MASTER IN COMPUTER SCIENCE

High-Level Modelling and Formal Semantics of Product-Line Behaviour

Jeanjot, Arno

Award date:
2013

Awarding institution:
University of Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Faculté d'Informatique
Année académique 2012–2013

High-Level Modelling and Formal Semantics of Product-Line Behaviour

Arno Jeanjot

Maître de stage : Mats Heimdahl

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Pierre-Yves Schobbens

Co-promoteur : Maxime Cordy

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Abstract

Software Product Line (SPL) methods have become more and more popular. They have been applied on several domains including critical systems which require quality assurance techniques in order to avoid error. An established quality assurance technique is model checking that verifies if a system under certain consideration holds certain properties. However, an SPL may potentially be compound of thousands of products and verify each product of a line at a time would be suboptimal. To avoid this problem, there is a formalism called Featured Transition System (FTS) that allows merging the behaviour of every product of a line in a single model and verifying this model by using algorithms specially defined for this formalism. Unfortunately, model checking techniques are not popular in industry. Indeed, FTS has its own language that requires some expertise to be fully understandable. This thesis aims to bridge the gap between FTS model checking and the industry by defining a variability-aware extension of a state-based language commonly used in industry. The semantics of this extension will be defined in terms of FTS which allow the use FTS model checking techniques on systems modelled in this language.

Résumé

Les méthodes Lignes de Produits Logiciels (SPL) sont devenues de plus en plus populaire. Elles ont été appliquées dans différents domaines incluant les systèmes critiques qui nécessitent des techniques d'assurance qualité afin d'éviter toute erreur. Cependant, une ligne de produits logiciels pouvant potentiellement contenir des milliers de produits différents, il serait inefficace de vérifier chaque produit un à un. Pour éviter ce problème, il existe un formalisme nommé *featured transition system*(FTS) qui permet de fusionner le comportement de chaque produit d'une ligne dans un seul modèle et de vérifier ce modèle grâce à des algorithmes définis spécifiquement pour ce formalisme. Malheureusement, l'utilisation de ce formalisme, et des techniques de vérifications de modèles en général, n'est pas encore répandue dans le monde industriel. En effet, FTS possède son propre langage et demande donc un effort d'apprentissage supplémentaire avant de pouvoir être utilisé. Ce mémoire a pour but de réduire l'écart entre les techniques de FTS model checking et l'industrie en définissant une extension d'un langage de modélisation populaire dans l'industrie permettant l'intégration de la variabilité. La sémantique de ce langage sera défini en terme de FTS ce qui permettre l'utilisation des technique de FTS model checking sur les systèmes modélisés à l'aide de ce langage.

Acknowledgements

The work described in this thesis is the result of a three months internship at the university of Minnesota in Minneapolis, United States, where I was part of the Software Engineering team and was supervised by Dr. Mats Heimdahl.

I would like to thank Dr. Pierre-Yves Schobbens for giving me the opportunity of doing this intership.

I would like to thank Maxime Cordy for his availability and his help in writing this thesis.

I would like to thank Dr. Mats Heimdahl for his supervision and his help in orienting me in the research process during the internship.

I would like to thank all the CriSys Group, and in particular, Ian De Silva, for their help during the internship.

Contents

List of Figures

List of Tables

List of Definitions

Chapter 1

Introduction

Nowadays, software takes such an important place in our daily lives that it would be unthinkable to live without them. Indeed, software is everywhere: at work, at school, at home, in smartphones, in cars, in airplanes, in banks, in shops ... The majority of our tasks and actions is managed by software.

Even though computer science is a really young science compared to physics or philosophy, its development is unparalleled. Thanks to a fast evolution of technologies, computers have become smaller, faster, more efficient and less expensive. This offers more possibilities to develop software that could achieve more complex tasks.

As a direct consequence of these evolutions, demand in software rapidly increased and companies had to develop more products in shorter periods of time. New development methods were thus required in order to meet this demand. Software reuse between products, that share common requirements, would result in saving a considerable amount of time and thus to be a good optimisation.

The desire of a systematic, planned and controlled software reuse during each development phase gave birth to software engineering methods called Software Product Line (SPL). These methods consist in developing, not a single software, but a whole set of different software products that share common requirements. Benefits of applying SPL methods to develop a set of products are economies of scale and a reduced time-to-market. [CN01]

A good analogy of this is the manufactured production of a line of cars. Cars of a same line share commonalities (such as wheels, tires, engine, etc) and each particular car has a set of options (such as ABS, sunroof, air conditioning, etc)

that differs from the other cars. In practice, the commonalities is assembled first, then some optional assets are added to the car in order to correspond with the set of options.

SPL are developed in various domains including critical systems such as airplane controls, bank transactions or pacemakers. In these kind of systems, an error may result in a loss of money and sometimes in a loss of human lives and this must not happen. It is thus essential to prevent these problems by using quality assurance techniques.

Model checking is an established quality assurance technique that verifies whether a system under certain consideration holds certain properties [GV08]. On the one hand, the system is modelled in a Transition System (TS) that consists of states and transitions. On the other hand, properties are expressed in Linear Temporal Logic (LTL) to specify the admissible behaviour of the system. Using the model of the system and the set of properties, the model checker explores all possible executing paths of the model in a brute-force manner and checks whether each path satisfies the properties. This technique presents the advantage to report an error if and only if there exists at least one in the model. Moreover, this error is represented by a path that leads the model to the violation of the property. In spite of its exhaustiveness, given that the model checker verifies a model of the system and not the system itself, it cannot guarantee the absence of errors in the actual system if the model is incorrect.

As mentioned above, we do not have a single product to verify but a set of products and depending on the variability, the size of this set can be huge. It would be too tedious to verify one product at a time. Since products possess commonalities, it is suboptimal to verify each product separately. In order to avoid model checking of each product, the Featured Transition System (FTS) was proposed. This formalism is designed to model the combined behaviour of a whole SPL. Basically, an FTS is a TS in which transitions are labelled with constraints over the features of an SPL, that is, a transition labelled with a given feature requires the presence of this feature to be available. Given that model checking techniques are not feature aware, they have been adapted to FTS.

Several FTS model checkers based on different languages have been implemented in the past three years. Similarly to traditional model checker, these explore all possible executing paths of a given model and check the satisfaction of given properties but in this case, a whole set of products is verified. A given property may be violated by only a subset of products whereas the others hold it. Therefore, in

addition to the violating path, FTS model checkers provide the subset of products that do not satisfy the property.

Although early experiments showed that FTS-based model checking is more efficient than a product-by-product application of single-system model checking, these techniques have not reached industry yet. Indeed, tools that implement these techniques rely on ad-hoc specification languages born from academic research. An expertise is thus required to be able to use one of these languages in a correct and efficient way. Industries do not want to waste time and money for such an expertise and therefore, FTS model checking is not used. This is an obstacle to the development of the FTS formalism.

The objective of this thesis is to bridge the gap between the FTS formalism and engineers. To achieve this, we propose a variability-aware behavioural modelling language based on Stateflow [Mat04], a state-based language commonly used in industry. We define the semantics of this language in terms of FTS, which allows us to use FTS model checking techniques on systems modelled in this language. We implemented our approach as part of ProVeLines, the latest incarnation of FTS model checkers. As a proof-of-concept, we modelled and checked using our method on an infusion pump system. The specification of this system was extracted from a requirements document created by the CriSys Group of the University of Minnesota.

This document is composed of four chapters preceded by this introduction and followed by a review and the perspectives of this work. First, in Chapter 2, we give a state of the art of the covered domains such as Software Product Line (2.1), Model Checking (2.2) and Model Checking of Software Product Lines (2.3). Then, in Chapter 3, we present the syntax(3.1) and the execution rules (3.2) of fStateflow followed by the definition of the semantics (3.3). Next, in Chapter 4, we model from scratch the behaviour of an Infusion Pump SPL. This SPL is described in Section (4.1) followed by a presentation of the variability (4.2) and the modelling of its behaviour (4.3). Finally, in Chapter 5 we defined the implementations of fStateflow model (5.1) and FTS model (5.2) then we describe some optimisations (5.3) and their evaluation (5.4).

Chapter 2

State of the Art

2.1 Software Product Line

In response to the increase of demand in software, industry had to optimize its development methods in order to produce software in a more efficient way. A good strategy for increasing productivity and improving quality is a systematic, planned and controlled software reuse.

Software Product Line (SPL) is a domain of software engineering that consists in developing a set of similar software products while maximizing reuse during each development phase [Tri09]. These products, or variants, share a set of common functionalities and satisfy requirements in a particular domain but they also have some differences that distinguish them. These differences, also called variability, can be defined by the *features* of this SPL. A feature is "a distinguishing characteristic of a software item (e.g., performance, portability, or functionality)" [IEE08]. Each software product is identified by a set of features. The main aim of this concept is to avoid creating a whole new system for each requested product by managing variability during the development process. This method presents the advantages of reducing the production costs, decreasing the time-to-market and improving the quality. However, this new approach implies an initial investment more important than the one for a classic approach due to a bigger complexity in developing a set of products in comparison to a single one.

2.1.1 Feature Diagram

Features of an SPL can be identified by analysing the variability of the products that compounds this SPL. However, a list of features is not enough to represent an SPL. For example, some features cannot be present in the same product or a

given feature requires the presence of another feature. A formalism is thus needed to express constraints over features. Several languages exist but in this thesis we use a language called Feature Diagram (FD). An *FD* is a compact representation of all the products of the SPL.

Graphically, an FD is illustrated by a tree representing the hierarchy of the features. Every node corresponds to a feature and every edge corresponds to the relationship parent-child between two features. Figure 2.1 depicts an example of a Feature Diagram of a mobile phone line. In the following, we will illustrate the syntax of FD using this example.

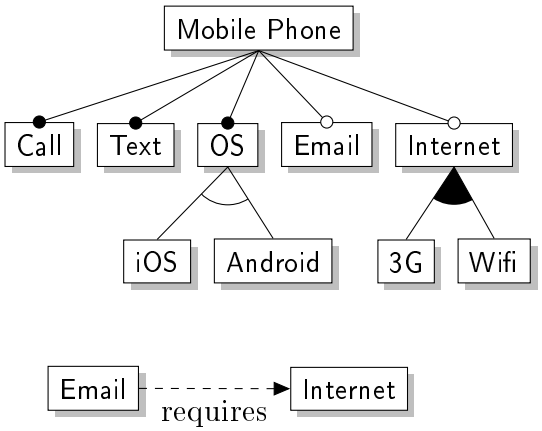


Figure 2.1: Feature Diagram of a line of Mobile Phones

The root of a tree is the only feature that does not have a parent and it is used as the entry point of the diagram. In this example, the root is named as the name of the SPL represented. Fig. 2.2 depicts the notation of a root.

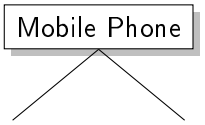


Figure 2.2: Notation of a root

The relationships between a parent feature and its child features are categorized as *And*, *Or* and *Alternative*.
The And-relationship is used between a parent feature and its child features to signify that the child features may be present in a product if and only if the parent feature is present in this product.

Each child feature in an And-relationship is either Mandatory or Optional. Mandatory means that if the parent feature is present in a specific product then the child feature must be present in this specific product. For example, the mandatory feature Call in Fig. 2.3 is present in each mobile phone.

Figure 2.3: Notation of a mandatory feature

If a child feature is optional, this feature is not necessarily present in a product even if its parent feature is present in this specific product. As Internet in Fig. 2.4 is optional, this feature may not be present in a mobile phone.

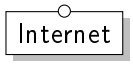


Figure 2.4: Notation of an optional feature

In Fig. 2.5, the Mobile Phone feature and its five child features are linked with an And-relationship where Call, Text, and OS are mandatory, and Internet and Email are optional.

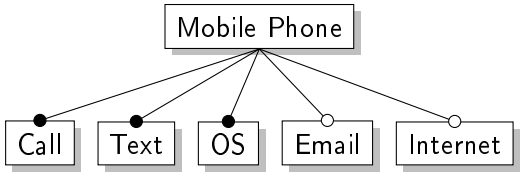


Figure 2.5: Notation of an AND-relationship

A parent feature and its child features can be linked with an Or-relationship, that is, if the parent feature is present in a specific product then at least one of its children features must be present in this specific product. For example, a mobile phone can connect to the Internet using 3G, or Wifi, or both. The notation of this example is depicted in Fig. 2.6.

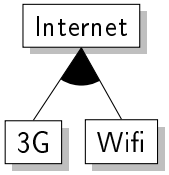


Figure 2.6: Notation of an OR-relationship

In the case of an Alternative-relationship, if a parent feature is present in a specific product then exactly one of its children features must be present in this specific product. Fig. 2.7 shows the notation of an Alternative-relationship. In this example, the OS of a mobile phone is either iOS or Android.

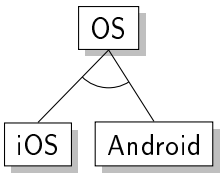


Figure 2.7: Notation of an Alternative-relationship

The parental relationship may not be sufficient to express all the dependencies between the features. Extensions of FDs allow one to express additional cross-tree constraints. The most common constraints are "requires" and "excludes", but any constraint coming down a propositional formula can be defined.

The "require" constraint written A *Requires* B means that if the feature A is present in a specific product then the feature B must be present in this specific product too. For example, mobile phones need an Internet connection to send an Email, hence Email requires Internet. Fig. 2.8 shows the notation of this constraint.

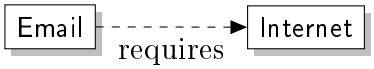


Figure 2.8: Notation of a requires constraint

In the case of the "excludes" constraint, written A *Excludes* B, if the feature A is present in a specific product, then the feature B must not be present in this specific product and vice versa. The notation is depicted in Fig. 2.9.

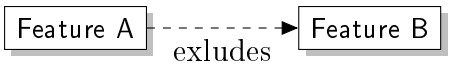


Figure 2.9: Notation of an excludes constraint

After the syntax of FD, we will now define the semantic of FD as follows [SHT06]:

Def. 2.1.1 Feature Diagram :
A Feature Diagram *FD* is a tuple (N, px), where N is the set of features and $px \subseteq 2^N$ is the set of products. ■

From the Feature Diagram depicted in Fig. 2.1, Table 2.1 lists the valid products (14) of the Mobile Phone line.

	Call	**Text**	**OS**	**iOS**	**Android**	**Email**	**Internet**	**3G**	**Wifi**
1	•	•	•	•					
2	•	•	•		•				
3	•	•	•	•			•	•	
4	•	•	•		•		•	•	
5	•	•	•	•		•	•	•	
6	•	•	•		•	•	•	•	
7	•	•	•	•			•		•
8	•	•	•		•		•		•
9	•	•	•	•		•	•		•
10	•	•	•		•	•	•		•
11	•	•	•	•			•	•	•
12	•	•	•		•		•	•	•
13	•	•	•	•		•	•	•	•
14	•	•	•		•	•	•	•	•

Table 2.1: List of valid products of the Mobile Phones SPL depicts on Figure 2.1

In single-system development, one will have to improve confidence in the product using appropriate Quality Assurance (QA) techniques. Two popular techniques are *model checking*, presented in section 2.2, and *testing* which consists in determining whether actual executions of the system behave as expected. However, testing is applicable in late phases of lifecycle, whereas model checking can be integrated earlier to decrease the modification costs. These techniques could be used on SPL but testing or model checking every possible software product would be a tedious work due to a possibly huge number of different combinations of features. In this thesis, we are focusing on adapting Model Checking to Software Product Line.

2.2 Model Checking

At the emergence of Computer Science, ICT systems were only used to help, manage or calculate for simple and inoffensive tasks. Now, due to the evolution and the expansion of these technologies, we have become reliant on ICT systems (also called software). Nowadays, it would be impossible to imagine our society without computer, software or the Internet. Our money, for instance, is handled by worldwide banks equipped with software. Some company are entirely dependant

on their ICT systems and the slightest error can compromise their financial situation. The consequences of software errors are not only limited to financial loss, there may also be human loss. For example, our security is guaranteed by software in airplanes. One bug can be catastrophic implying, in this situation, a loss of money as well as a loss of human lives. We also have to keep in mind that the sooner we catch errors the lower is the cost to repair. There are several verification techniques to prevent these problems. To avoid or find bugs, one can use peer reviewing, testing, emulation, simulation, etc.

In this work, we use model checking to verify systems. Basically, model checking is employed to establish if a system under certain consideration holds certain properties. This technique presents a number of advantages compared to other verification techniques: it is fast, automatic and it shows why a property is not satisfied by giving a counterexample [CLA08]. Moreover, this domain has been rewarded by an A.M. Turing Award in 2007 thanks to the work of E. M. Clarke, E. A. Emerson and J. Sifakis. Briefly, on the one hand we have a model representing the system and on the other hand we have a set of properties formalizing the requirements. The model checker explores all possible executing paths of this model in a brute-force manner and verifies if each path satisfies the properties. If a path does not hold one of the properties, the model checker gives a counterexample representing the violating path. One can then correct the model or adapt the properties and run the model checker again until no counterexample is found.

According to [BK08, ch. 1] in practice the model checking process consists of three phases: the Modelling phase, the Running phase and the Analysis phase. At first, the system under consideration is modelled using the model description language of the model checker at hand and the property to be checked is formalized using the property specification language. Moreover, some simulations of the model are performed as a first sanity check and a quick assessment. Then, the model checker is run to check the validity of the property in the system model. Finally, the result is analysed. If the property is satisfied, the following property is checked. In case the property is falsified, the model, the design or the property has to be refined. Whenever the computer runs out of memory, the model has to be reduced.

Some of the strengths of model checking could be that, in comparison with testing and simulation, it will report an error if and only if there exists at least one in the model. It is also a general verification approach applicable to a wide range of applications. Finally, it provides diagnostic information in case a property is not validated. However, model checking has also some weaknesses. The main problem is that it suffers from state-space explosion: the number of states grows exponen-

tially in the number of program variables. For example, the model of a program with three 32-bits integers needs $(2^{32})^3 = 2^{96}$ states. The number of states can exceed the amount of available computer memory and if this problem happens, one has to reduce the model. Another weakness is that we verify a model of the system, and not the system itself, thus if the model is not correct, or incomplete, the results could be incorrect and useless. Finally, only stated requirements are verified and the validity of properties that are not checked cannot be judged. In spite of its exhaustiveness, model checking cannot guarantee the absence of errors int the actual system.

All the definitions defined later in this section come from chapters 2, 3 and 5 in [BK08].

2.2.1 Transition system

As we do not check the system itself, we need a model of this system for model checking. Transition systems are commonly used to represent hardware and software system behaviours. Basically, they are directed graphs where nodes represent states of the system under consideration and edges represent transitions between these states. Formally, they are defined in [BK08, ch. 2] as follows :

Def. 2.2.1 Transition System (TS) :
A *Transition System* (TS) is a tuple $(S, Act, trans, I, AP, L)$ where

- S is a set of states,
- Act is a set of actions,
- $trans \subseteq S \times Act \times S$ is a transition relation,
- $I \subseteq S$ is a set of initial states,
- AP is a set of atomic propositions, and
- $L : S \to 2^{AP}$ is a labelling function.

In this thesis, we assume that S, Act and AP are finite. Moreover, we also use $s \xrightarrow{\alpha} s'$ to denote $(s, \alpha, s') \in trans$. ■

The intuitive behaviour of a TS can be described as follows. It starts in a state $s_0 \in I$ and it progresses according to the transition relation $trans$. If a state has more than one outgoing transition, the transition to take is selected in a non-deterministic way. Let be s the current state and $s \xrightarrow{\alpha} s'$ the selected transition, α is performed and the TS evolves from state s to state s'. Similarly, if

the set of initial states contains several elements, the start state is selected non-deterministically.
The labelling function L relates a set $L(s) \in 2^{AP}$ of atomic propositions to any states s where 2^{AP} denotes the power set of AP. $L(s)$ is the set of atomic propositions that are satisfied by state s.
According to [BK08, ch. 2], we defined additional concepts that will be used throughout this thesis. We first introduce the notions of predecessor and successor.

Def. 2.2.2 Direct Predecessors and Successors :
Let $TS = (S, Act, trans, I, AP, L)$ be a transition system. For $s \in S$ and $\alpha \in Act$, the set of direct α-successors of s is defined as:

$$Post(s, \alpha) = \{s' \in S | s \xrightarrow{\alpha} s'\}$$

The set of direct successors of s is defined by:

$$Post(s) = \bigcup_{\alpha \in Act} Post(s, \alpha)$$

Similarly, the set of α-predecessors of s is defined by:

$$Pre(s, \alpha) = \{s' \in S | s' \xrightarrow{\alpha} s\}$$

The set of direct predecessors of s is defined by:

$$Pre(s) = \bigcup_{\alpha \in Act} Pre(s, \alpha)$$

■

A terminal state s of a TS is a state without outgoing transition. We can formally define this concept as follows:

Def. 2.2.3 Terminal state :
State s in transition system TS is called terminal if and only if $Post(s) = \varnothing$. ■

As explained above, we try to avoid terminal states that could be synonym of deadlock in the system. In the following, we assume that a TS has no terminal state.

Executions

We will now formalize the notion of *execution* that describes a possible behaviour of a transition system. An execution is defined as follows.

Def. 2.2.4 Execution Fragment :
Let $TS = (S, Act, trans, I, AP, L)$ be a transition system. A finite execution fragment ϱ of TS is an alternating sequence of states and actions ending with a state

$$\varrho = s_0\alpha_1 s_1\alpha_2 \ldots \alpha_n s_n \text{ such that } s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \text{ for all } 0 \leq i \leq n$$

n is the length of ϱ. An infinite execution fragment ρ of TS is an infinite, alternating sequence of states and actions:

$$\rho = s_0\alpha_1 s_1\alpha_2 s_2\alpha_3 \ldots \text{ such that } s_i \xrightarrow{\alpha_{i+1}} s_{i+1} \text{ for all } 0 \leq i$$

■

An execution fragment is called maximal when it cannot be prolonged:

Def. 2.2.5 Maximal and Initial Execution Fragment :
A maximal execution fragment is either a finite execution fragment that ends in a terminal state, or an infinite execution fragment. An execution fragment is called initial if it starts in an initial state, i.e., is $s_0 \in I$ ■

An execution fragment is simply called execution when it starts in an initial state and ends in a terminal state or is infinite.

Def. 2.2.6 Execution :
An execution of transition system TS is an initial, maximal execution fragment.
■

If it exists an execution fragment that starts in an initial state and ends in a state s, this state s is called reachable

Def. 2.2.7 Reachable state :
Let $TS = (S, Act, \longrightarrow, I, AP, L)$ be a transition system. A state $s \in S$ is called reachable in TS if there exists an initial, finite execution fragment

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_n = s.$$

$Reach(TS)$ denotes the set of all reachable states in TS. ■

Paths

In this thesis, only atomic propositions of the states are taken into consideration in order to formulate system properties. Hence, actions of an execution can be omitted to obtain a *path* defined as follows

Def. 2.2.8 Path Fragment :
A $finite$ path fragment $\hat{\pi}$ of TS is a finite state sequence $s_0 s_1 \dots s_n$ such that $s_i \in Post(s_{i-1})$ for all $0 < i \leq n$, where $n \geq 0$. An $infinite$ path fragment π is an infinite state sequence $s_0 s_1 s_2 \dots$ such that $s_i \in Post(s_{i-1})$ for all $i > 0$. ■

Def. 2.2.9 Path :
A *path* of transition system TS is an infinite path fragment that starts in an initial state, i.e., is $s_0 \in I$.. ■

$Paths(s)$ is the set of maximal path fragments π with $first(\pi) = s$, where $first(\pi)$ denotes the initial state of π. And $Paths(TS)$ is the set of paths of TS.

