabalho apenas inclui o enunciado de *ind_after_reflect*, sem realizar a sua prova.
- Implementar o algoritmo de geração de testes. Esta é uma parte importante da

teoria de Tretmans (2008) que ainda não foi contemplada nem por este, nem pelos trabalhos anteriores.

- Provar em Coq que a teoria de testes proposta por Tretmans (2008) é correta. O artigo que define a teoria apresenta apenas uma prova informal (textual) de que a teoria de testes baseada em **ioco** é correta: se uma implementação não passa um teste gerado a partir de uma dada especificação, então esta implementação não é **ioco**-conforme a especificação em questão.

- Desenvolver uma GUI integrando todos os resultados obtidos. Considera-se aqui a criação de editores que permitam modelar especificações e implementações, a partir de *Behaviour Expressions* (editor textual) ou diretamente como sistemas de transições rotuladas (editor gráfico). Adicionalmente, pela GUI, deve ser possível verificar conformidade de acordo com **ioco**, assim como gerar e executar casos de teste.

Referências

BERTOT, Y.; CASTRAN, P. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions*. 1st. ed. [S.l.]: Springer Publishing Company, Incorporated, 2010. ISBN 3642058809, 9783642058806.

CAVALCANTI, A.; HIERONS, R. M.; NOGUEIRA, S. Inputs and outputs in csp: A model and a testing theory. *ACM Trans. Comput. Logic*, Association for Computing Machinery, New York, NY, USA, v. 21, n. 3, may 2020. ISSN 1529-3785.

COQUAND, T.; HUET, G. The calculus of constructions. *Information and Computation*, v. 76, n. 2, p. 95–120, 1988. ISSN 0890-5401. Disponível em: <https://www.sciencedirect.com/science/article/pii/0890540188900053>.

ELLSON, J. et al. Graphviz—open source graph drawing tools. In: SPRINGER. *International Symposium on Graph Drawing*. [S.l.], 2001. p. 483–484.

GAUDEL, M.-C. Testing can be formal, too. In: MOSSES, P. D.; NIELSEN, M.; SCHWARTZBACH, M. I. (Ed.). *TAPSOFT '95: Theory and Practice of Software Development*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995. p. 82–96. ISBN 978-3-540-49233-7.

RTI. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, p. 1, 2002.

SANTANA, L. V. da C. *Formalização em Coq de uma linguagem de processos para ioco*. 2020. Trabalho de Conclusão de Curso, Centro de Informática, Universidade Federal de Pernambuco.

SOBRAL, F. N. P. *Uma Formalização em Coq de Testes Baseados em Modelos*. 2019. Trabalho de Conclusão de Curso, Centro de Informática, Universidade Federal de Pernambuco.

TRETMANS, J. Model based testing with labelled transition systems. In: *Formal methods and testing*. [S.l.]: Springer, 2008. p. 1–38.