Traces

The definition of path allows us to describe the sequences states that the system can visit during an execution. When verifying system behaviour, what is of interest is not the visited states but the properties they exhibit. Hence, instead of the path $s_0 s_1 s_2 \dots$, we consider the trace $L(s_0)L(s_1)L(s_2)\dots$ that reports the set of atomic propositions that are valid along the execution. Thus, a trace σ of a TS is an *infinite* words over the alphabet 2^{AP}, that is, $\sigma \in (2^{AP})^{\omega}$. $(2^{AP})^{\omega}$ denotes the set of words that arise from the infinite concatenation of words 2^{AP}. Formally, traces are defined as follows.

Def. 2.2.10 Trace and Trace Fragment :
Let $TS = (S, Act, trans, I, AP, L)$ be a transition system without terminal states. The *trace* of the infinite path fragment $\pi = s_0 s_1 \dots$ is defined as $trace(\pi) = L(s_0)L(s_1)\dots$. The trace of the finite path fragment $\hat{\pi} = s_0 s_1 \dots s_n$ is defined as $trace(\hat{\pi}) = L(s_0)L(s_1)\dots L(s_n)$. ■

The set of traces of a set Π of paths is denoted by

$$trace(\Pi) = \{trace(\pi) | \pi \in \Pi\}$$

A trace of state s is the trace of an infinite path fragment π with $first(\pi) = s$. Let $Traces(s)$ denotes the set of traces starting from a given state s :

$$Traces(s) = trace(Paths(s))$$

Finally, the traces of a TS designate the set of traces leaving any initial state:

$$Traces(TS) = \bigcup_{s \in I} Traces(s)$$

2.2.2 Linear Temporal Logic

We also need to specify the requirements which we want to verify on the TS under consideration. In order to formalize these requirements, we use linear-time properties to specify the admissible (or desired) behaviour of the system. More concretely, we will express these properties in Linear Temporal Logic (LTL), a model logic to reason on executions over time. We first define what is a linear-time property and their satisfaction. Then we present the syntax and semantics of LTL. All the following definitions come from [BK08, ch. 3 and 5].

Def. 2.2.11 LT Property :
An LT property over the set of atomic propositions AP is a subset of $(2^{AP})^{\omega}$. ■

An LT property over AP and a set of traces of a TS over AP are both sets of words over 2^{AP} and can be compared to determine if TS satisfies an LT property. Intuitively, a TS satisfies an LT property P if all its observable behaviours are admissible with respect to the property.

Def. 2.2.12 Satisfaction Relation for LT Properties :
Let P be an LT property over AP and $TS = (S, Act, trans, I, AP, L)$ a transition system without terminal states. Then, TS *satisfies* P, noted $TS \models P$, iff $Traces(TS) \subseteq P$. A state $s \in S$ satisfies P, noted $s \models P$, iff $Traces(s) \subseteq P$. ■

Syntax of LTL

It is obviously impractical to specify LT-properties as a set of traces. Instead, one commonly uses a modal logic equipped with temporal operators. In this thesis, we more particularly focus on LTL. Formally, the syntax of LTL is defined as follows:

Def. 2.2.13 Syntax of LTL :
LTL formulae over the set AP of atomic propositions are formed according to the following grammar:

$$\varphi ::= true \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc\varphi \mid \varphi_1 \bigcup \varphi_2$$

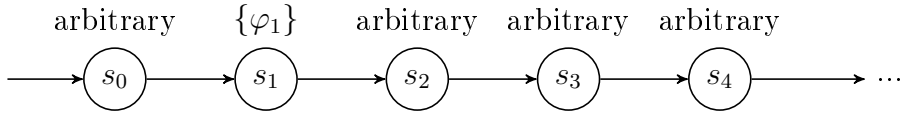
where $a \in AP$ and φ, φ_1 and φ_2 are LTL formulae.

■

LTL formulae are composed of atomic propositions from AP (states labels defined in Definition 2.2.1), the Boolean connectors conjunction $\wedge$ (**"and"**) and negation $\neg$ (**"not"**), and two temporal operators $\bigcirc$ (**"next"**) and $\bigcup$ (**"until"**). Note that LTL formulae express properties about paths and not about states. In the following, given a path, "the current moment" will refer to the current state of this path,

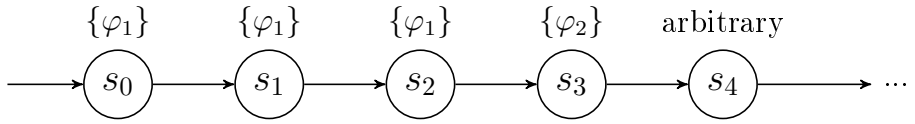
"the next step" will mean the immediate successor state visited in this path and "a moment in the future" will refer to a further state visited in this path. Each operator will be illustrated by an example formula and a path that satisfies this formula.

The $\bigcirc$-operator is unary and requires an LTL formula as argument. Formula $\bigcirc\varphi$ holds at the current moment if φ holds in the next step. An example of path where $\bigcirc\varphi$ is satisfied would be :



In this path, the current moment is state s_0 and the next step is state s_1. As s_1 satisfies φ, the formula $\bigcirc\varphi$ is satisfied. In this case, the properties that hold in the following states do not matter in this case and are *arbitrary*.

The $\bigcup$-operator is binary and requires two LTL formulae as argument. This operator is used to define which property has to hold in a path **"until"** another property holds in the future which means further in the path. $\varphi_1 \bigcup \varphi_2$ holds at the current moment if there is a future moment for which φ_2 and φ_1 hold at all time before this state. $\varphi_1 \bigcup \varphi_2$ holds in the next example:



In this path, s_3 satisfies φ_2 and all states before this one fulfil φ_1. Thus, the formula $\varphi_1 \bigcup \varphi_2$ is satisfied.

We can now introduce some new operators created from those defined above. At first, common propositional logic operators derived from conjunction $\wedge$ (**"and"**) and negation $\neg$ (**"not"**):
$\vee$ (**"or"**), $\Rightarrow$ (**"implies"**), $\Leftrightarrow$ (**"equivalent"**), $\oplus$ (**"xor"**) and $false$ can be derived as follows

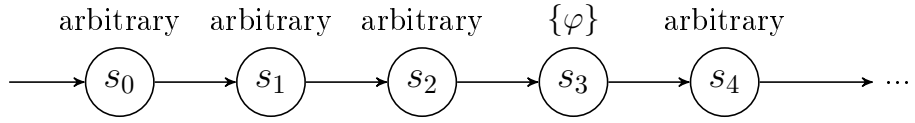
$\varphi \vee \psi$	$\equiv$	$\neg(\neg\varphi \wedge \neg\psi)$	**"or"**
$\varphi \Rightarrow \psi$	$\equiv$	$\neg\varphi \vee \psi$	**"implies"**
$\varphi \Leftrightarrow \psi$	$\equiv$	$(\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$	**"equivalent"**
$\varphi \oplus \psi$	$\equiv$	$(\varphi \wedge \neg\psi) \vee (\neg\varphi \wedge \psi)$	**"xor"**
$false$	$\equiv$	$\neg true$	**"false"**

Then, we can add two temporal operators $\Diamond$ (**"eventually"**) and $\Box$ (**"always"**) derived as follows.

$$\Diamond\varphi \equiv true \bigcup \varphi \quad \textbf{"eventually"}$$
$$\Box\varphi \equiv \neg\Diamond\neg\varphi \quad \textbf{"always"}$$

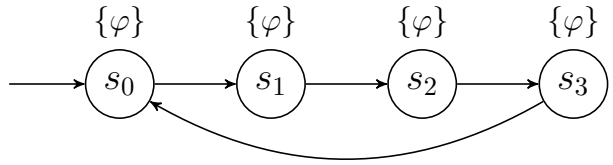
They are both unary operators and require a single LTL formula as argument.

The $\Diamond$-operator describes a property that must **"eventually"** be satisfied in the future, and is derived from $(true \bigcup \varphi)$. For example, $\Diamond\varphi$ means that φ must hold at a moment in the future and moments coming after that one do not matter. An example of path where $\Diamond\varphi$ is satisfied would be:



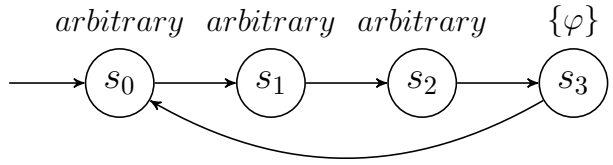
Since s_3 fulfils φ, $\Diamond\varphi$ is satisfied.

The $\Box$-operator is used to define a property that must be satisfied from now on and forever, i.e.,**"always"**. It is derived from $\neg\Diamond\neg\varphi$ which means that $\neg\varphi$ will eventually not hold in the future. For example, $\Box\varphi$ means that φ must hold at the current moment and at all moments in the future. $\Box\varphi$ holds in the following path:



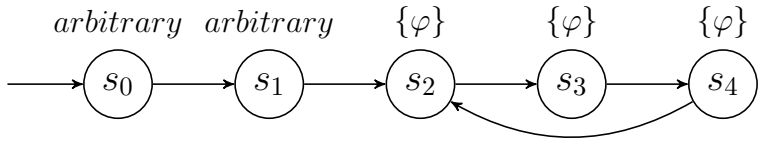
Thanks to the edge from s_3 to s_0 creating a cycle, φ will always hold.

By combining the temporal operators $\Diamond$ and $\Box$, we obtain two new temporal operators : $\Box\Diamond$ (**"infinitely often"**) and $\Diamond\Box$ (**"eventually forever"**). $\Box\Diamond\varphi$ describes the property stating that at any moment there is a moment in the future where φ holds. The following is an example where $\Box\Diamond\varphi$ holds :



Thanks to the edge from s_3 to s_0 creating a cycle, from now on and forever, s_3 where φ is satisfied will always be visited at a moment in the future.

The dual property $\Diamond\Box\varphi$ expresses that from a moment (current or in the future), and then all moments after this one, φ holds. An example of path where $\Diamond\Box\varphi$ is satisfied would be:



φ holds from s_2 and all states after this one thanks to the cycle created by the edge from s_4 to s_2.

Semantics of LTL

We will now define the semantics of LTL formulae with respect to a trace $\sigma = A_0A_1A_2\ldots$. In the following, A_i denotes the set of atomic propositions that are valid in the i^{st} state of the corresponding path of σ, and $\sigma[j\ldots] = A_jA_{j+1}A_{j+2}\ldots$ is the suffix of σ starting from symbol A_j.

The satisfaction relation $\models$ between traces and LTL formulae is defined as follows:

Def. 2.2.14 Semantics of LTL over Traces :
Let φ, φ_1 and φ_2 be LTL formulae over AP and let σ be a trace of a TS over AP. $\models$ is the satisfaction relation between traces and LTL formulae with the following properties:

$$
\begin{array}{lcll}
\sigma & \models & \mathit{true} & \\
\sigma & \models & a & \text{iff} \quad a \in A_0 \text{ (i.e.,} A_0 \models a) \\
\sigma & \models & \varphi_1 \wedge \varphi_2 & \text{iff} \quad (\sigma \models \varphi_1) \wedge (\sigma \models \varphi_2) \\
\sigma & \models & \neg\varphi & \text{iff} \quad \sigma \not\models \varphi \\
\sigma & \models & \bigcirc\varphi & \text{iff} \quad \sigma[1..] = A_1A_2A_3\ldots \models \varphi \\
\sigma & \models & \varphi_1 \bigcup \varphi_2 & \text{iff} \quad \exists j \geqslant 0.\ \sigma[j..] \models \varphi_2 \text{ and } \sigma[i..] \models \varphi_1, \text{ for all } 0 \leqslant i < j
\end{array}
$$

■

From this definition, we can obtain the semantics of derived operators $\Diamond$ and $\Box$ and their combinations $\Box\Diamond$ and $\Diamond\Box$.
For the property $\Diamond\varphi$, as $\Diamond$ is equivalent to $\mathit{true}\bigcup$ we obtain: $\mathit{true}\bigcup\varphi$. Thus, the semantic of $\Diamond$ is

$$\begin{array}{lll} \sigma \models \Diamond\varphi & \equiv & \sigma \models true \bigcup \varphi \\ & \text{iff} & \exists j \geqslant 0.\ \sigma[j..] \models \varphi \text{ and } \sigma[i..] \models true, \text{ for all } 0 \leqslant i < j \\ & \text{iff} & \exists j \geqslant 0.\ \sigma[j..] \models \varphi \end{array}$$

Similarly, $\Box\varphi$ being equivalent to $\neg\Diamond\neg\varphi$, its semantic is defined as follows:

$$\begin{array}{lll} \sigma \models \Box\varphi & \equiv & \sigma \models \neg\Diamond\neg\varphi \\ & \text{iff} & \neg\exists j \geqslant 0.\ \sigma[j..] \models \neg\varphi \\ & \text{iff} & \neg\exists j \geqslant 0.\ \sigma[j..] \not\models \varphi \\ & \text{iff} & \forall j \geqslant 0.\ \sigma[j..] \models \varphi \end{array}$$

Now, the semantics of the combinations of $\Box\Diamond$ and $\Diamond\Box$ can be derived:

$$\begin{array}{lll} \sigma \models \Box\Diamond\varphi & \text{iff} & \forall i \geqslant 0.\ \sigma[i..] \models \Diamond\varphi \\ & \text{iff} & \forall i \geqslant 0.\ \exists j \geqslant i.\ \sigma[j..] \models \varphi \end{array}$$

and

$$\begin{array}{lll} \sigma \models \Diamond\Box\varphi & \text{iff} & \exists i \geqslant 0.\ \sigma[i..] \models \Box\varphi \\ & \text{iff} & \exists i \geqslant 0.\ \forall j \geqslant i.\ \sigma[j..] \models \varphi \end{array}$$

2.2.3 LTL Model Checking

On one hand, we have a transition system TS that represents the behaviours of the real system under consideration. On the other hand, we have a requirement formalized as an LTL formula φ. We have also defined the satisfaction relation between traces of TS and φ. However, we would like to check the validity of φ against TS. In the following, we determine the semantics of LTL over paths, states and TS.

Def. 2.2.15 Semantics of LTL over Paths and States :
Let $TS = (S, Act, trans, I, AP, L)$ be a transition system without terminal states, and let φ be an LTL formula over AP.

For path π of TS, the satisfaction relation is defined by

$$\pi \models \varphi \text{ iff } trace(\pi) \models \varphi$$

For state s of TS, the satisfaction relation is defined by

$$s \models \varphi \text{ iff } \forall \pi \in Paths(s).\ \pi \models \varphi$$

TS satisfies φ, denoted $TS \models \varphi$, if $\forall \sigma \in Traces(TS).\ \sigma \models \varphi$ ■

From this definition,

$$
\begin{array}{lll}
TS \models \varphi & \text{iff} & \sigma \models \varphi \text{ for all } \sigma \in Traces(TS) \\
& \text{iff} & \pi \models \varphi \text{ for all } \pi \in Paths(TS) \\
& \text{iff} & s_0 \models \varphi \text{ for all } s_0 \in I
\end{array}
$$

Thus, TS satisfies φ if, and only if, all initial states of TS satisfy φ.

In practice, an LTL Model Checking algorithm is a decision procedure which returns "yes" if $TS \models \varphi$ or "no" with a counterexample if $TS \not\models \varphi$. The counterexample consists in a prefix of an infinite path in TS where φ does not hold. The basic idea of the algorithm is to try to disprove $TS \models \varphi$ by looking for a path π in TS with $\pi \models \neg\varphi$. If such a path is found, a prefix of π is returned as error trace. If not, $TS \models \varphi$.

In the following, we will show the satisfaction of LTL formulae considering the TS depicted in Figure 2.10 with the set of propositions $AP = \{a, b\}$.

$TS \models \Box b$?
TS satisfies formula $\Box b$ if and only if $\pi \models \Box b$ for all $\pi \in Paths(TS)$. In the path $\pi = s_0 s_1 s_2 s_3 s_2 s_3 ...$, the state s_1 is not labelled with b. Hence, $\exists \pi \in Paths(TS)$ where $\pi \not\models \Box b$, and $TS \not\models \Box b$.

$TS \models \Diamond\Box b$?
All paths in $Paths(TS)$ begin with $s_0 s_1 s_2 ...$ or $s_0 s_2 ...$, thus s_2 is always reached. Moreover, in all paths, s_2 and all states after this one are labelled with b. Hence, $TS \models \Diamond\Box b$.

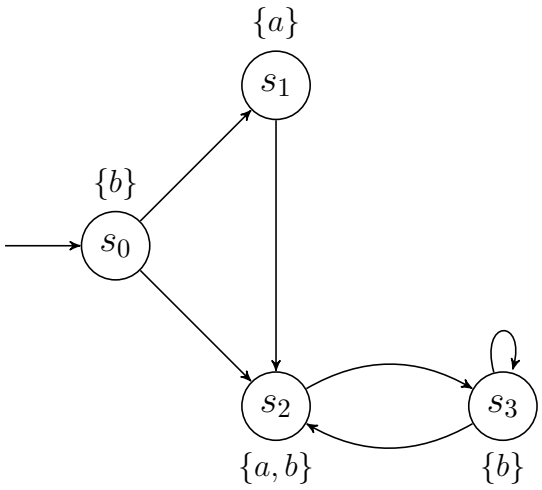


Figure 2.10: Example of TS for semantics of LTL

2.3 Model Checking of Software Product Lines

As a quality assessment, it would be interesting to use model checking techniques on SPL. However, as explained above, model checking suffers from the state-space explosion problem. When applied to SPL, this problem is more important. Indeed, with a set of N features, an SPL may potentially be compound of 2^N different products in the worst case. Given that even one model can result in a state-space explosion, a large set of models would worsen this problem. Since product variants share commonalities, it is also suboptimal to check each product separately. Dedicated model checking methods aware of variability are therefore needed.

2.3.1 Featured Transition System

The Feature Transition System (FTS) formalism has been proposed to adapt model checking to SPL. FTS is designed to describe the combined behaviour of a whole SPL. FTS are transition systems in which transitions are labelled with constraints over the features of an SPL (in addition to being labelled with actions) to restrict the set of products able to execute a given transition. An FTS is defined in [CCS+12] as follows:

Def. 2.3.1 Featured Transition System :
An FTS is a tuple $(S, Act, trans, I, AP, L, d, \gamma)$, where

- $S, Act, trans, I, AP, L$ are defined as in Definition 2.2.1
- d is an FD, as defined in Definition 2.1.1, used to restrict verification to valid products.
- $\gamma : trans \longrightarrow \mathbb{B}(F)$ is a total function, labelling each transition with a feature expression, i.e., a Boolean expression over the features.

■

Figure 2.11 shows an example of an FTS of the aforementioned mobile phone SPL with the set of features $F = \{Email, Internet, 3G, Wifi\}$ and the set of actions $Act = \{text, send, dial, call, close, connect, email\}$. For instance, transitions from state a to state e are guarded by feature expressions $(Internet \wedge Wifi)$ and $(Internet \wedge 3G)$, specifying that a product must possess features $Internet$ and, $Wifi$ or $3G$ to move from a to e. Similarly, the transition from state e to state f is guarded by the feature expression $(Email)$. Hence, a product must have feature $Email$ to move from e to f.

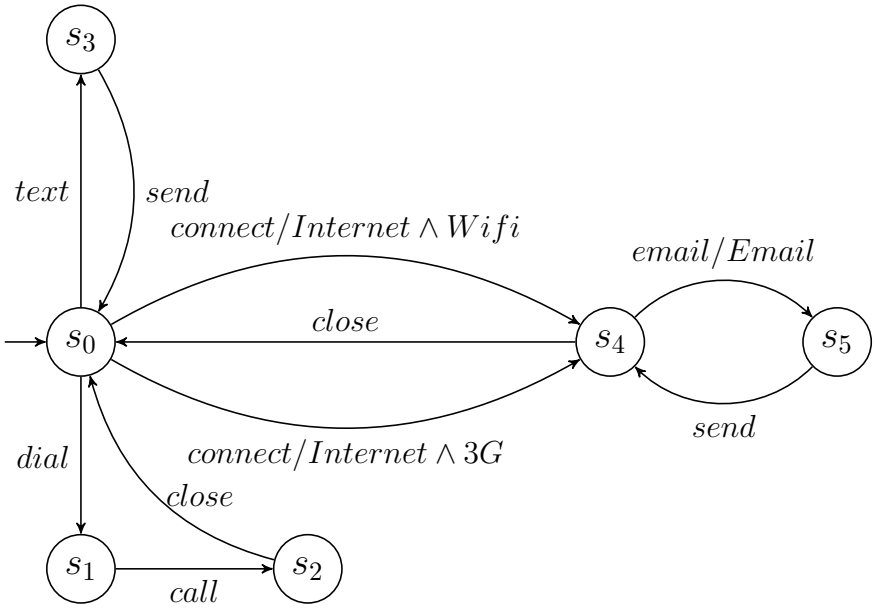


Figure 2.11: Example of FTS modelling the mobile phone FTS

The TS of a particular product p is obtained by removing all transitions whose feature expression is not satisfied by the features of p. This operation is called *projection*. Figure 2.12 shows the projection of the FTS depicted in Figure 2.11 to the product numbered 1 in Table 2.1 with features $\{Call, Text, Os, iOS\}$.

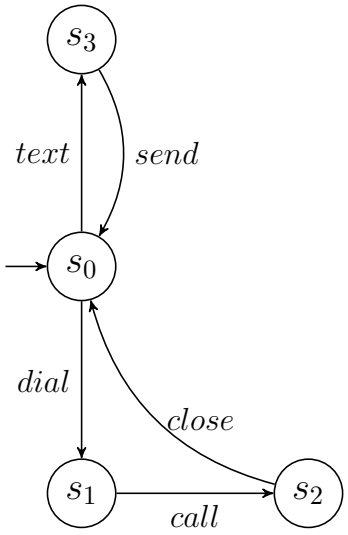


Figure 2.12: Projection of the FTS depicts in Figure 2.11 to the product numbered 1 in Table 2.1

2.3.2 FTS Model Checking

We explained in section 2.2.3 that given a transition system TS and a LTL Property φ, the model checker returns "yes" if $TS \models \varphi$ or "no" with a path counterexample. In FTS model checking, given that we check a set of products, and not a single product, the result is a set of counterexamples each of which is associated to a subset of products that can execute it. For a product to execute a transition, its features must satisfy the feature expression labelling the transition. To produce a whole execution, which is $\varrho = s_0\alpha_1 s_1\alpha_2 \ldots \alpha_n s_n$, the product must be able to execute all the transitions of this execution. Hence, the set of products that can produce the execution is encoded by $\gamma(s_0 \xrightarrow{\alpha_1} s_1) \cap \gamma(s_1 \xrightarrow{\alpha_2} s_2) \cap \ldots \cap \gamma(s_{n-1} \xrightarrow{\alpha_n} s_n)$ where γ is defined in 2.3.1.

In the past three years, several FTS model checkers have been implemented. A first model checker is a Haskell library implementing algorithms for LTL model checking of FTS [CHS+10]. Then, an FTS extension of the NuSMV model checker has been developed [Cla10]. Unlike the Haskell library, NuSMV extension is based on another logic called CTL [BK08, Ch. 6]. Next, SNIP has been proposed in 2012. It has a specification language based on Promela and it offers the possibility to integrate an FD in order to restrict the verification to valid products [CCH+12]. Finally, an SPL of SPL model checker called ProVeLines has been presented in 2013. Variability within ProVeLines comes from four factors such as the type of system to verify, the type of properties to check, the complexity of features and the used data structures [CCS+13].

For example, SNIP is a variant of ProVeLines with support for LTL verification of purely boolean, discrete software product lines, and equipped with binary decision diagrams. Some extensions are under development to introduce variability in input languages but currently fPromela is the only available input language.

Because of its fundamental formalism, FTS is really tedious to be modelled manually. Existing tools propose languages above FTS in order to ease modelling. These languages include advance constructions such as variables, data structures and process. Despite the better usability of these languages of behavioural modelling, SPL model checking techniques will not reach the industry if an expertise is required to use these new languages. It is therefore important to propose a more widespread language in the industry as input language of SPL model checker.

In this thesis, we create a variability-aware extension of Stateflow, a tool integrated in the Matlab environment and used for the development and the simulation of complex reactive systems [SP00]. It uses a variant of the finite state machine

notation established by Harel [Har87] enabling the representation of hierarchy, parallelism and history within a state chart. In the next chapter, we define the syntax and the execution rules of fStateflow. Then, we define its semantics in terms of FTS in order to use SPL model checking techniques on system modelled in this language.

Chapter 3

Modelling SPL behaviour with fStateflow

As FDs do not allow behaviour expressing, a feature-aware language is needed to model the behaviour of an SPL. In this thesis, we present the Stateflow language based on the finite state machine notation established by Harel [Har87]. A Stateflow model is composed of a set of states and a set of transitions. It also provides advanced constructs such as hierarchy and parallelism. At first, we present the different notations used in the Stateflow tool. Illustrating all the constructions would be too tedious, thus, we only present a subset of these notations. Then, we propose an extension of this language, called fStateflow, which permits to express variabilities.

3.1 Syntax

In the following, we present all the notations and concepts used to model the behaviour of an SPL in Stateflow [Mat04].

3.1.1 States label

A Stateflow model includes a set of states. Each state has a label which starts with the name of the state followed by an optional set of actions. A name is valid if (1) it is only composed of alphanumeric characters including the underscore (_) and (2) it does not begin with a numeric character.

After the name, one can add state actions that are statements over variables written in C or Matlab syntax. Here, we focus on C syntax. State actions are separated by semicolons and preceded by their type. There are three types of actions which are *entry*, *during* and *exit*. An empty type corresponds to entry actions. Other

types of action exist but we only use these three types.

Figure 3.1 depicts an example of a state labelled with the name "On" and a set of actions where "$i = 0$;" and "$j = 0$;" are entry actions, "$i++$;" is a during action and "$j = i * 2$;" is an exit action.

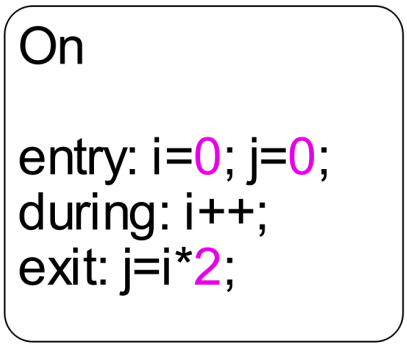


Figure 3.1: State label example

Formally, let V be a set of variables and $Stmnt(V)$ the set of statements over V. A Stateflow model includes the following:

- S is a set of states,
- $entry : S \longrightarrow Stmnt(V)$ is a function between a state and its set of entry statements;
- $during : S \longrightarrow Stmnt(V)$ is a function between a state and its set of during statements;
- $exit : S \longrightarrow Stmnt(V)$ is a function between a state and its set of exit statements.

3.1.2 Hierarchy

In Stateflow, states are organized as a hierarchy. This allows one to represent multiple levels of subcomponents in a system. Graphically, drawing a state A within the boundaries of another state B indicates that A is a substate of B and B is

the superstate of A. Each state, excepted from the top-level state, has exactly one superstate and it may have several substates. Different states can have the same name. To distinguish between those, we use the so-called absolute name, which consists in the concatenation of parents names and the state name separated by period

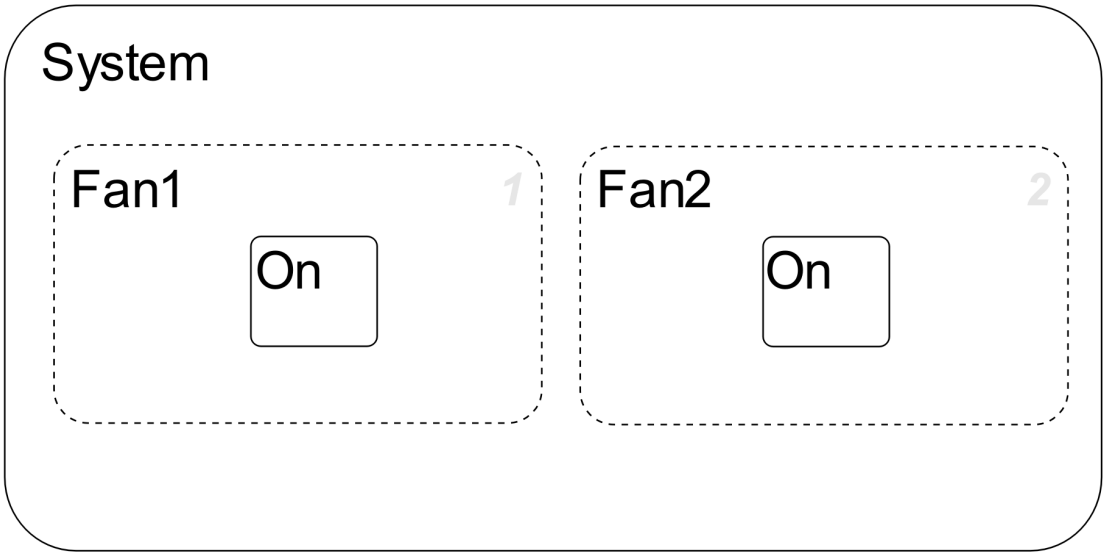


Figure 3.2: States hierarchy example

For example, the list of full names of the states depicted in Figure 3.2 are "$System$", "$System.Fan1$", "$System.Fan2$", "$System.Fan1.On$" and "$System.Fan2.On$". Even if there are two states labelled with "On", as they have different full names, there is no syntax error.

3.1.3 Decomposition

Every state has a decomposition that defines the type of all its substates. There are two types of decomposition: exclusive or parallel substates.

Exclusive substates are indicated by solid borders. Additionally, if a given state has several exclusive substates, at least one of these must be targeted by a transition without source, called *Initial Transition*. It means that this substate is the first to be activated if the superstate becomes active (see Subsection 3.2). As illustrated in Figure 3.3, substates B and C are exclusive and B is targeted by an initial transition.

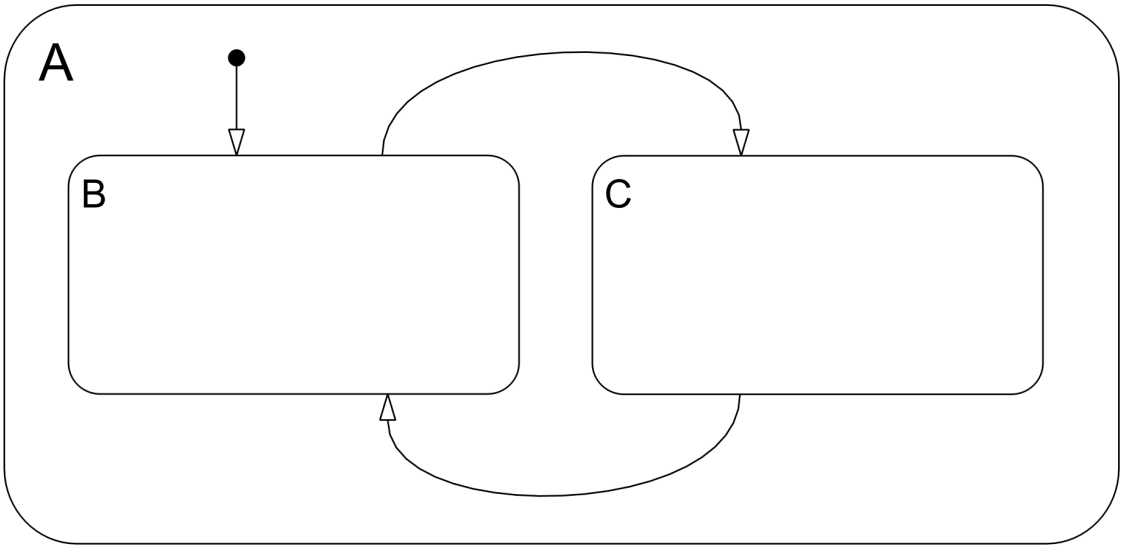


Figure 3.3: Exclusive states example

In the case where substates are parallel, they are represented by dashed borders. Unlike exclusive states, a parallel state must not be the source or the target of transitions. In addition, a total order is defined over parallel states. This order influences the executions of the Stateflow model (see Subsection 3.2). Each substate has a number written at the top right corner representing its order. Figure 3.4 depicts an example where substates B and C are parallel. B is numbered 1 and C is numbered 2.

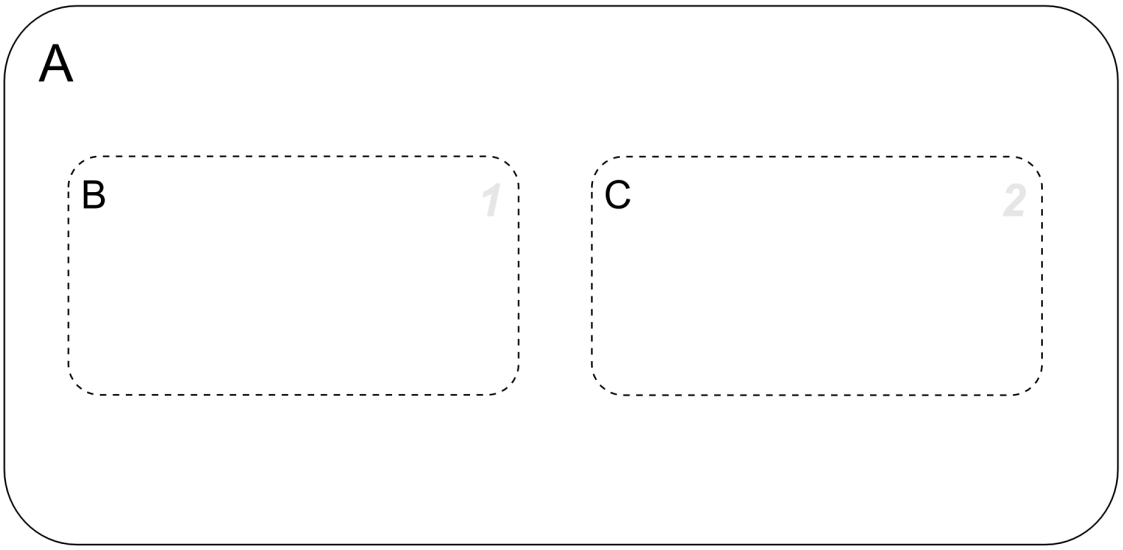


Figure 3.4: Parallel states example

Formally, hierarchy function *sub* is a partial function that associates a state with (1) the set of its substates, (2) a decomposition type, (3) for exclusive decomposition the child/children that is/are initial state(s) within that decomposition, and (4) for parallel states, the order of each substate as follows:

$$sub : S \longrightarrow 2^S \times \begin{pmatrix} [\{Excl\} & \times & (S \longrightarrow (Pred(V) \times \mathbb{N}))] \\ & \cup & \\ [\{Paral\} & \times & (S \longrightarrow \mathbb{N})] \end{pmatrix}$$

3.1.4 Transitions label

Transition may be labelled with a guard and a set of actions. Labels must respect the following syntax: $[guard]^?\{actions\}^?$. The guard is a predicate that defines if a transition is valid or not (see Subsection 3.2). The actions are statements written in C and performed when the transition is valid. Note that both elements are optional. Figure 3.5 depicts an example of transition label. In this case, "$i < 0$" is the guard and "$i++;$" is an action.

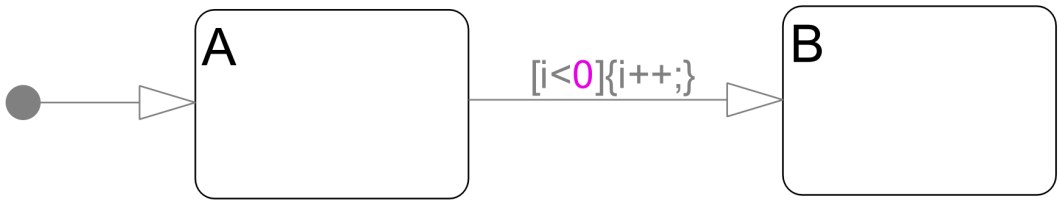


Figure 3.5: Example of a transition label

3.1.5 Order

In the case there are more than one exiting transition of a given state, they must be ordered by writing an order number right at the beginning of the transition. Similarly, an order must be defined between several initial transitions of a given state. During execution, transitions are evaluated following this order to determine their validity (see Subsection 3.2).

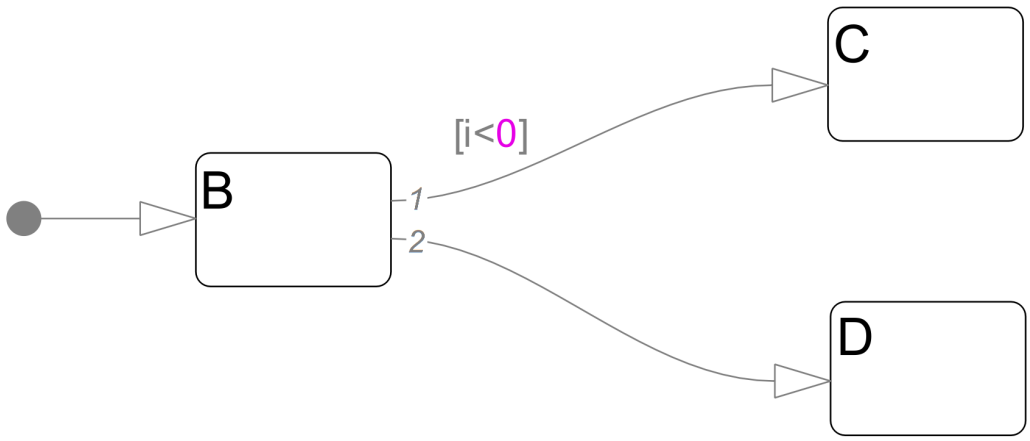


Figure 3.6: Example of multiple exiting transitions

In Figure 3.6, the transition labelled with the condition $i < 0$ from state B to C is the first in the order. The transition from B to D is the second one.

Formally, transition function *trans* is the transition relation which notably describes the guard, the actions and the order of each transition as follows:
$trans \subseteq (S \times \mathbb{N}) \times Pred(V) \times Stmnt(V) \times S$
where $Pred(V)$ is the set of predicates over V.

3.1.6 Junction

Junctions, represented by empty circles, are syntactic sugars that allow for dividing transitions into transition segments. Junctions permit to represent

- "*if-then-else*" decision construct
- transition from a common source to multiple destination (fork)
- transition from a multiple source to common destination (join)
- ...

Formally, the transition segments relation is defined as
$trans_{segments} : ((S \cup J) \times \mathbb{N}) \times Pred(V) \times Stmnt(V) \times (S \cup J)$ where J is a set of junction. From this relation we can obtain the transition relation *trans* as follows.

Let be j a junction, $segment_1 = ((s_1, n_1), pred_1, act_1, j)$ a transition segment that targets j and $segment_2 = ((j, n_2), pred_2, act_2, s_2)$ a transition segment that exits j. From $segment_1$ and $segment_2$ we obtain the transition $trans_{jn_1} = ((s_1, n_2), pred_1 \cap pred_2, act_1 + act_2, s_2)$. Hence, if there are i transition segments that target j and k transition segments that exit j, we have $i \times k$ transitions.

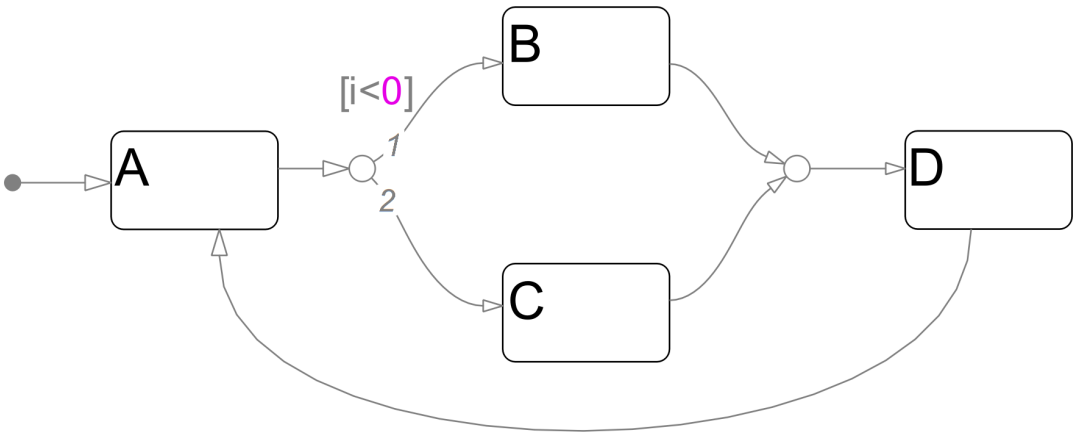


Figure 3.7: Example of transitions with junctions

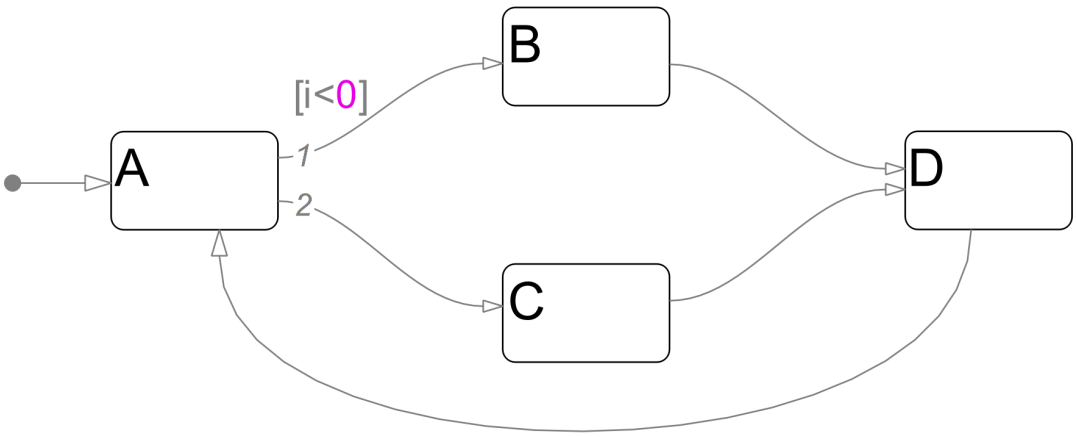


Figure 3.8: Same model as depicted in Figure 3.7 where junctions have been removed

Following the aforementioned process, Figure 3.8 depicts a model obtained by removing junctions from the model illustrated in Figure 3.7. Both model have the same behaviour.

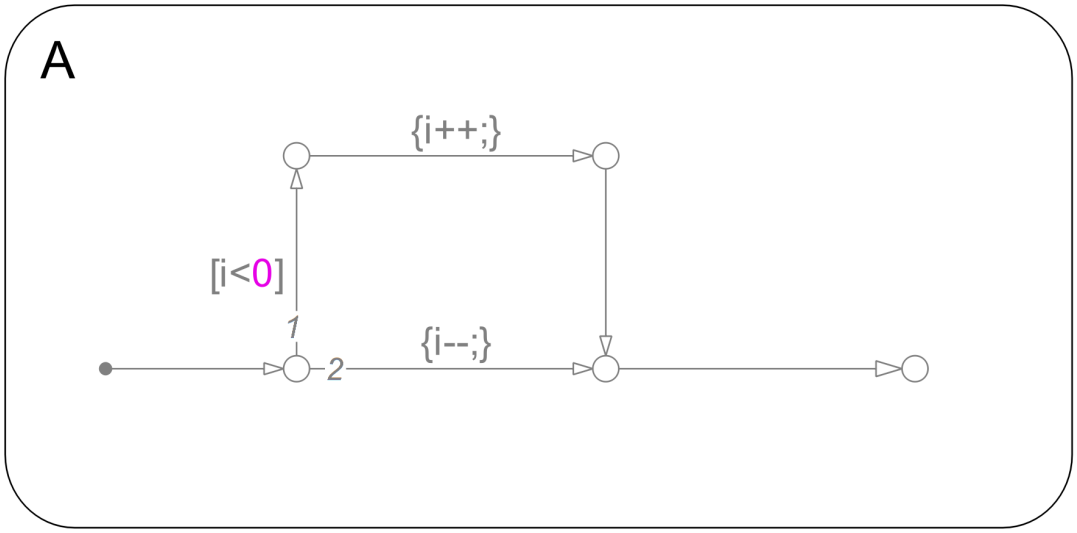


Figure 3.9: Example of an *if-then-else* construct

Figure 3.9 depicts an example of an *if-then-else* construct. In this example, the construct represents the following instructions:

```
if (i<0)
        i++;
else
        i--;
```

3.1.7 Variables

As mentioned above, state actions and transition actions are statements over a set of variables. Variables must have a type and a scope declared in the model explorer of the Matlab environment. In this thesis, we only use boolean and integer of 32 bits as type of variables. Regarding scope, variable can be either constant local, constant input or undefined input[1]. Local are variables initialized to 0 that can be modified by actions. Constant input variables are initialized apart from the model

[1]There is also an output scope but output variables behave the same as local variables. In the Matlab environment, several models can sent or receive variables. Where inputs variables are received by the model, output variables are sent by the model.

and, by definition, cannot not be modified. Undefined inputs are variables which values over time are determined before the execution of the model. For model checking purpose, these variables are considered as undefined in order to check the model in a non-deterministic way.

3.1.8 Features

These notations are the basics for modelling in Stateflow but there is no concept to express variabilities. Hence, we need to extend this language to make Stateflow aware of features. Similarly to FTS, we label transition with feature expressions. In order to differentiate boolean condition and feature expression, both are surrounded by parenthesis and separated by the AND connector &&. Formally, labels must follow this syntax:

$$
\begin{array}{lcl}
label & ::= & (\text{"}[\text{"}guard\text{"}]\text{"})^{?}(\text{"}\{\text{"}actions\text{"}\}\text{"})^{?} \\
guard & ::= & \text{"}(\text{"}condition\text{"})\text{"} \;\&\&\; \text{"}(\text{"}feature\_expression\text{"})\text{"} \\
 & | & \text{"}(\text{"}condition\text{"})\text{"} \\
 & | & \text{"}(\text{"}feature\_expression\text{"})\text{"}
\end{array}
$$

Graphically, *condition* and $feature\_expression$ are aggregated. In order to distinguish them during the parsing, we associate a feature diagram to this model.

Formally, let F be a set of features, $\gamma : trans \longrightarrow \mathbb{B}(F)$ is a function that associates each transition with a feature expression over F and d is a feature diagram over F.

Note that in the Matlab Environment, feature expression are declared as undefined inputs.

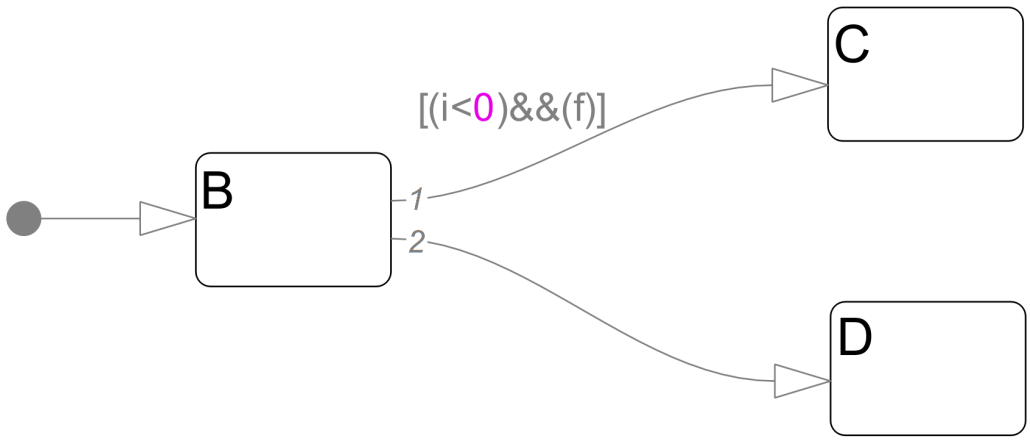


Figure 3.10: Example of transition with a feature expression

Figure 3.10 depicts an example of a transition labelled with feature expression f.

3.1.9 fStateflow model definition

Formally, an fStateflow model is defined as follows

Def. 3.1.1 fStateflow model :
An *fStateflow model* is a tuple $(S, entry, during, exit, sub, trans, d, \gamma)$ where

- S is a set of states;
- $entry$, $during$ and $exit$ are partial functions from S to $Stmnt(V)$ that associate a state with the statement to execute when the system enters, remains and exit this state respectively;
- $sub : S \longrightarrow 2^S \times \begin{pmatrix} [\{Excl\} & \times & (S \longrightarrow (Pred(V) \times \mathbb{N}))] \\ & \cup & \\ [\{Paral\} & \times & (S \longrightarrow \mathbb{N})] \end{pmatrix}$
  is the hierarchy function, i.e., a partial function that associates a state with (1) the set of its substates, (2) a decomposition type, (3) for exclusive decomposition the child/children that is/are initial state(s) within that decomposition, and (4) for parallel states, the order of each substate;
- $trans \subseteq (S \times \mathbb{N}) \times Pred(V) \times Stmnt(V) \times S$ is the transition relation which notably describes the guard, the actions and the order of each transitions;
- d is a feature diagram over features F;
- $\gamma : trans \longrightarrow \mathbb{B}(F)$ associates each transition with a feature expression over F;

where F is a set of features, V is a set of variables, $Stmnt(V)$ is the set of statements over V and $Pred(V)$ is the set of predicates over V. ■

In the following, we also use functions derived from Def. 3.1.1. These are defined as follows:

- $subonly : S \longrightarrow 2^S$ is the partial function that associates a state with its direct substates;
- $super : S \longrightarrow S$ is the partial function that associates a state with its superstate;
- $decomp : S \longrightarrow \{Excl, Paral\}$ is the partial function that associates a state with its decomposition type;
- $sub : (S \times \mathbb{N}) \longrightarrow S$ is the partial function that associates a state with its i^{th} parallel substate;

- $initsub : S \longrightarrow S$ is the partial function that associates a state with its initial substate, i.e., the substate targeted by the valid initial transition;
- $source : trans \longrightarrow S$ is the partial function that associates a transition with its source state;
- $target : trans \longrightarrow S$ is the partial function that associates a transition with its target state;
- $pred : trans \longrightarrow Pred(V)$ is the partial function that associates a transition with its predicate over variables V;
- $act : trans \longrightarrow Stmnt(V)$ is the partial function that associates a transition with its statements over variables V;

3.2 Execution rules

The previous section presents the syntax of fStateflow. Now, we need to present how a model expressed in fStateflow is executed step by step, i.e., how the model evolves from state(s) to state(s). This allows us to define the semantics of fStateflow further in this section.

3.2.1 Activity

Stateflow (and in extenso fStateflow) models evolve at discrete time steps. At a given point of time, state can be active, which means that this state is visited at the current step of execution. Activity is influenced by the state decomposition. In the case of exclusive states, if the superstate becomes active, an initial substate is activated. Afterwards, only one exclusive substate of a given state can be active at a time. In the case of parallel states, if the superstate becomes active then all of these child states are activated one after the other following the order determined previously. We use the function $in(State)$ to denote the activity of $State$

Activity of states also trigger states actions depending on their type:

- An entry action is executed when the state becomes active.
- An exit action is executed when the state becomes inactive.
- A during action is executed when the state is active and there is no valid transitions exiting this state.

Let *execute*(*act*) be the procedure that executes the statements of *act*. The deactivation procedure of the state s is given in Algorithm 1.

Algorithm 1 *deactivate*(s)

```
1:  if in(s) == 1 then
2:     sup := super(s);
3:     if decomp(sup) == Paral then
4:        for i = |subonly(sup)| to 1 do
5:           s_sub = sub(sup, i);
6:           deactivate(s_sub);
7:        end for
8:     end if
9:     for all s_sub ∈ sub(s) do
10:       deactivate(s_sub);
11:    end for
12:    in(s) := 0;
13:    execute(exit(s));
14:    deactive(sup);
15: end if
```

Algorithm 1 can be explained as follows:

1. If the state is already inactive, do nothing (line 1).
2. Sibling parallel states are deactivated starting with the last-entered and progress in reverse order to the first-entered (lines 2-8).
3. If a state has active children, these are deactivated (lines 9-11).
4. Mark the state as inactive (lines 12).
5. Execute exit actions (lines 13).
6. Deactivate superstate (lines 14).

The activation procedure of the state s is given in Algorithm 2.

Algorithm 2 *activate*(s)

```
1: if in(s) == 0 then
2:    sup := super(s);
3:    if in(sup) == 0 then
4:       activate(sup);
5:    end if
6:    in(s) := 1
7:    execute(entry(s))
8:    if decomp(s) == Excl then
9:       activateSubExcl(s);
10:   else if decomp(s) == Paral then
11:      activateSubParal(s);
12:   end if
13: end if
```

Algorithm 3 *activateSubExcl*(s)

```
1: for all s_sub ∈ subonly(s) do
2:    if in(s_sub) == 1 then
3:       active :=true;
4:       break
5:    end if
6: end for
7: if not active then
8:    s_sub = initsub(s);
9:    activate(s_sub);
10: end if
```

Algorithm 4 *activateSubParal*(s)

```
1: for i = 1 to |subonly(s)| do
2:    s_sub = sub(s, i);
3:    activate(s_sub);
4: end for
```

The intuitive explanation of the Algorithm 2 is the following:

1. If the state is already active, do nothing (line 1).
2. If the superstate is not active, perform all steps for the superstate first (lines 2-5).

3. Mark the state active (line 6).
4. Execute entry actions (line 7).
5. Activate substates, if needed:
   - If this state has exclusive substates (lines 8-9):
     (a) Check if there is an active substate (Algorithm 3, lines 1-5).
     (b) If not, perform all steps for the valid initial state (Algorithm 3, lines 6-9).
   - If this state has parallel substates (lines 10-11):
     (a) perform all steps for all substates following their order (Algorithm 4, lines 1-4).

3.2.2 Hierarchy diagram

These activation rules can be represented using the FD syntax as defined in 2.1.1. The relation between a state and its substates is represented by respectively a parent feature and its child features. The type of the relationship is *Alternative* if substates are exclusive and *And* if substates are parallel (in this case, all child features are mandatory). The produced diagram, called a *hierarchy diagram* (HD), represents all the possible "configurations" which means the set of set of states that can be active at the same time. Figure 3.11 depicts a more complex example of a fStateflow model and Figure 3.12 illustrates the *HD* of the model.

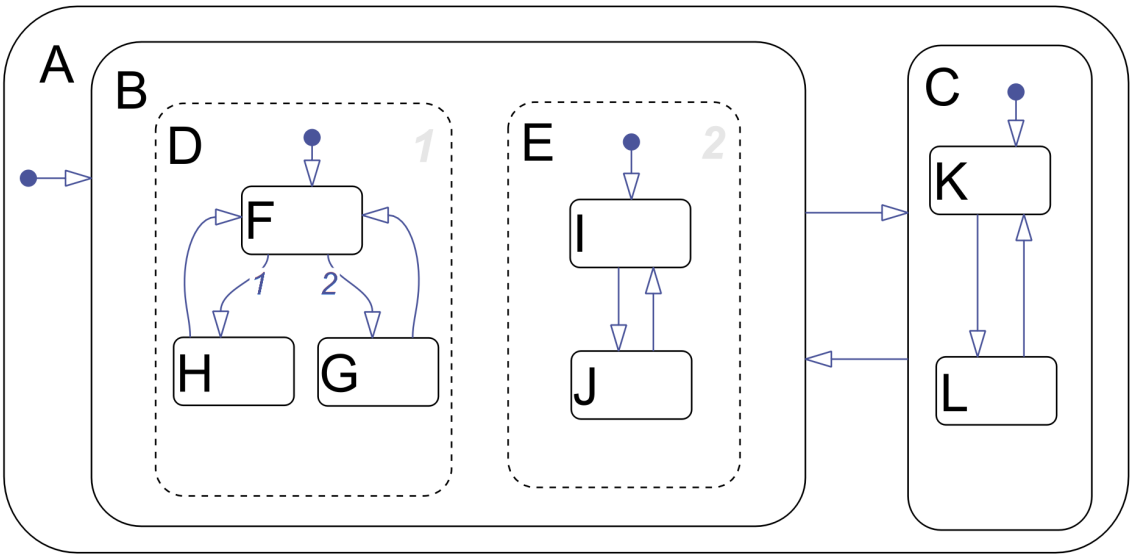


Figure 3.11: Example of a fStateflow model

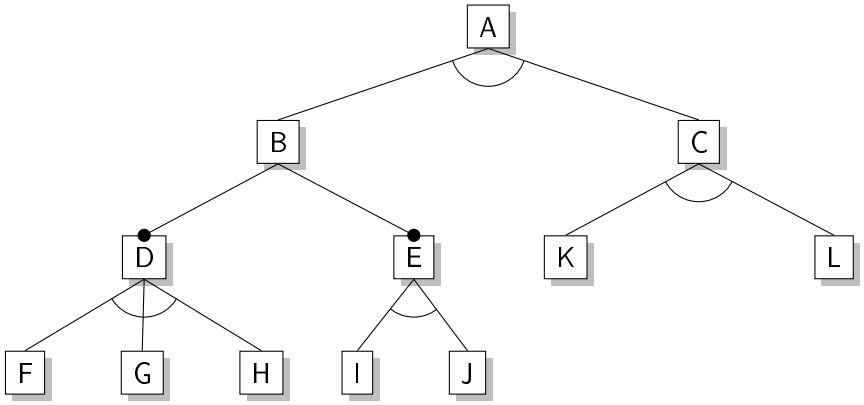


Figure 3.12: Hierachy diagram of the model depicted in Figure 3.11

From this diagram, we can make the set of set of states that can be active at the same time as illustrated in Table 3.1.

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**
1	•	•		•	•	•			•			
2	•	•		•	•	•				•		
3	•	•		•	•		•		•			
4	•	•		•	•		•			•		
5	•	•		•	•			•	•			
6	•	•		•	•			•		•		
7	•		•								•	
8	•		•									•

Table 3.1: Set of set of states that can be active at the same time

3.2.3 Validity

From a given active state, any transition cannot be taken at any time: only a valid transition can be performed. The validity of a given transition is defined by its guard. Condition can be evaluated to true, false or undefined (in the case of evaluation of undefined variables). If its condition is not evaluated to false, then the transition is valid. In the case where there are more than one transition exiting a given state, transitions are evaluated following their order. Hence, the i^{th} transition is valid iff (1) its condition is not evaluated to false and (2) the $(i-1)$ first transitions are not valid. For example, the transition from B to D in Figure 3.6 is performed only if the condition of the first evaluated transition is false, i.e., iff $i \geq 0$.

From definitions of activity and validity, we can use $s \xrightarrow{[g_i]a_i} s'$ to denote $((s,i), pred_i, act_i, s') \in trans$, that is, the i^{th} transition from s to s' where

- $pred_j$ and act_j are respectively the predicate and the statements over V of the j^{th} transition from s to s';
- $actStmnt(s)$ is the set of statements executed when state s is activated as defined in Alg. 2;
- $deactStmnt(s)$ is the set of statements executed when state s is deactivated as defined in Alg. 1;
- $a_i = deactStmnt(s) \cup act_i \cup actStmnt(s')$;
- $g_i = \neg(\bigwedge_{j=1}^{i-1} pred_j) \bigwedge pred_i$;

3.2.4 Execution

The execution of a fStateflow model begins by activate the top-level state. The next steps consist in evaluate if a state has a valid transition. First the top-level state is evaluated. If no transition is valid, its during actions are executed and substates are evaluated afterwards. In the case of an exclusive decomposition, only the active substate is evaluated. In the case of a parallel decomposition, each substate is evaluated one after the other following the order. Once a valid transition is found, this transition is applied and the substates are not evaluated. The application of a transition consists in (1) deactivate the source state, (2) execute the transition actions and (3) activate the target state.

Let $valid(s)$ be a function that associates a state with its valid transition. The execution procedure applied on the top-level state s is given in Algorithm 5.

Algorithm 5 $execution(s)$

```
1: if |valid(s)| == 0 then
2:    execute(during(s));
3:    if decomp(s) == Excl then
4:       execution(initsub(s));
5:    else if decomp(s) == Paral then
6:       for all s_sub ∈ subonly(s) do
7:          execution(s_sub);
8:       end for
9:    end if
10: else
11:    trans = valid(s);
12:    deactivate(source(trans));
13:    execute(act(trans));
14:    activate(target(trans));
15: end if
```

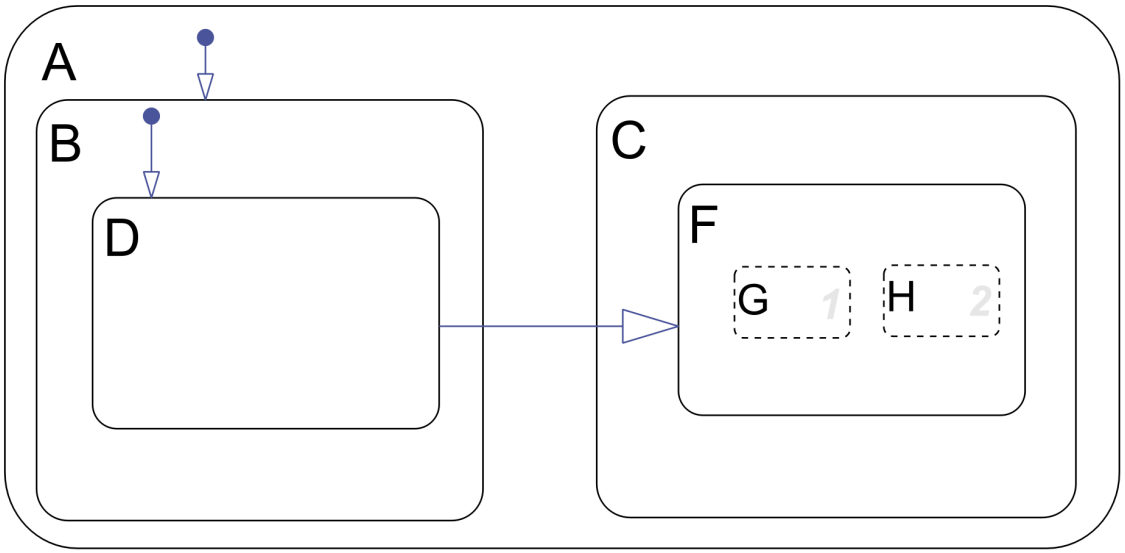


Figure 3.13: States activation

For example, in Figure 3.13, states A, B and D are active and the transition $trans_{DF}$ from state D to F is the first valid transition found during the evaluation described as follows:

1. Execution of the during actions of A then B;

2. Deactivation and execution of exit actions of C, B then A;
3. Execution of the actions of $trans_{DF}$;
4. Activation and execution of entry actions of C, F, G, then H;

3.3 Semantics

In this section, we define the semantics of fStateflow expressed in terms of FTS. This allows us to reuse the efficient model checking algorithms specifically deigned for FTS [CHS+10], [Cla10], [CHSL11], [CCH+12] and [CCS+12]. In order to define this semantics, we systematically present how each element of an FTS is derived from the fStateflow model.

3.3.1 States

The state space of FTS is defined by (1) the state space of the fStateflow model and (2) the set of variables. In the following, we distinguish the set of states S of the FTS model and the set of states S^f of the fStateflow model. More precisely, a state of the FTS is a combination of states of S^f and an evaluation of the set of variables V.

Activation rules, defined in Section 3.2.1, lead us to the definition of S: a state $s_i \in S$ in FTS corresponds to a set of states $S_i^f \in 2^{S^f}$ that are active at the same time and an evaluation of the variables $eval(V)_i \in Eval(V)$. Formally, $S \subseteq 2^{S^f} \times Eval(V)$. Not all combinations of states are allowed though. For example, a substate cannot be active if its superstate is not. Therefore, only a substate of 2^{S^f} determines the actual state space of the FTS. This substate is represented by $Comb(S^f)$. Hence, the size of S is the size of $Comb(S^f)$ multiplied by the size of $Eval(V)$.

The size of $Comb(S^f)$ can be computed using the hierarchy diagram depicted in Figure 3.12 and the function $active : S^f \longrightarrow \mathbb{N}$ defined as follows:

- If a state $s^f \in S^f$ has no substate

$$active(s^f) = 1$$

- If a state $s^f \in S^f$ is decomposed in n **exclusive** substates $(s_1^f, \ldots, s_n^f)$

$$active(s^f) = \sum_{i=1}^{n} active(s_i^f)$$

- If a state $s^f \in S^f$ is decomposed in n **parallel** substates $(s_1^f, \ldots, s_n^f)$

$$active(s^f) = \prod_{i=1}^{n} active(s_i^f)$$

Hence, $active(s_{root}^f)$, where s_{root}^f is the root of the HD that represents the hierarchy of S^f, gives the size of $Comb(S^f) \subseteq 2^{S^f}$.

For example, from the HD depicted in Figure 3.12, one can compute the size of $Comb(S^f)$ as illustrated in Figure 3.14.

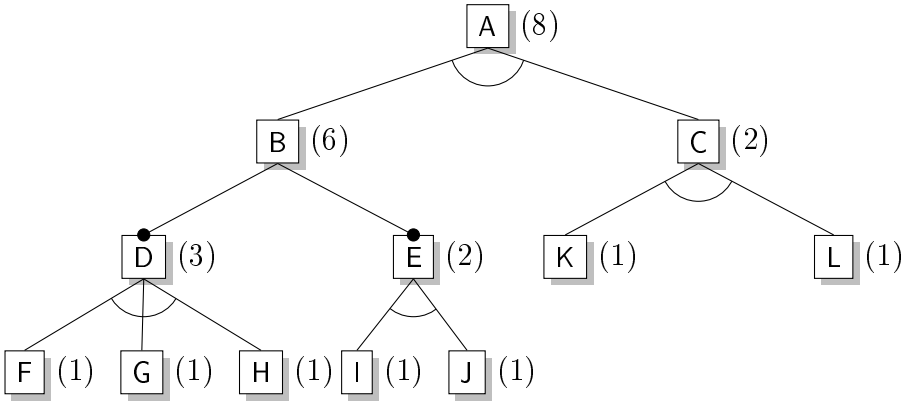


Figure 3.14: Hierachy diagram, computation of the number of set of set of states that can be active at the same time

In this case, from 10 states (S^f) in fStateflow we have $active(A) = 8$ states (S) in FTS. Each of state of S corresponds to a subset of S^f as follows:

$$\begin{array}{lcl}
S^f & = & \{A, B, C, D, E, F, G, H, I, J, K, L\} \\
Comb(S^f) & = & \{s_0^f, s_1^f, s_2^f, s_3^f, s_4^f, s_5^f, s_6^f, s_7^f\} \\
s_0^f & = & \{A, B, D, E, F, I\} \\
s_1^f & = & \{A, B, D, E, F, J\} \\
s_2^f & = & \{A, B, D, E, G, I\} \\
s_3^f & = & \{A, B, D, E, G, J\} \\
s_4^f & = & \{A, B, D, E, H, I\} \\
s_5^f & = & \{A, B, D, E, H, J\} \\
s_6^f & = & \{A, C, K\} \\
s_7^f & = & \{A, C, L\}
\end{array}$$

In addition to states in fStateflow, variables also influence the number of states in FTS. Indeed, from a set of n states in FTS, a given variable that has m different

possible values multiplies the number of states by m. In this thesis, we only use boolean and integer of 32 bits (2^{32} values) as type of variables. Let b and i be be respectively the number of boolean variables and the number of integer variables of V. The size of $Eval(V)$ is thus $2^b \times (2^{32})^i$.

For instance, if the HD depicted in Figure 3.12 is accompanied by 3 boolean and 1 integer variables than we obtain 343.597.383.680 states in term of FTS as described in the following:

$$
\begin{array}{cccccc}
 & |Comb(S^f)| & \times & & |Eval(V)| & \\
= & \text{10 combinations} & \times & \text{3 boolean} & \times & \text{1 integer} \\
= & 10 & \times & 2^3 & \times & 2^{32} \\
= & 10 & \times & 8 & \times & 4.294.967.296 \\
= & \multicolumn{5}{l}{\text{343.597.383.680 FTS states}}
\end{array}
$$

More often than not, only a small portion of this state space will be reachable, and thus explored by the model checking algorithms. In the above example, it may indeed happen that the integer variable only receives a very small subset of value during an execution. The reachable part of the state space is determined by the transition relation.

3.3.2 Transitions

As we defined previously, the set of states in FTS is determined by the combinations of active states and the evaluation of the variables in fStateflow. Hence, a transition between two states in FTS consists in (1) a sequence of transitions between a set of states to another and (2) a set of statements on the variables in fStateflow. In the following, we distinguish the transition relation *trans* of the FTS model and the transition relation $trans^f$ of the fStateflow model. Regarding the set of statements, this is only determined by transitions. Except transition transition actions that are executed when the transition is applied, during actions are executed when no transition is valid and entry, exit actions are respectively executed when the state becomes active and inactive. The set of statements can thus be derived from the set of transitions. First, we define how to compute the list of transitions that exit a given state and its substates. Then, we formally define the transition relation of an FTS.

As mentioned above, a given state s_i in S corresponds to a set of states S_i^f in 2^{S^f} (and an evaluation $Eval_i(V)$). The set of transitions that exits s_i is thus a set of set of transitions that exit states of S_i^f where each set of transitions represents a sequence of transitions. Recall that states of S_i^f are organized following a hierar-

chy, thus if a transition of a superstate is performed, transitions of its substates cannot be performed after that.

Let s^f be an active state in fStateflow and $exit(s^f)$ be the transitions leaving s^f and $during(s^f)$ be a transition to itself. This self-transition represents the possibility for s^f to stay active. Indeed, if any exiting transition is valid, the state executes its during actions as defined in Section 3.2.1 and in this case, the state is not deactivated then activated. This self transition is thus denoted as $s \xrightarrow[during]{[g]a} s$ where g is the negation of the conjunction of all transitions exiting s and a is only the actions of the transition.
The function $Trans$ that associates an active state with a set of sequences of transitions is defined as follows:

- If a state s^f has no substate, then only its transitions can be performed

$$Trans(s^f) = \{exit(s^f)\} \cup \{during(s^f)\}$$

- If state s^f is decomposed in n **exclusive** substates $(s_1^f, \ldots, s_n^f)$ and s_i^f is the active substate

$$Trans(s^f) = \{exit(s^f)\} \cup (\{during(s^f)\} \times Trans(s_i^f))$$

  In this case, the state or its active substate can perform a transition. This is represented by the union of the transitions of s^f and the transitions of its active substate s_1^f.

- If a state s is decomposed in n **parallel** substates $(s_1^f, \ldots, s_n^f)$

$$Trans(s^f) = exit(s^f) \cup (\{during(s^f)\} \times (\overset{n}{\underset{i=1}{\times}} Trans(s_i^f)))$$

  If an active state has parallel substate then all of these substates are active. Hence, either the state performs a transition or all substates perform a transition in a sequence. This is represented by the union of the transitions of s^f and the Cartesian product ($\times$) of the sets of transitions of the substates.

$Trans(s)$ is the set of sequences of transitions of state s and all its active substates. For instance, if states A, B, D, E, F and I are active in the model depicted in Figure 3.11 the set of sequences of transitions $Trans(A)$ is computed as follows:

In the following $A \rightarrow B$ denotes an **exit** transition from state A to state B and $A \circlearrowleft$ denotes the self transition on state A representing the execution of **during** actions.

$$
\begin{aligned}
& Trans(A) \\
= \;& \overbrace{\{exit(A)\}}^{\emptyset} \cup (\{during(A)\} \times Trans(B)) \\
= \;& \{A \circlearrowleft\} \times Trans(B) \\
= \;& \{A \circlearrowleft\} \times (\{exit(B)\} \cup (\{B \circlearrowleft\} \times Trans(D) \times Trans(E))) \\
= \;& \{A \circlearrowleft\} \times (\{B \to C\} \cup (\{B \circlearrowleft\} \times (\overbrace{\{exit(D)\}}^{\emptyset} \cup (\{during(D)\} \times Trans(F))) \times (\overbrace{\{exit(E)\}}^{\emptyset} \cup (\{during(E)\} \times Trans(I))))) \\
= \;& \{A \circlearrowleft\} \times (\{B \to C\} \cup (\{B \circlearrowleft\} \times (\{D \circlearrowleft\} \times Trans(F)) \times (\{E \circlearrowleft\} \times Trans(I)))) \\
= \;& \{A \circlearrowleft\} \times (\{B \to C\} \cup (\{B \circlearrowleft\} \times (\{D \circlearrowleft\} \times (\{exit(F)\} \cup \{during(F)\})) \times (\{E \circlearrowleft\} \times (\{exit(I)\} \cup \{during(I)\})))) \\
\\
= \;& \{A \circlearrowleft\} \times (\{B \to C\} \cup (\{B \circlearrowleft\} \times (\{D \circlearrowleft\} \times (\{F \to G; F \to H\} \cup \{F \circlearrowleft\})) \times (\{E \circlearrowleft\} \times (\{I \to J\} \cup \{I \circlearrowleft\})))) \\
= \;& \{A \circlearrowleft\} \times (\{B \to C\} \cup (\{B \circlearrowleft\} \times (\{D \circlearrowleft\} \times \{F \circlearrowleft; F \to G; F \to H\}) \times (\{E \circlearrowleft\} \times \{I \circlearrowleft; I \to J\}))) \\
= \;& \{A \circlearrowleft\} \times (\{B \to C\} \cup (\{B \circlearrowleft\} \times \{(D \circlearrowleft, F \circlearrowleft); (D \circlearrowleft, F \to G); (D \circlearrowleft, F \to H)\} \times \{(E \circlearrowleft, I \circlearrowleft); (E \circlearrowleft, I \to J)\})) \\
= \;& \{A \circlearrowleft\} \times \{(B \to C); (B \circlearrowleft, D \circlearrowleft, F \circlearrowleft, E \circlearrowleft, I \circlearrowleft); (B \circlearrowleft, D \circlearrowleft, F \circlearrowleft, E \circlearrowleft, I \to J); (B \circlearrowleft, D \circlearrowleft, F \to G, E \circlearrowleft, I \circlearrowleft); \\
& (B \circlearrowleft, D \circlearrowleft, F \to G, E \circlearrowleft, I \to J); (B \circlearrowleft, D \circlearrowleft, F \to H, E \circlearrowleft, I \circlearrowleft); (B \circlearrowleft, D \circlearrowleft, F \to H, E \circlearrowleft, I \to J)\} \\
= \;& \{(A \circlearrowleft, B \to C); \\
& (A \circlearrowleft, B \circlearrowleft, D \circlearrowleft, F \circlearrowleft, E \circlearrowleft, I \circlearrowleft); \\
& (A \circlearrowleft, B \circlearrowleft, D \circlearrowleft, F \circlearrowleft, E \circlearrowleft, I \to J); \\
& (A \circlearrowleft, B \circlearrowleft, D \circlearrowleft, F \to G, E \circlearrowleft, I \circlearrowleft); \\
& (A \circlearrowleft, B \circlearrowleft, D \circlearrowleft, F \to G, E \circlearrowleft, I \to J); \\
& (A \circlearrowleft, B \circlearrowleft, D \circlearrowleft, F \to H, E \circlearrowleft, I \circlearrowleft); \\
& (A \circlearrowleft, B \circlearrowleft, D \circlearrowleft, F \to H, E \circlearrowleft, I \to J)\}
\end{aligned}
$$

The set $Trans(s^f)$ is thus composed of all possible sequences of transitions. However, all sequences may not be valid. Indeed, this method does not take count of the condition of each transition. Moreover, given that it is a sequence of transitions, the validity of the i^{th} transition depends on the application of the $i-1$ first transitions of the sequence. Recall that execution of fStateflow model modifies states activity and variables values. These modifications are defined as functions as follows:

- $apply : (S \times trans') \longrightarrow S$ is the partial function that associates (1) a state in FTS and (2) a transition in fStateflow with the state in FTS resulting of the application of the transition;
- $execute : (Eval(V) \times Stmnt(V)) \longrightarrow Eval(V)$ is the partial function that associates (1) an evaluation of variables V and (2) a set of statements with an evaluation of the variables resulting of the execution of the statements;

For a proper understanding of the transition relation, we first define the case where transition in FTS consists in one transition in fStateflow. Then, we define the general case where transitions in FTS are sequences of transitions in fStateflow.

A transition from a state $(S, eval(V)_i)$ to a state $(S', eval(V)_{i+1})$ in FTS exists iff

1. there is a transition $s^f \xrightarrow{[g]a} s'^f$ where $s^f \in S$ and $s'^f \in S'$;
2. this transition is valid, i.e., the predicate g over V is not false considering the evaluation of the variables $eval(V)_i$, denoted by $eval(V)_i \models g$;
3. S' is the resulting state of the application of the transition $s^f \xrightarrow{[g]a} s'^f$ on the state S and
4. $eval(V)_{i+1}$ is the resulting evaluation the variables obtained by the execution of the statements a on $eval(V)_i$.

Formally, in the case of single transition in fStateflow, the transition relation $trans \subseteq S \times Stmnt(V) \times S$ is the smallest relation satisfying the following

$$\frac{\begin{array}{ll} & s^f \in S \\ \wedge & s^f \xrightarrow{[g]a} s'^f \\ \wedge & eval(V)_i \models g \\ \wedge & S' = apply(S, s^f \longrightarrow s'^f) \\ \wedge & eval(V)_{i+1} = execute(eval(V)_i, a) \end{array}}{(S, eval(V)_i) \xrightarrow{a} (S', eval(V)_{i+1})}$$

However, more often than not, transitions in FTS are sequences of transitions in fStateflow. Let n be the number of transitions of a give sequence. A given transition$(S, eval(V)_i) \xrightarrow{a} (S', eval(V)_{i+n})$ in FTS exists iff, for all $k : 0 \leq k \leq n-1$ the k^{th} transition is valid after the application of the $k-1$ transitions. Similarly, the set of states S' is the results of the application of the sequence of transitions. Formally, the general transition relation $trans \subseteq S \times Stmnt(V) \times S$ is the smallest relation satisfying the following

$$\frac{\begin{array}{ll} & \forall k, 0 \leq k \leq n-1 : s_k^f \xrightarrow{[g_k]a_k} s_k'^f \\ \wedge & s_k^f \in S \\ \wedge & eval(V)_{i+1+k} = execute(eval(V)_{i+k}, a_k) \\ \wedge & eval(V)_{i+k} \models g_k \\ \wedge & a = \bigcup_k a_k \\ \wedge & S' = apply(...(apply(apply(S, s_0^f \longrightarrow s_0'^f), s_1^f \longrightarrow s_1'^f), ...), s_{n-1}^f \longrightarrow s_{n-1}'^f) \end{array}}{(S, eval(V)_i) \xrightarrow{a} (S', eval(V)_{i+n})}$$

3.3.3 Feature Expressions

From the definition of transition in terms of FTS, we can define the feature expression of a given transition in FTS as the conjunction of feature expressions of corresponding transitions in fStateflow. Let $(S, eval(V)_i) \xrightarrow{a} (S', eval(V)_{i+n})$ be a transition in FTS where n is the number of corresponding transitions in fStateflow. The derived feature expression f is

$$f = \bigwedge_{i=0}^{n-1} \gamma(s_i^f \longrightarrow s_i'^f)$$

where $\gamma : trans^f \longrightarrow \mathbb{B}(F)$ associates each transition in fStateflow with a feature expression over F;

Chapter 4

Case study: the PCA Infusion Pump SPL

In this chapter, we present a case study of a Patient Controlled Analgesic (PCA) Infusion Pumps SPL. Specification of this system was extracted from a requirements document written by the Critical System Group of the University of Minnesota. This document provides a detailed description of the functionalities and constraints of PCA Infusion Pump in its intended environment of use. PCA Infusion Pump is a type of pump used to infuse analgesic in intravenous. Several infusion modes can be configured by a clinician. One of these mode allows patient to request an extra amount of analgesic if he feels too much pain. However, some of these modes imply the presence of extra components such as a control panel or a remote control. In order to reduce the cost of production, these extra components can be removed and, therefore, modes that require them. Hence, it would be interesting to consider PCA Infusion Pump as an SPL. In addition, PCA Infusion Pumps are considered as highly safety-critical systems and they must be verified in order to prevent error. This case-study is thus a good candidate for SPL model checking.

In the following, we do not mention the "hardware" features and we focused on "software" features. Indeed, the purpose of this work is to model the behaviour of this SPL and hardware differences are not relevant for this case study. At first, we will describe what is an PCA Infusion Pump and its potential functionalities. Then, we define the variability of these pumps. Finally, we take a look on how model the behaviour of this SPL using Stateflow.

4.1 Description

A PCA Infusion Pump is a type of pumps allowing patient in pain to administer their own painkiller by intravenous. Infusion pumps have to be configured by

a clinician with the patient information, the drug information and the infusion parameters like the Volume To Be Infuse (VTBI) and the maximum duration. Clinicians can access to a control panel thanks to which they can start or stop the system and initiate or inhibit the infusion. During infusion, patients can partially control the amount of analgesic that will be injected in order to suppress pain. Infusions are ended when the configured duration of the infusion is exceeded or when the volume to be infused is reached. In the following, we define all the functionalities.

At first, infusions are programmed by a clinician using modes as basal or square bolus. Basal is the regular mode configured by the clinician. This mode requires a constant flow rate as parameter (for example, 5ml/h). Square bolus is the fluctuating mode where the flow rate rises up to a given amount for a given period of time every x minutes or hours - a bolus denotes an extra amount of drug. For instance, the flow rate rises up to 10 ml/h for 15 minutes every hour.

Patients can request an extra amount of drug if he is in pain. In this mode, called Patient Bolus, the pump infuses a bolus represented by a determined flow rate for a given period of time. For example, when the patient commands a bolus, the flow rate rises up to 20 ml/h for 10 minutes. This functionality can be restrained by a maximum number of bolus. When a patient bolus is over, there is a lockout period during which patients cannot request another bolus.

Clinicians can also request a bolus without restriction in the number of requests triggering the Clinician Bolus mode. This mode requires two parameters: a flow rate and a period of time; e.g. 15 ml/h for 20 minutes. Clinicians are allowed to pause the infusion if necessary, which sets the flow rate to 0 ml/h.

In summary, that parameters that have to be initialised before any infusion are:

- Total VTBI.
- Maximum duration.
- Flow rates of modes: Basal, Square Bolus, Patient Bolus, Clinician Bolus and Pause.
- Duration of modes: Square Bolus, Patient Bolus ans Clinician Bolus.
- Interval between Square Bolus.
- Lockout period after Patient bolus.

- Maximum number of Patient Bolus.

More than one mode can be activated at the same time. In this case, the actual flow rate is determined by the mode with the highest priority. For example, during the basal mode, a patient can request a bolus and the bolus mode is selected instead of the basal mode. The hierarchy between these modes defines the priority of each mode:

$$\text{Pause} > \text{Patient Bolus} > \text{Clinician Bolus} > \text{Square Bolus} > \text{Basal}$$

Thus, if the patient requests a bolus during the square bolus mode, the infusion pump will select the patient bolus mode and adapt the flow rate. Then, if a clinician requests a bolus during a patient bolus mode, the flow rate will not change until the end of the patient bolus since patient bolus has more priority than clinician bolus.

Besides modes, pumps are also equipped with hazard detectors grouped in two categories. The first category monitors environment parameters as temperature, humidity or air pressure. If one of the parameters is out of a certain range, an environment hazard is reported.

The second category controls infusion parameters. These concern the monitoring of the pipe linking the pump to the needle in order to avoid issues such as occlusion, air in line, free flow or reverse delivery. A hazard is respectively reported iff

- the pipe is twisted and the infusion is blocked;
- there is an air bubble in the pipe;
- the flow rate becomes out of control; or
- the flow is reversed and the pump siphons blood from patient.

Additionally, pumps provide a reservoir in which the analgesic is poured. Hazards are reported when the level of the reservoir is too low than a certain value and when the reservoir is empty. The flow rate and the VTBI are also controlled in order to avoid overdose problems. That is, flow rate cannot be higher or lower than a certain range and VTBI cannot exceed a certain value before the end of the infusion.

All of these detectors trigger alarms classified in three levels of severity. Alarms change the flow rate of the infusion pump depending on their level and are deactivated by resolving issues. More precisely, the consequences of these alarm levels are the followings:

Level 1: the flow rate is adjusted to the basal flow rate and cannot increase or decrease until the alarm is deactivated.

Level 2: the infusion is paused until the alarm is deactivated.

Level 3: the current infusion is cancelled and a new one has to be configured.

Besides modes and hazard detectors that are main components, infusion pumps have a list of auxiliary functionalities as:

Logging: All operations and events are recorded in a log file.

Drug Library: A set of different analgesics are available for infusion.

Access control: Accesses of the pump software are protected by a code.

Print: Logs or statistics can be printed.

Scanning: Patient information, drug information and prescriptions can be scanned to accelerate and ease the configuration of infusions.

In a given pump, all these functionalities may not be needed. In order to reduce de development cost, it is therefore essential to discard the unneeded ones.

All these variations lead us to the definition of a PCA Infusion Pump variability model. Basically, each variation gives rise to a new feature. The following section describes their dependencies as expressed in an *FD*.

4.2 Variability

First, we describe the top-level features. Then, we describe the decomposition of each top-level features.

4.2.1 Top-level features

We can enumerate seven top-level features that can be present in a software product of the line: Infusion Modes, Hazard Detection, Logging, Drug Library, Access Control, Print and Scanning. The first two compose the basics of an Infusion Pump and must be present in every product. On the one hand, a pump needs at least one mode of infusion to correctly perform. On the other hand, as a critical system, an infusion pump cannot malfunction and all issues must be avoided. The other five features are considered optional because they are not crucial in the functioning of a pump per se. Graphically, the root of this FD, named *PCA Infusion*

Pump, has an AND-relationship with its child features but only Infusion Modes and Hazard Detection are mandatory; the rest is optional. Figure 4.1 depicts this relationship.

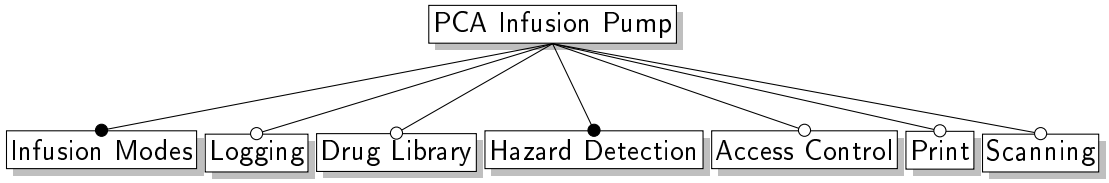


Figure 4.1: Main features

4.2.2 Hazard detections

Hazard detections can be separated in two categories: environment hazards and infusion hazards. The first category concerns issues related to temperature, humidity and air pressure whereas the second includes issues related to the infusion parameters (VTBI and Flow rate), the pipe (Free flow, Air in Line, Reverse Delivery and Occlusion) and the reservoir (Low and Empty). Even one hazard may be fatal for the patient, thus, every products must integrate all these detection abilities. This requirement is materialised as an And-relationship where all features are mandatory. For clarity, the main feature Hazard Detection is firstly decomposed into two child features Environment and Infusion which are themselves decomposed into aforementioned features. Figure 4.2 shows this relationship.

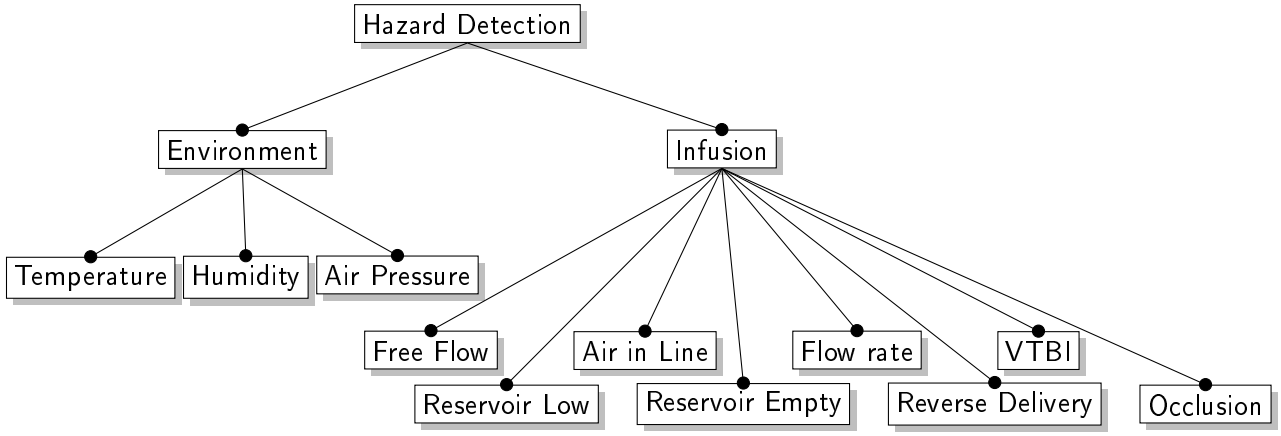


Figure 4.2: Hazard detections features

4.2.3 Scanning options

If the optional Scanning feature is selected, several type of information may be scanned instead of being entered manually. A scanner can read patient information, drug information or prescription details in order to accelerate the configuration process. Figure 4.3 illustrates this OR-relation.

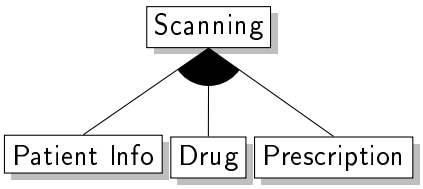


Figure 4.3: Scanning features

4.2.4 Infusion modes

The most important variabilities lie in the different infusion modes. Only two modes are mandatory: Basal and Pause. Basal is the basic mode configured for each infusion with a determined flow rate. In addition, clinicians need a mode to set the flow rate at 0 ml/h in order to inhibit infusion in case of problem. Besides these basic modes, there are three optional modes that infuse bolus of analgesic: Patient Bolus, Clinician Bolus or Square Bolus. These are grouped into an OR-relationship with bolus feature. Patient Bolus may also be restricted by a lockout period represented by an optional child feature. Figure 4.4 depicts this relationship.

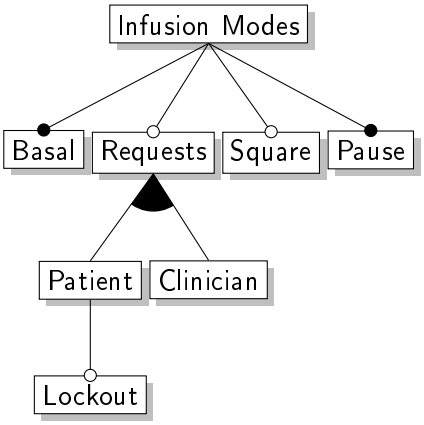


Figure 4.4: Infusion Modes features

By assembling all features together, we obtain the Feature Diagram in Figure 4.5 representing the variability model of the PCA Infusion Pump SPL.

Figure 4.5: Feature Diagram of a PCA Infusion Pump SPL

4.3 Behavioural modelling with Stateflow

We now model the behaviour of the infusion pump SPL using the fStateflow language. Since the model is too large to be fully described, we focus on the modelling of the infusion modes. There are five different modes (see Section 4.2.4) such as pause, basal, patient bolus, clinician bolus and square bolus. Pause and basal are mandatory whereas the others are optional. At the beginning of an infusion, the active mode is basal. Then, depending on the configuration of the pump and the requests of the patient and clinicians, the active mode will switch from basal to another mode and so on until the end of the infusion. For recall, more than one mode can be active at the same time and, in this case, the actual flow rate is determined by the mode with the highest priority. This priority is defined as follows:

$$\text{Pause} > \text{Patient Bolus} > \text{Clinician Bolus} > \text{Square Bolus} > \text{Basal}$$

4.3.1 Variables

Before describing modes, we define the list of variables used in this model and their meaning in the system. We can separate the variables in four different categories such as configured parameters, buttons, counters and additional variables that represent features.

The first category groups all the parameters that clinicians must configure before any infusion like the duration and the flow rate of each mode. These parameters are defined in Table 4.1.

Name	Description
FlowRate_ pause	flow rate of the pause mode
FlowRate_ cbolus	flow rate during the clinician bolus mode
FlowRate_ sbolus	flow rate during the square bolus mode
FlowRate_ pbolus	flow rate during the patient bolus mode
Duration_ cbolus	maximum period of time of a bolus requested by a clinician
Duration_ sbolus	maximum period of time of a bolus during the square bolus mode
Duration_ pbolus	maximum period of time of a bolus requested by the patient
sbolus_ interval	period of time between two square bolus
NumberMax_ pbolus	maximum number of bolus that a patient can request
LockOutPeriod_ pbolus	minimum period of time between two occurrences of the patient bolus mode
Duration_ total	maximum total duration of an infusion
VTBI_ total	maximum volume to be infused during an infusion

Table 4.1: List of parameters to be configured by a clinician before any infusion

The second category represents the different buttons used to control the pump. Basically, the value of a given button in set to 1 when this button is pushed on and 0 otherwise. Buttons are listed in Table 4.2.

Name	Description
Infusion_ initiate	starts the infusion after configuration or releases pauses
Infusion_ inhibit	pauses the infusion.
cbolus_ req	requests a clinician bolus
pbolus_ req	requests a patient bolus

Table 4.2: List of buttons used to control the infusion pump

The third category are local variables handling parameters and timers. Table 4.3 lists all these variables.

Name	Description
Actual_ Infusion_ Duration	period of time elapsed from the start of the infusion
Actual_ Volume_ Infused	amount of drug infused since the start of the infusion
flow	current flow of the infusion
number_ pbolus	number of requests of patient bolus executed
cbolus_ dur_ timer	period of time elapsed from the start of the clinician bolus mode
sbolus_ dur_ timer	period of time elapsed from the start of the square bolus mode
pbolus_ dur_ timer	period of time elapsed from the start of the patient bolus mode
lock_ timer	period of time elapsed from the start of the lock out period

Table 4.3: List of local variables

Finally, the fourth category is composed of a list of boolean variables. Each of which corresponds to a feature. They are used to form feature expressions. Each boolean has the name in upper case of the feature that it represents such as : *SQUARE*, *CLINICIAN_ BOLUS*, *PATIENT_ BOLUS* and *LOCKOUT*.

4.3.2 Infusion modes

We followed two systematic procedures to model the behaviour of the infusion modes: one based on exclusives states and the other based on parallel states.. As a first attempt, exclusive states have been used to represent each mode with transition from each state to all the other states. As they are exclusive, there is only one state active at a time which represents the active mode. However, this representation becomes more and more complex with the number of states (i.e. modes). Formally, with n states we have $n \times (n-1)$ transitions. Figure 4.6 depicts this attempt with $n = 5$ states and $n \times (n-1) = 20$ transitions. If a new mode is created, adding a new state to this model would be very tedious.

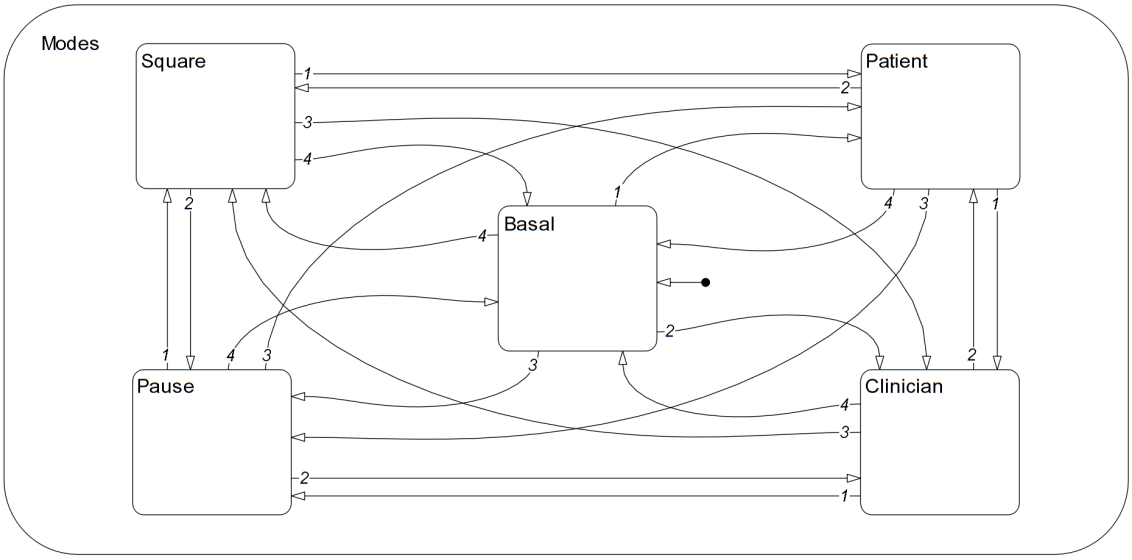


Figure 4.6: Exclusive states attempt

For scalability reasons, we came up with the second alternative based on parallel states. In this case, each mode has a corresponding parallel state containing two exclusive states, On and Off, that represent the activity of the mode. An additional state, called $Umpire$[1], sets the flow rate depending on the active modes and their priority. This state is decomposed in substates where each of these represents an infusion mode. Even if there are more states in the model, there are much less transition to handle which makes this model less complex. All these parallel states are substates of the state labelled $Therapy$ which is activated when the infusion is initiated. In the following, each substate representing an infusion mode is described separately by defining transitions between On and Off states.

Basal is the first mandatory mode. This mode is active from the beginning of the infusion, thus On is directly activated. This mode remains active until the maximum duration is reached, that is, when variables $Actual\_Infusion\_Duration$ and $Duration\_total$ are equal (see Figure 4.7).

[1]In American Football, the umpire is the referee behind the defensive line and linebackers that observes the scrimmage.

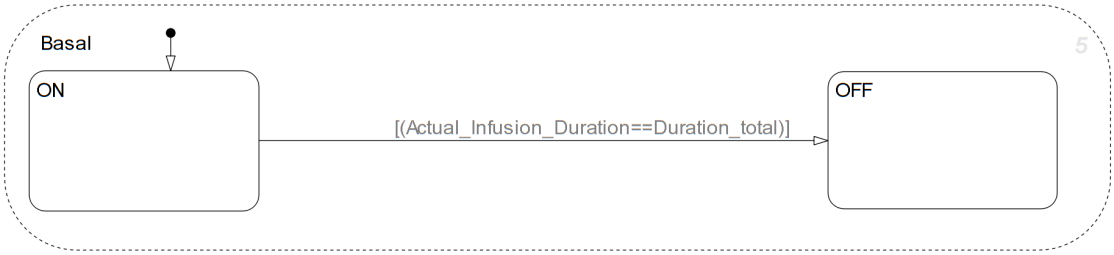


Figure 4.7: Basal mode

The second mandatory mode is the pause mode depicted in Figure 4.8. Initially, this mode is set on Off. The first transition from Off to On is labelled with the condition $Infusion\_inhibit > 0$. The variable $Infusion\_inhibit$ represents the pause button that a clinician can push on. This variable is set to 1 if the clinician pushes on the button and it is set to 0 otherwise. The second transition represents alarm triggers (see section 4.3.3). Basically, this transition becomes valid iff an alarm of level 2 is triggered. The state Off becomes active again iff, (1) similarly to inhibition button, the clinician pushes on the button to initiate the infusion ($Infusion\_initiate > 0$) and, (2) the total infusion duration has not been reached ($Actual\_Infusion\_Duration < Duration\_total$).

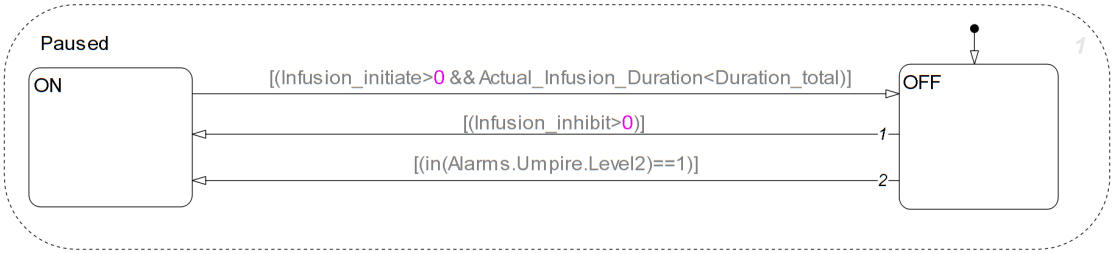


Figure 4.8: Pause mode

The square bolus mode is initialised on Off where the corresponding timer is set to 0. As an optional feature, this mode can become active iff the square feature is present, i.e. iff the feature expression $SQUARE$ is satisfied. In the boolean condition, the modulo operator %% is used to satisfy the minimum period of time between two square bolus. For example, if $sbolus\_interval = 10$, this transition becomes valid every 10 minutes. Then, the variable $sbolus\_dur\_timer$ is incremented as long as state On remains active, that is, no alarm of level one is triggered in the mean time. Once the variable $Duration\_sbolus$ is equal to $sbolus\_dur\_timer$, On becomes inactive and square bolus mode comes back to Off. Figure 4.9 illustrates this behaviour.

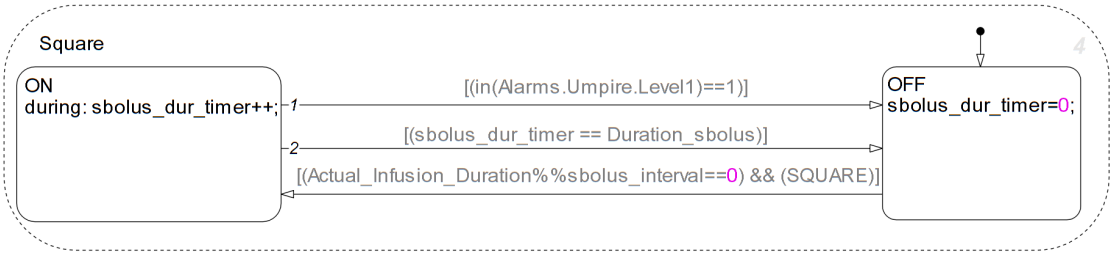


Figure 4.9: Square bolus mode

Clinician bolus mode is very similar to square bolus in its modelling. The initial state is also Off and the two transition from On to Off are the same except for variables $sbolus\_dur\_timer$ and $Duration\_sbolus$ that are respectively replaced by $cbolus\_dur\_timer$ and $Duration\_cbolus$. The condition and the feature expression of the transition from Off to On is also different. In this case, the feature expression refers to the clinician bolus mode $CLINICIAN\_BOLUS$ and the condition is satisfied if $cbolus\_req > 0$ which means that the clinician has pushed on its request button. Figure 4.10 depicts this mode.

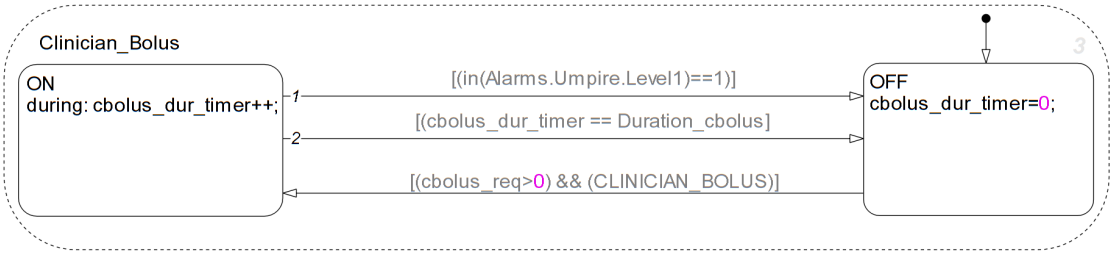


Figure 4.10: Clinician bolus mode

The last mode is the patient bolus. This mode is exactly the same as the clinician bolus mode with an additional optional feature: the lockout period of time between two occurrences of this mode. This feature is represented by another substate called $Lockout$. The behaviour of this mode is the following. Initially, the Off state is active and $pbolus\_dur\_timer$ is initialised to 0. Then, the transition from Off to On is valid iff (1) its feature expression, $PATIENT\_BOLUS$, is satisfied and (2) its condition is evaluated to true. It means that the patient has to make a request ($pbolus\_req > 0$) and the maximum number of patient bolus has not yet been reached ($number\_pbolus < NumberMax\_pbolus$). When On becomes active, the number of patient bolus is incremented ($entry : number\_pbolus++$) and the patient bolus timer increases as long as On remains active. There are two pairs of transitions exiting On that have exactly the same condition but their

feature expression are opposed. The pair of transitions with $LOCKOUT$ as feature expression targets the state *Lockout* and the other pair targets the Off state. Hence, if the timer reaches its limit ($pbolus\_dur\_timer == Duration\_pbolus$) or if an alarm of level 1 is triggered, the execution leaves state On to move to *Lockout* or Off depending on the presence of absence of feature Lockout. If *Lockout* state is activated, the lockout timer is initialised to 0 and it will increase until it reaches its limit ($LockOutPeriod\_pbolus$). Then, this mode goes back to state Off.

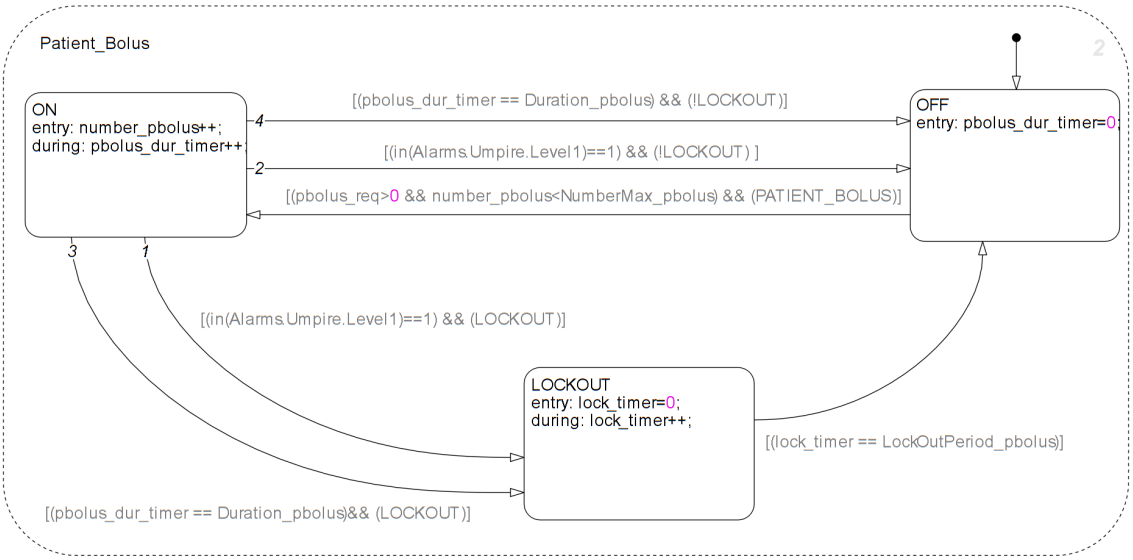


Figure 4.11: Patient bolus mode

We have described all the different modes that can be active at a time. As mentioned before, we need an additional state that analyses which modes are active and set the flow rate depending on the active mode with the highest priority. This state, called $Umpire$, is the last parallel substate in the order of execution. Hence, at each step, all modes first update their activity before the $Umpire$ checks which mode is active. The behaviour of this state is modelled as an "*if-then-else-if-then-else-...*" construct using junctions and each mode is represented by an exclusive state in which the flow is adjusted. At first, the umpire checks if the pause mode is active, i.e., if the active substate of $Pause$ is On ($in(Paused.On) == 1$). If not, it checks the activity of the Patient_Bolus mode (and the feature expression), and so on from the mode with the highest priority to the one with the lowest. If a mode is active, then its corresponding state is activated and the flow rate is adjusted for this mode. The first step ends here. The second step begins by increase the actual volume infused by the mode flow rate as an exit action. Then, the actual infusion duration is incremented as a transition action. Finally, the state checks again the

activity of the modes and the process starts again. This state is illustrated in Figure 4.12.

For instance, let us assume that patient bolus and square bolus ate the only active modes, and that features *PATIENT_BOLUS* and *SQUARE* are enabled. The execution steps of the *Umpire* state are the following:

Initialisation

1. The initial transition is taken to reach the biggest junction.

2. The first exiting transition is not valid because pause mode is not active.

3. The second exiting transition is valid as there is no condition.

4. This transition is performed to reach another junction.

5. The first exiting transition is valid since (1) patient bolus mode is active and (2) feature *PATIENT_BOLUS* is enabled.

6. This transition is performed, and the system reaches state *Umpire.Patient_Bolus*.

7. The entry action is executed and the flow rate is adjusted to *FlowRate_pbolus*.

Execution step

1. *Umpire.Patient_Bolus* is deactivated.

2. The exit action is executed and the actual infused volume is increased by the flow rate of the mode.

3. The four next exiting transitions are performed to reach the biggest junction and the actual infusion duration is incremented by one as the transition action.

4. Repeat steps 2 to 7 in the initialisation.

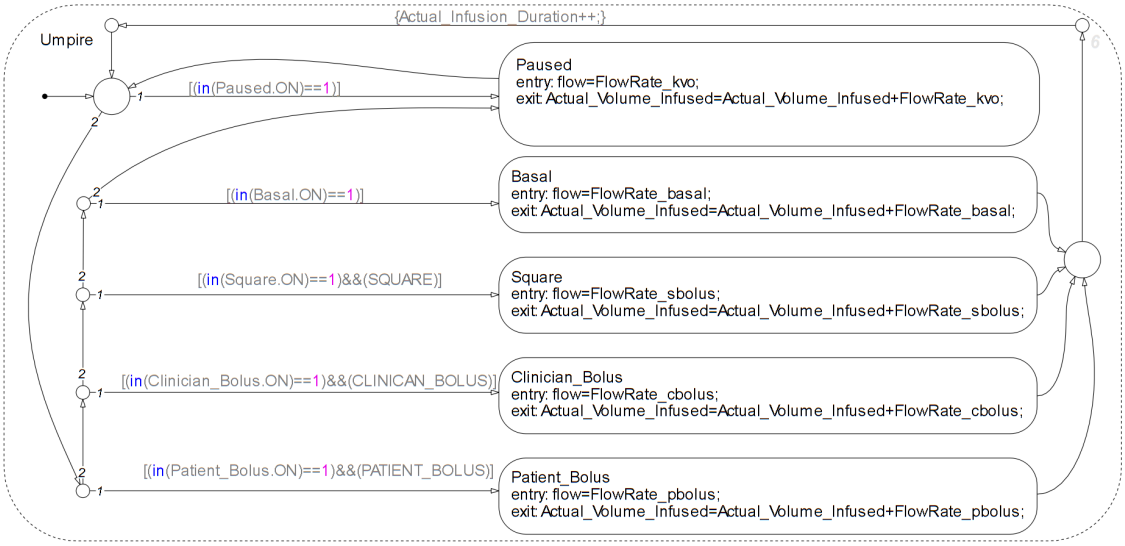


Figure 4.12: Umpire state for infusion modes

4.3.3 Alarms

As described in Section 4.2.2, an infusion pump is equipped with hazard detection capabilities, which can trigger different alarms. These alarms are classified in three levels of severity as follows

Level 1: Low reservoir

Level 2: Empty reservoir, Flow rate and VTBI

Level 3: Air in line, Reverse delivery, Occlusion and Free flow

The modelling of the alarms is very similar to the infusion modes. For each hazard detection, there is a state composed of two substates representing whether the hazard is detected (On) or not (Off). In addition to detectors, there is also an *Umpire* that activates the right level of alarms represented by four states such as *Level1*, *Level2*, *Level3* and *Ok* if there is no problem. The *Umpire* state is illustrated in Figure 4.3.3.

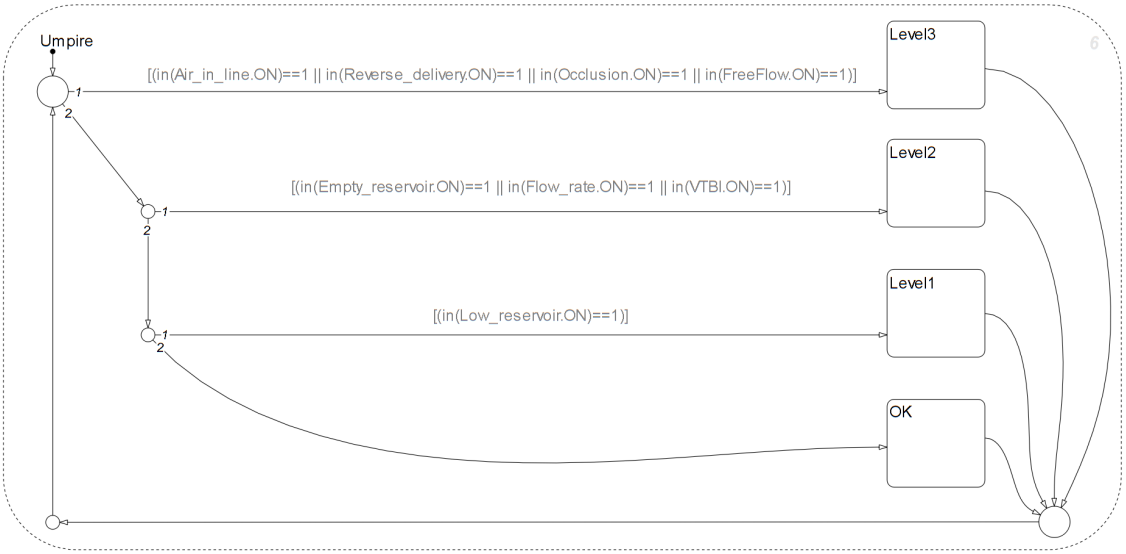


Figure 4.13: Umpire state for alarms

Chapter 5

Implementation

In this chapter, we describe how FTS states and transitions are derived from the fStateflow model. Then, we provide some optimizations of computations and their evaluation.

5.1 fStateflow model

Each fStateflow project is saved in an .mdl file that can be parsed to extract the model. The parsing of a .mdl file consists in three phases. First, the file is filtered to remove useless data such as parts of data that do not concern the fStateflow model or graphics coordinates of states, etc. Then, the filtered file is parsed to obtain a list of states, a list of transitions and a list of variables. Finally, from states list and transitions list, the hierarchy of states is built and states are linked together following transitions. The implemented fStateflow model is represented by the class diagram depicted in Figure 5.1.

Figure 5.1: Class diagram of a fStateflow model

In the following, we describe the implementation in C of each fStateflow elements such as states, transitions and variables.

5.1.1 State

A state is implemented as a structure with 5 attributes and 5 pointers. State attributes are the following:

- id of the state,
- name of the state,
- decomposition of the state: either *Exclusive* or *Parallel*,
- order of the state in the case where it belongs to a parallel decomposition (otherwise its order is set to 0) and
- offset of the state used for FTS states defined later.

Each state has also five pointers to (1) a list of states that represents its substates, (2) a state that represents the superstate, (3) a list of exiting transitions, (4) a during transition and (5) a syntax tree that represents its sets of actions.

5.1.2 Transition

A transition is a structure with three attributes defining its id, its type (initial, during or exit) and its order. In addition, transition has pointers to its source state, its target state and a syntax tree that represents its condition, its feature expression and its set of actions.

These representations of states and transitions allow us to keep only a pointer to a state (initially the top-level state) and move from state to state following the hierarchy of the transitions.

5.1.3 Variables

Regarding variables, these are organized in a list where each variable is a structure with six attributes defined as follows:

- id of the variable,

- name of the variable,
- scope of the variable: either *Constant*, *Undefined* or *Local*,
- type of the variable: either *boolean* or *integer*,
- value of the variable if its scope is *Constant*, 0 otherwise and
- offset of the variable used for FTS states defined later.

5.2 FTS model

5.2.1 States

As we defined in Section 3.3, a state in terms of FTS corresponds to a set of states that are active at the same time in fStateflow and an evaluation of the variables. From this definition, we can represent a state of FTS by a chunk of adjacent memory space, called payload, representing the concatenation of the activity of each state and the value of the variables. The size and the value of the memory allocated for of each state and each variables are the follows:

- One byte for each state. This byte is set to 1 if the state is active and 0 otherwise.
- One byte for each boolean variable. This byte is set to 1 if the value of the variable is true and 0 otherwise.
- Four bytes for each integer variable. These bytes are set to the value of the variable.

The position of each states and variables in the payload are saved into their respective offset attribute. Let n be the number of states, b and i be the numbers of boolean and integer variables. The total size of allocated memory for each FTS state is thus $(n + b + 4i)$ bytes. This representation allow us to optimize the comparison of two states. Instead of comparing each state activity and each value of variable, we use the C function *memcmp* that compares directly adjacent bytes in the memory.

Once the payload size calculated, and the memory allocated, the initial FTS state can be created. The top-level state of the fStateflow model is activated and, depending on its decomposition, one or all substates are activated in turn, etc. Every time a state becomes active, its corresponding byte in the payload is updated. Activation of a state may trigger some entry actions that also update value

of variables directly in the payload. Once this first activation process is over, the initial state of FTS is created and the set of transitions can be computed.

5.2.2 Transitions

The set of transitions in FTS, that is, the set of sequences of transitions in fStateflow, is derived from the set of active states as defined in 3.3.2. However, all sequences of transitions are not valid and these have to be checked separately. For each sequence, a copy of the current FTS state is created. Then, the first transition of the first sequence is evaluated: if it is false, the checking process for this sequence is stopped and the next one is checked. Otherwise, the transition is applied on the copy of the FTS state and the next transition is evaluated, and so on until the end of the sequence. Iff all transitions have been evaluated to true or undefined, this sequence is added to the set of sequences that could be applied on the current FTS state. In the case where all transitions of a sequence are evaluated to true, following the semantics of fStateflow, next sequences cannot be applied and therefore, they are not checked. Otherwise, it at least one transition of the sequence is undefined, the next sequence is checked.

5.3 optimization

A first space optimization is to take only local variables into consideration to compute the payload size. Constants, by definition, are never modified and inputs are always undefined, therefore, their value are the same in all FTS states.

A second optimization concerns the validity checking of sequence of transitions. Let $(t_1, t_2, \ldots, t_{n-1}, t_n)$ be a sequence of n transitions. As mentioned above, if the transition t_i with $1 \leq i \leq n$ is evaluated to false, this sequence is not a possible transition in terms of FTS. Hence, if another sequence begins with the subsequence $(t_1, t_2, \ldots, t_i)$, t_i will be evaluated to false again. Therefore, all sequences that begin with the subsequence $(t_1, t_2, \ldots, t_i)$ are not possible sequences of transitions.

For example, let us consider the following set of sequences computed at the end of Section 3.3.2:
$\{(A \circlearrowleft, B \rightarrow C)$;
$(A \circlearrowleft, B \circlearrowleft, D \circlearrowleft, F \circlearrowleft, E \circlearrowleft, I \circlearrowleft)$;
$(A \circlearrowleft, B \circlearrowleft, D \circlearrowleft, F \circlearrowleft, E \circlearrowleft, I \rightarrow J)$;
$(A \circlearrowleft, B \circlearrowleft, D \circlearrowleft, F \rightarrow G, E \circlearrowleft, I \circlearrowleft)$;
$(A \circlearrowleft, B \circlearrowleft, D \circlearrowleft, F \rightarrow G, E \circlearrowleft, I \rightarrow J)$;
$(A \circlearrowleft, B \circlearrowleft, D \circlearrowleft, F \rightarrow H, E \circlearrowleft, I \circlearrowleft)$;

$(A \circlearrowleft, B \circlearrowleft, D \circlearrowleft, F \rightarrow H, E \circlearrowleft, I \rightarrow J)\}$

If the second transition of the second sequence is evaluated to false, other sequences beginning with $(A \circlearrowleft, B \circlearrowleft)$ will evaluate the second transition to false. Therefore, we do not need to check the other sequences and only the first one is possible.

5.4 Evaluation

To evaluate the impact of the aforementioned optimizations, we verified the model described in Chapter 4. This model is composed of 28 states and 14 integers. The characteristics of the computer on which the model checker is executed are the following:

OS	Ubuntu 11.10
CPU	Intel Core I3 2.13GHz
	Quad core
	64-bit
RAM	6 Go

Executions of the model checker are described by the number of explored states, the number of re-explored states, and two types of execution time: Real and User. Real refers to wall clock time, i.e. time from start to finish of the call and user refers to the amount of CPU time spent in user-mode code within the process.

The first table represents the results of three executions without optimizations.

Explored states	45298	45298	45298
Re-explored states	15634	15634	15634
Real time	$2'15''.424$	$2'14'', 706$	$2'24'', 580$
User time	$1'1''.164$	$1'0'', 692$	$1'0'', 972$

The second table represents the results of three executions with optimizations.

Explored states	45298	45298	45298
Re-explored states	15634	15634	15634
Real time	$1'59''.247$	$1'53'', 429$	$1'53'', 713$
User time	$36''.154$	$35'', 874$	$36'', 546$

As we can see, the number of explored/re-explored states are not modified by optimizations. However, execution times are different. In average, the real execution time without optimization is $2'18''$ whereas it takes only $1'55''$ with optimizations. The difference is bigger for user time with $1'1''$ without optimizations and only $36''$ when optimizations are applied which represents a decrease of 40%.

Chapter 6

Review and Perspectives

6.1 Summary

We demonstrated by this thesis the possibility to adapt high level language for the modelling of SPL behaviour. First, we created a variability-aware extension of Stateflow by integrating feature expressions on transitions. Then, we defined the semantics of this language in terms of FTS which permits to use FTS model checking techniques on fStateflow models. Finally, we modelled from scratch the behaviour of a PCA Infusion Pump SPL showing that fStateflow is applicable on real system. FTS seems thus to be enough flexible to act as a unified semantics for any SPL behavioural model.

During the implementation, we also faced difficulties such as handling of parallelism and hierarchy in fStateflow. Indeed, these concepts involve major syntactic and semantic differences between fStateflow and FTS. For example, parallelism is the reason that a transition in FTS consists in a sequence of transitions in fStateflow. Hierarchy and parallelism also imply that several states can be active at a time. Then, Stateflow has also some particular semantics rules that differ from other state-based languages such as the order of evaluation of exiting transitions and the different types of actions. Finally, some optimizations were required in order to reduced execution time and used memory space.

6.2 Critical Outlook

We present in this section, the strengths and the weaknesses of the fStateflow language.

Strengths

First, it is an extension of a popular language which could bridge the gap between SPL model checking and industry. Then, as a high level language, fStateflow allows to verify model of complex systems using constructs such as hierarchy or parallelism. Finally, integration of variability in fStateflow model by labelling transitions with feature expression is very easy.

Weaknesses

In spite of its popularity in the industry, fStateflow requires some expertise to be fully understood which could discourage its use. Another weakness is the possibility to have an extremely long execution time. This can be caused by a too large set of variables that implies a larger state space. In addition, variability integration in fStateflow model may alter its execution in the Matlab Environment given that they are declared as undefined inputs. Moreover, this annotative style of variability specification could not scale with the number of features. A too large set of feature could make the model too complex for engineers.

6.3 Future Work

In this work, we integrated only a small subset of the functionalities of the Stateflow environment. Such concepts as events or transition tables could be useful and interesting. We also described the weakness of an annotative specification of the variability, therefore, we could consider a compositional method. This method consists in defining a basic model and, for each feature, a module that describes its effects on the basic model. Finally, some optimizations are always possible in order to decrease execution time.

Bibliography

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* The MIT Press, 2008.

[CCH+12] Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Model checking software product lines with SNIP. *International Journal on Software Tools for Technology Transfer (STTT), Springer-Verlag*, 14(5):589–612, 2012. DOI 10.1007/s10009-012-0234-1.

[CCS+12] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. Featured transition systems: Foundations for verifying variability-intensive systems and their application to ltl model checking (to appear). *IEEE Trans Software Eng (TSE)*, 2012.

[CCS+13] Andreas Classen, Maxime Cordy, Pierre-Yves Schobbens, Patrick Heymans, Axel Legay, and Jean-François Raskin. Fts – featured transition systems: Provelines. http://www.info.fundp.ac.be/fts/provelines/, 2013.

[CHS+10] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-François Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *32nd International Conference on Software Engineering, ICSE 2010, May 2-8, 2010, Cape Town, South Africa, Proceedings*, pages 335–344. ACM, 2010. Acceptance rate: 13.7

[CHSL11] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In *33rd International Conference on Software Engineering, ICSE 2011, May*

21-28, 2011, Waikiki, Honolulu, Hawaii, Proceedings, pages 321–330. ACM, 2011. Acceptance rate: 14

[CLA08] Edmund CLARKE. *25 Years of Model Checking*, chapter The Birth of Model Checking. In Grumberg and Veith [GV08], 2008.

[Cla10] Andreas Classen. CTL model checking for software product lines in NuSMV. Technical Report P-CS-TR SPLMC-00000002, PReCISE Research Center, University of Namur, 2010.

[CN01] Paul C. Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley, August 2001.

[GV08] Orna Grumberg and Helmut Veith, editors. *25 Years of Model Checking*. Springer Berlin Heidelberg, 2008.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[IEE08] IEEE. Standard for software and system test documentation. 2008.

[Mat04] MathWorks. Stateflow - model and simulate decision logic using state machines and flow charts. http://www.mathworks.nl/products/stateflow/, 2004.

[SHT06] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature diagrams: A survey and a formal semantics. In *RE*, pages 136–145. IEEE Computer Society, 2006.

[SP00] A. Sahbani and J.C Pascal. Simulation of hybrid systems using stateflow. In *In 14th European Simulation Multiconference (ESM'2000)*, pages 271–275, Ghent, Belgique, 2000.

[Tri09] Jean-Christophe Trigaux. Smals, techno 35 : Les lignes de produits logiciels, 2009.

Appendix A

Models

The following figures depict all the behavioural model defined in Chapter 4. Gray states are defined further in the document.

Figure A.1: Top-level state

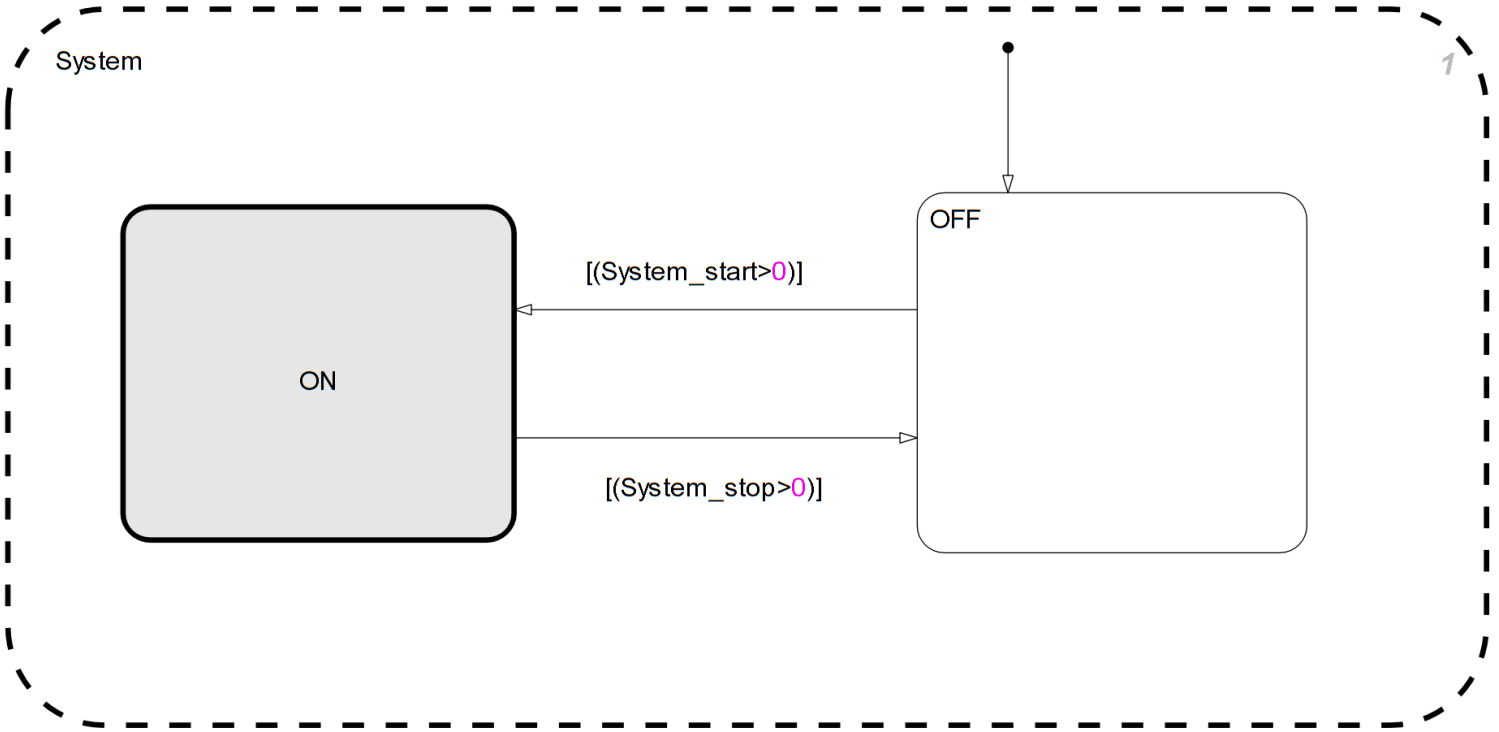


Figure A.2: System

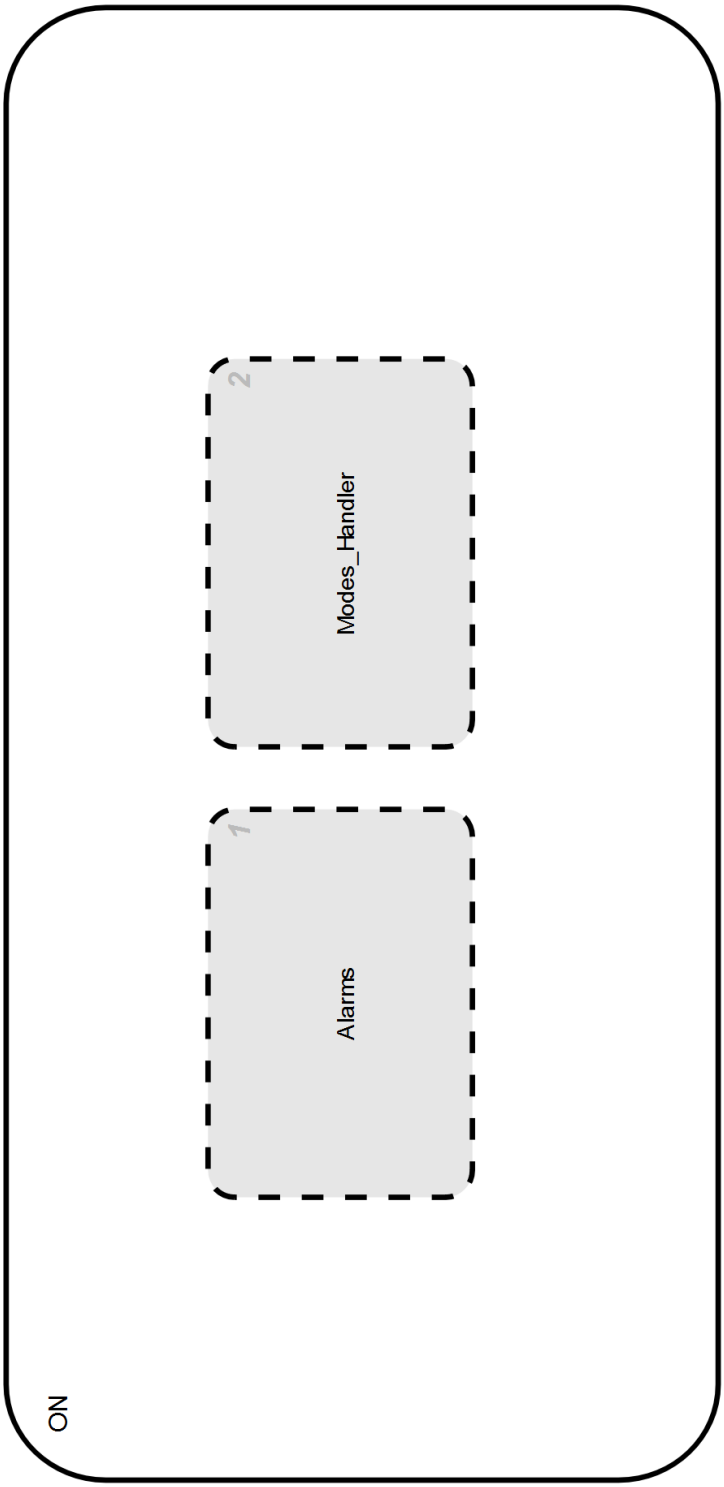


Figure A.3: System.ON

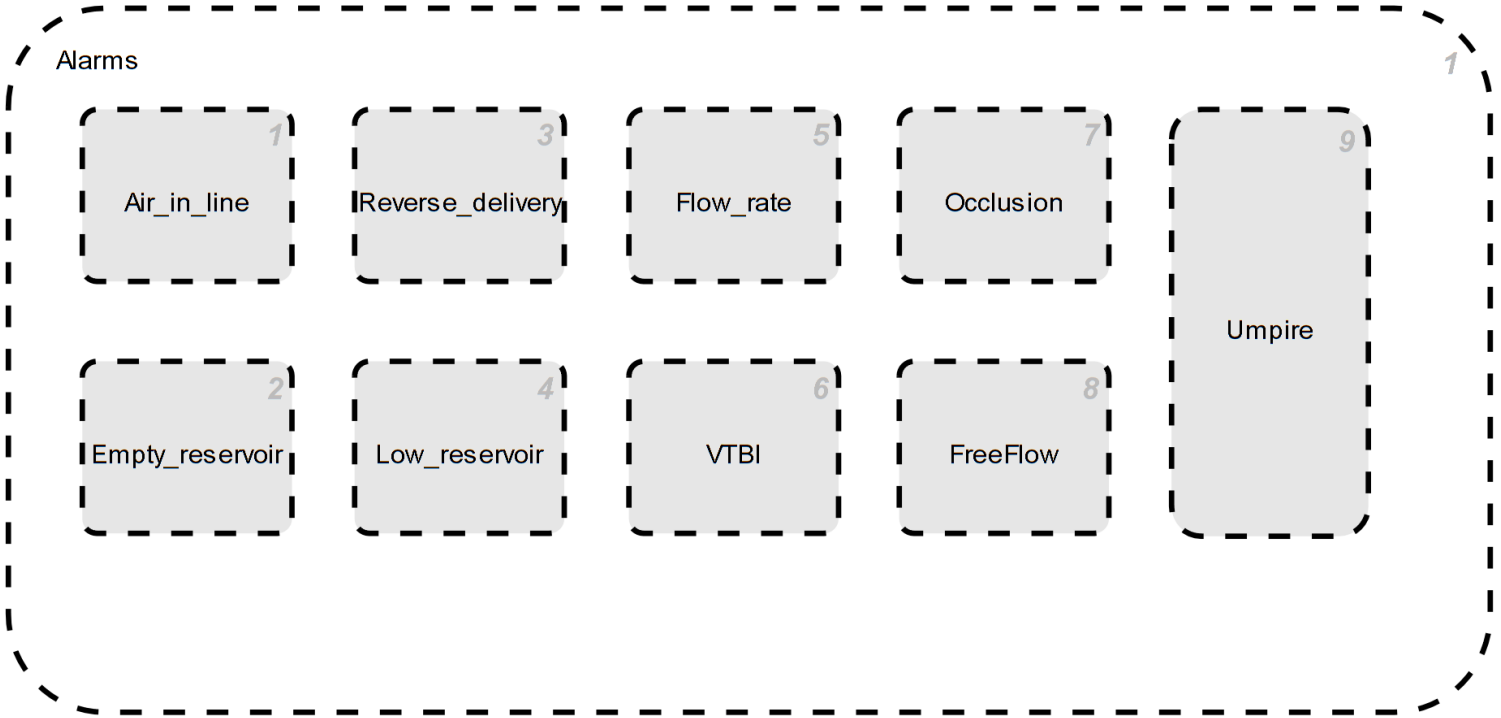


Figure A.4: System.ON.Alarms

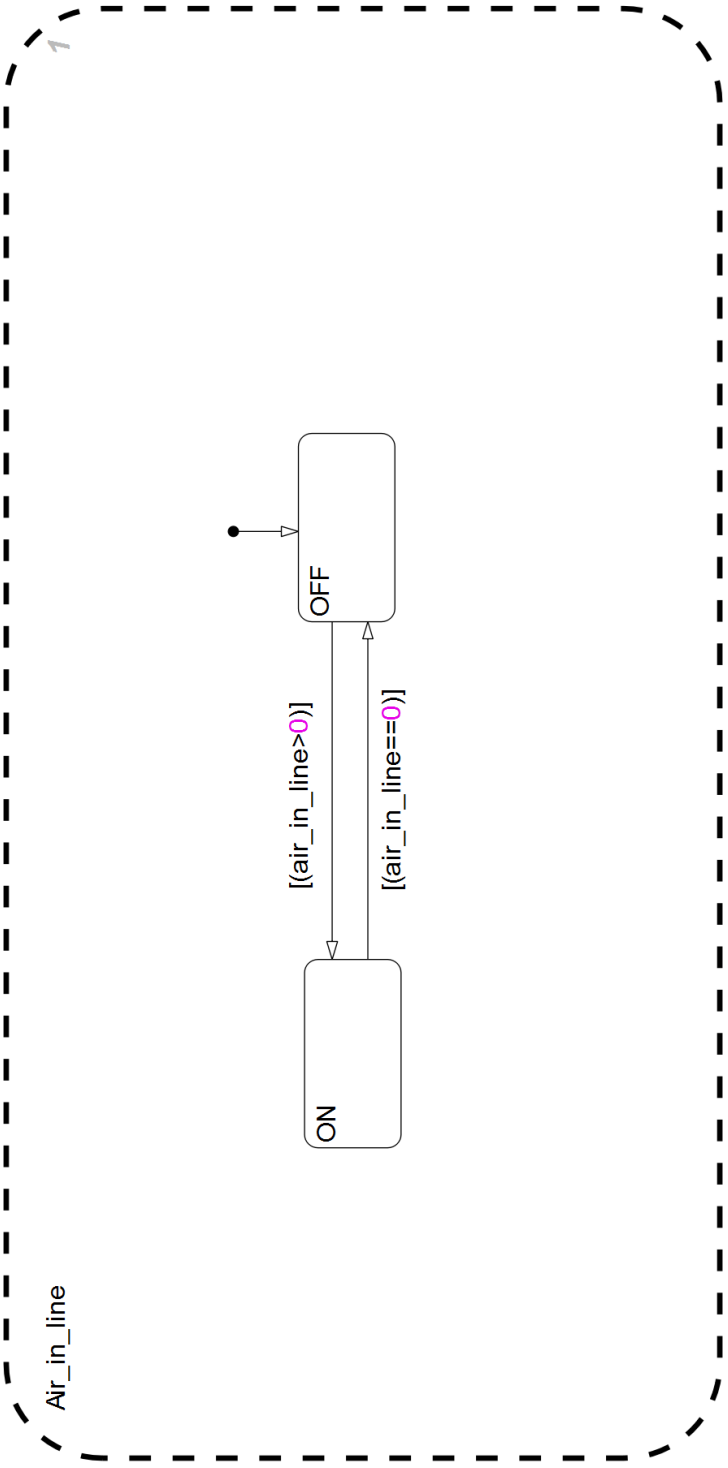


Figure A.5: System.ON.Alarms.AirinLine

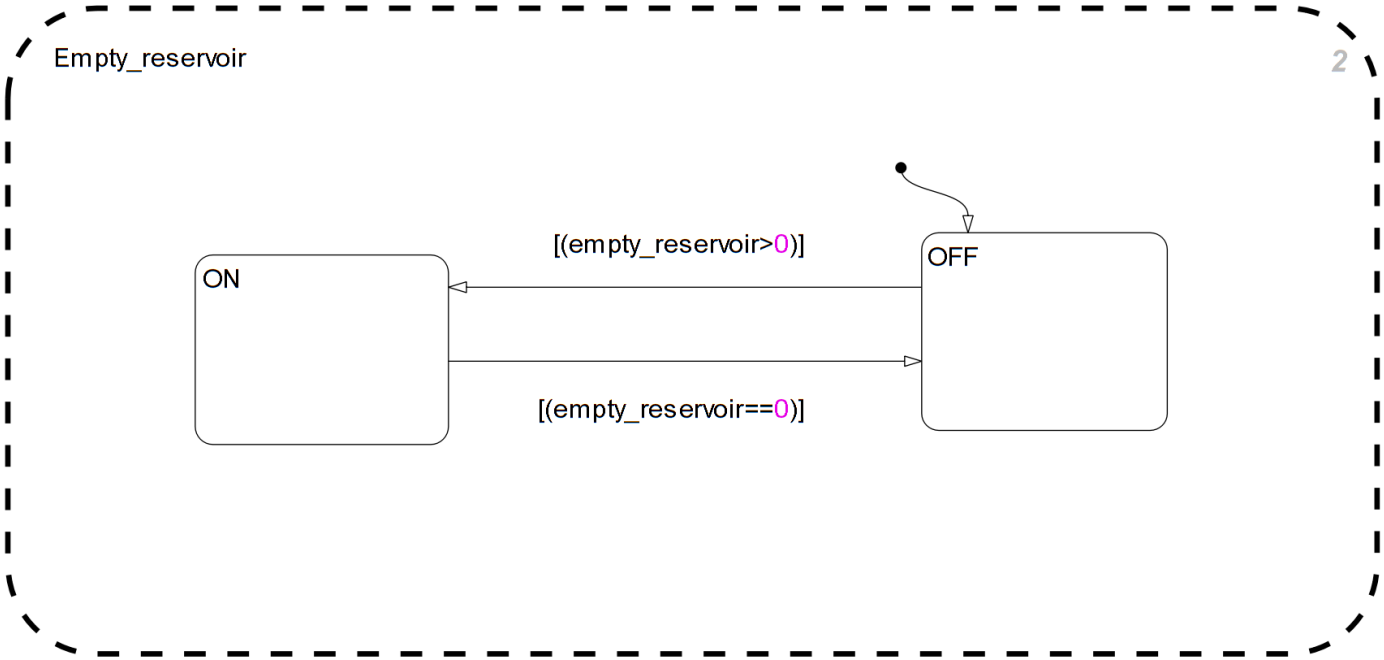


Figure A.6: System.ON.Alarms.Emptyreservoir

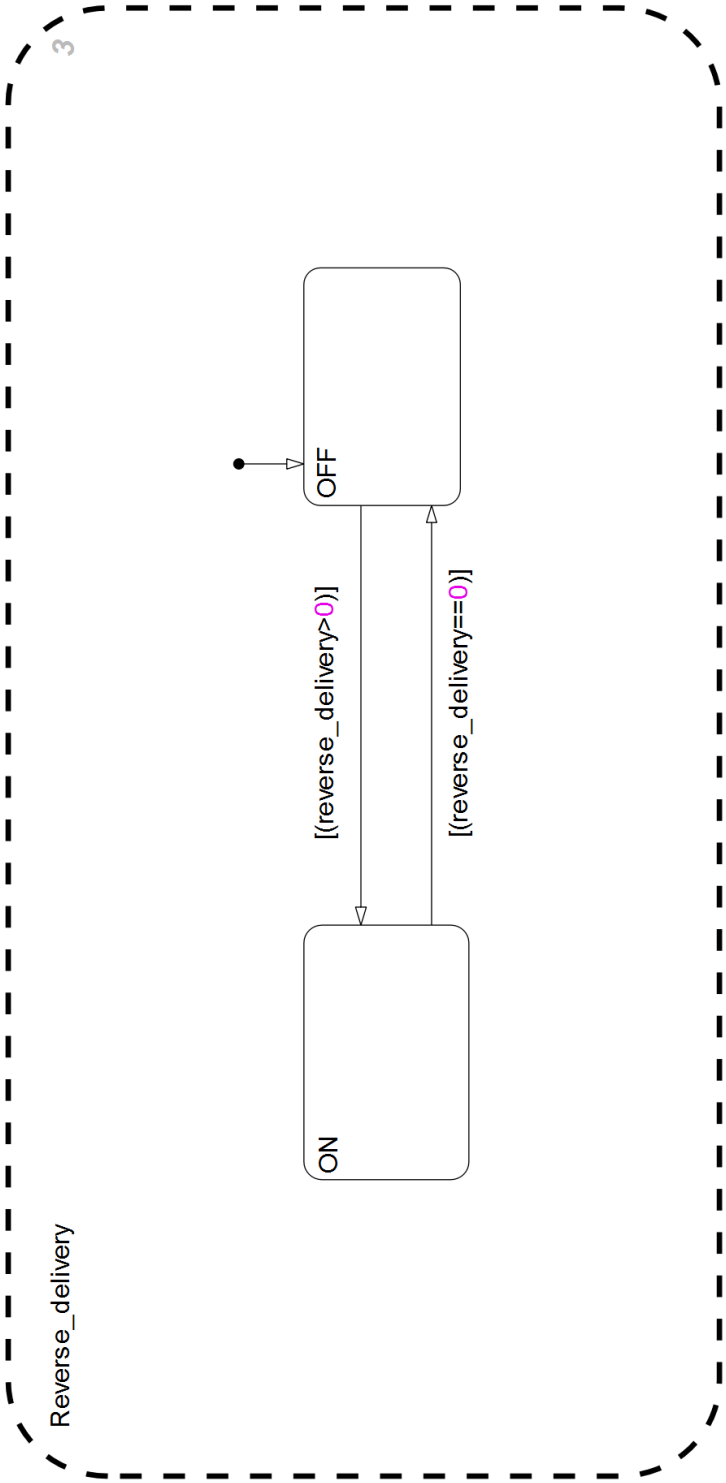


Figure A.7: System.ON.Alarms.Reversedelivery

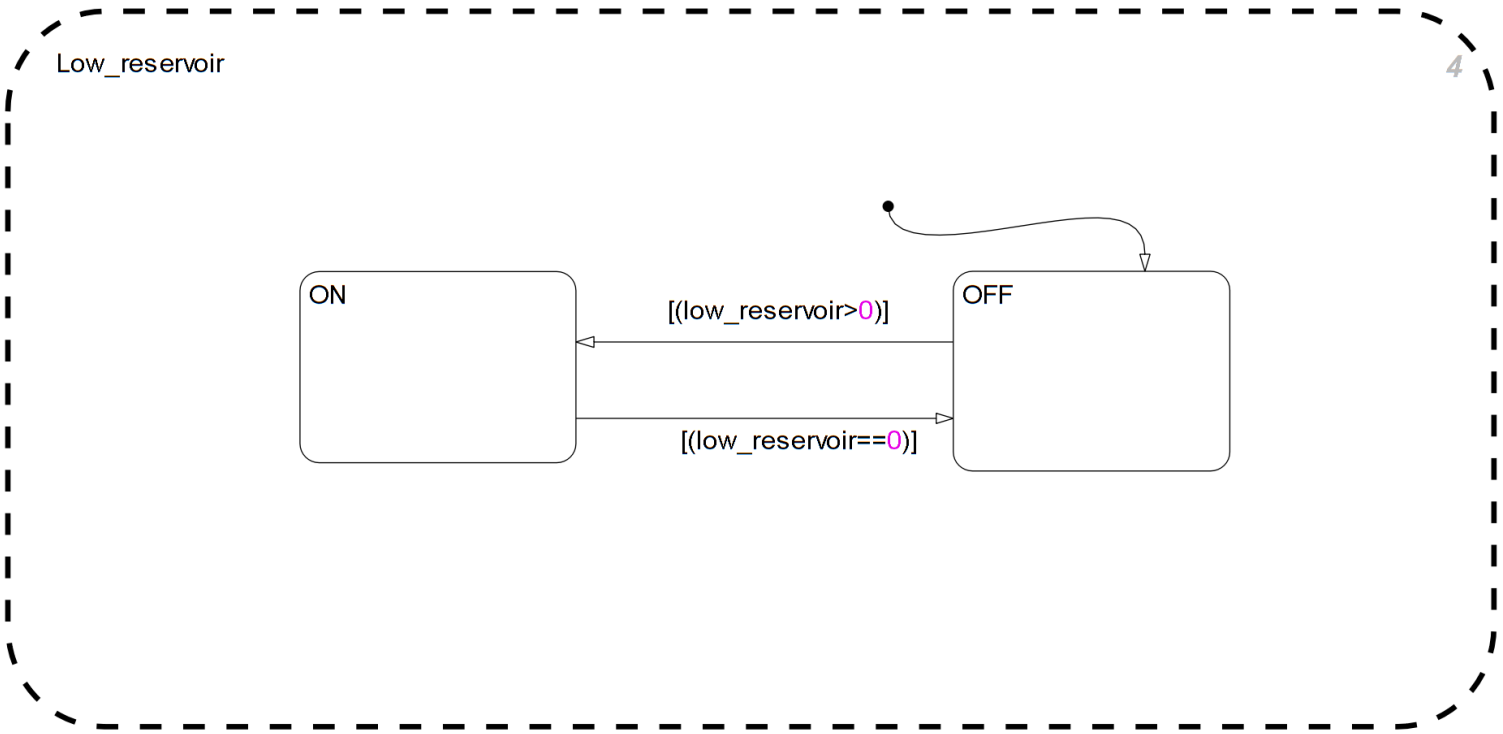


Figure A.8: System.ON.Alarms.Lowreservoir

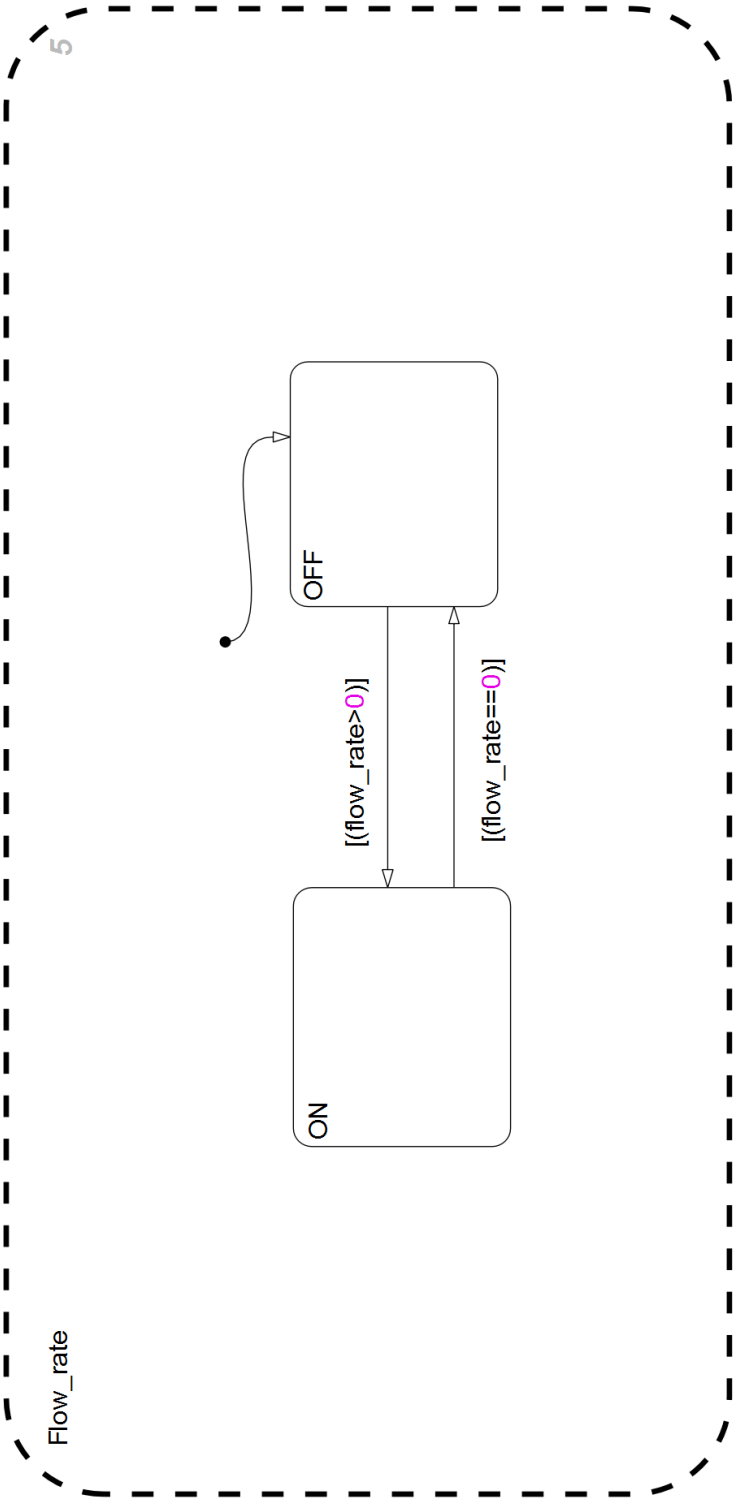


Figure A.9: System.ON.Alarms.Flowrate

Figure A.10: System.ON.Alarms.VTBI

Figure A.11: System.ON.Alarms.Occlusion

Figure A.12: System.ON.Alarms.FreeFlow

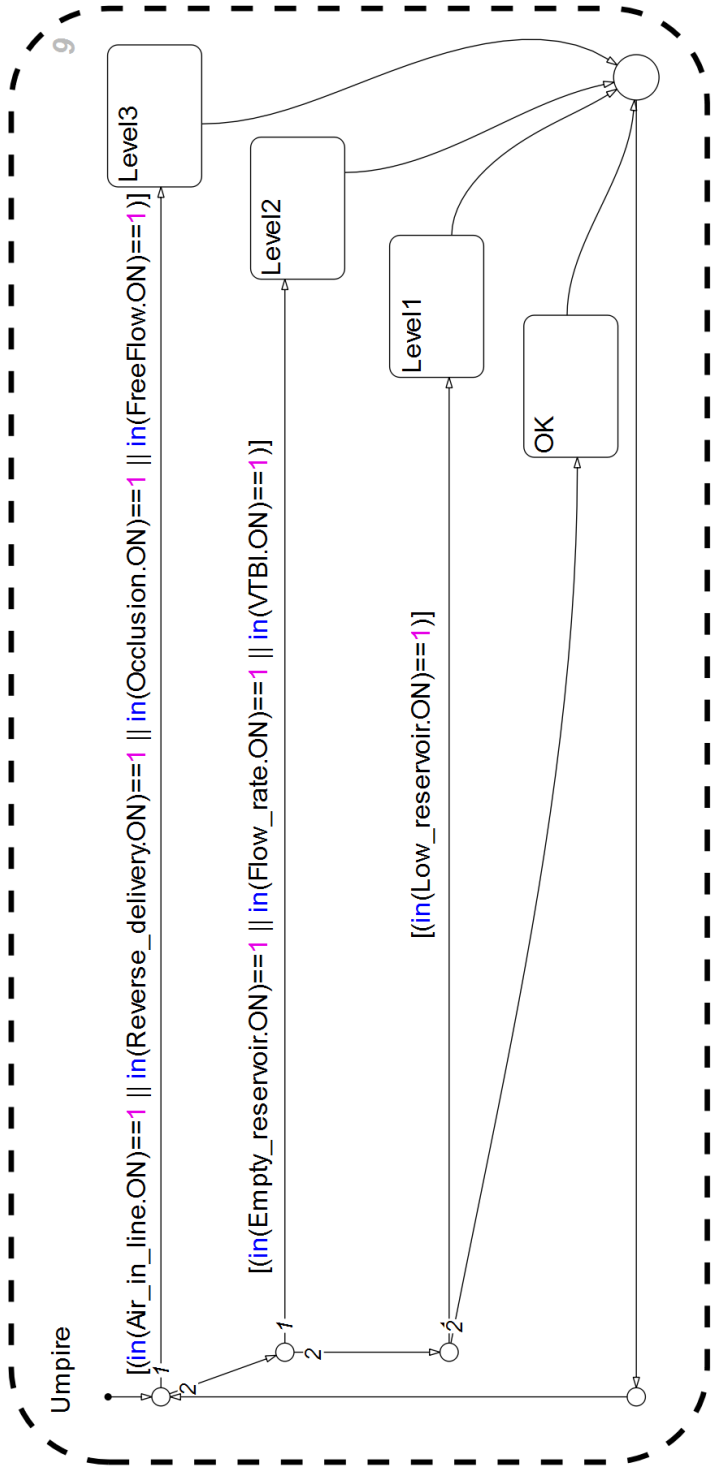


Figure A.13: System.ON.Alarms.Umpire

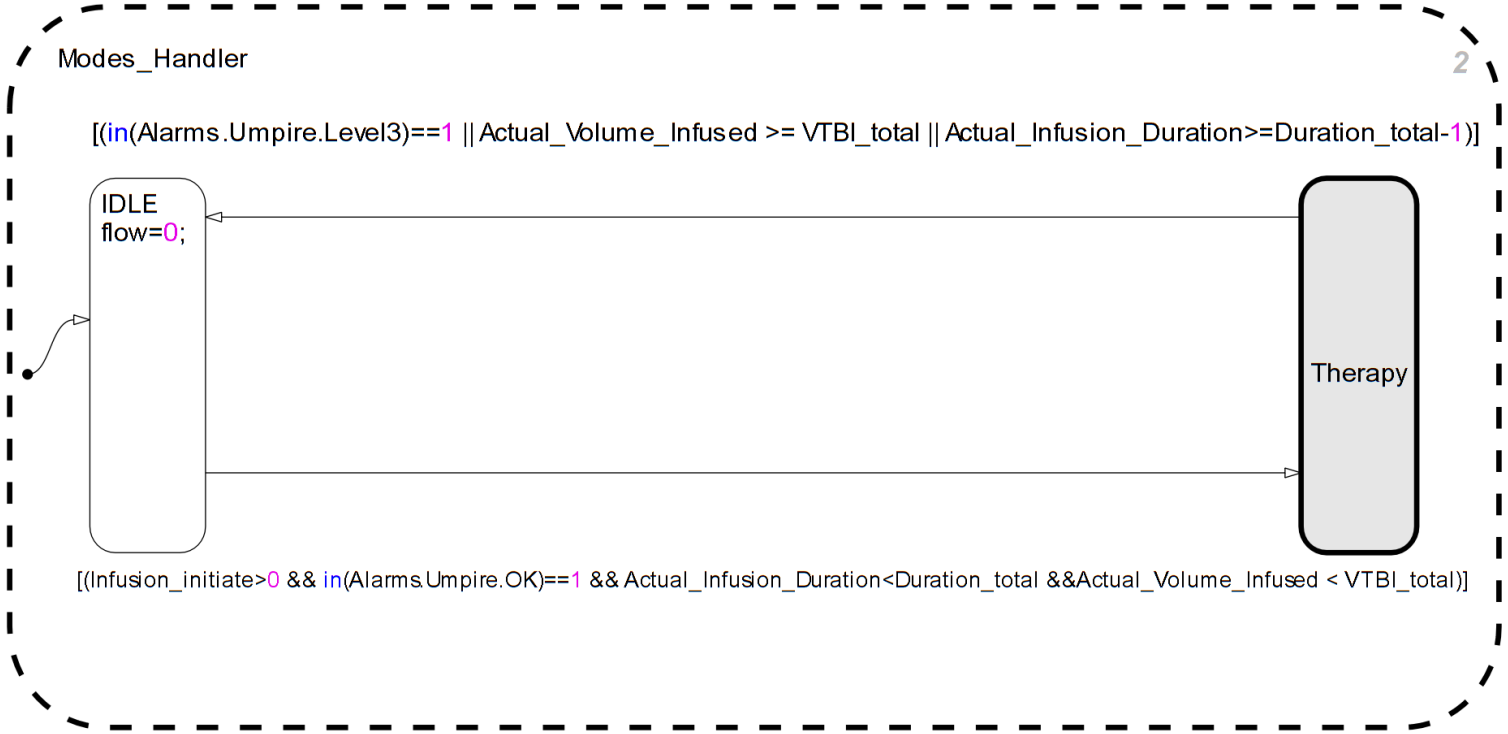


Figure A.14: System.ON.ModesHandler

Figure A.15: System.ON.ModesHandler.Therapy

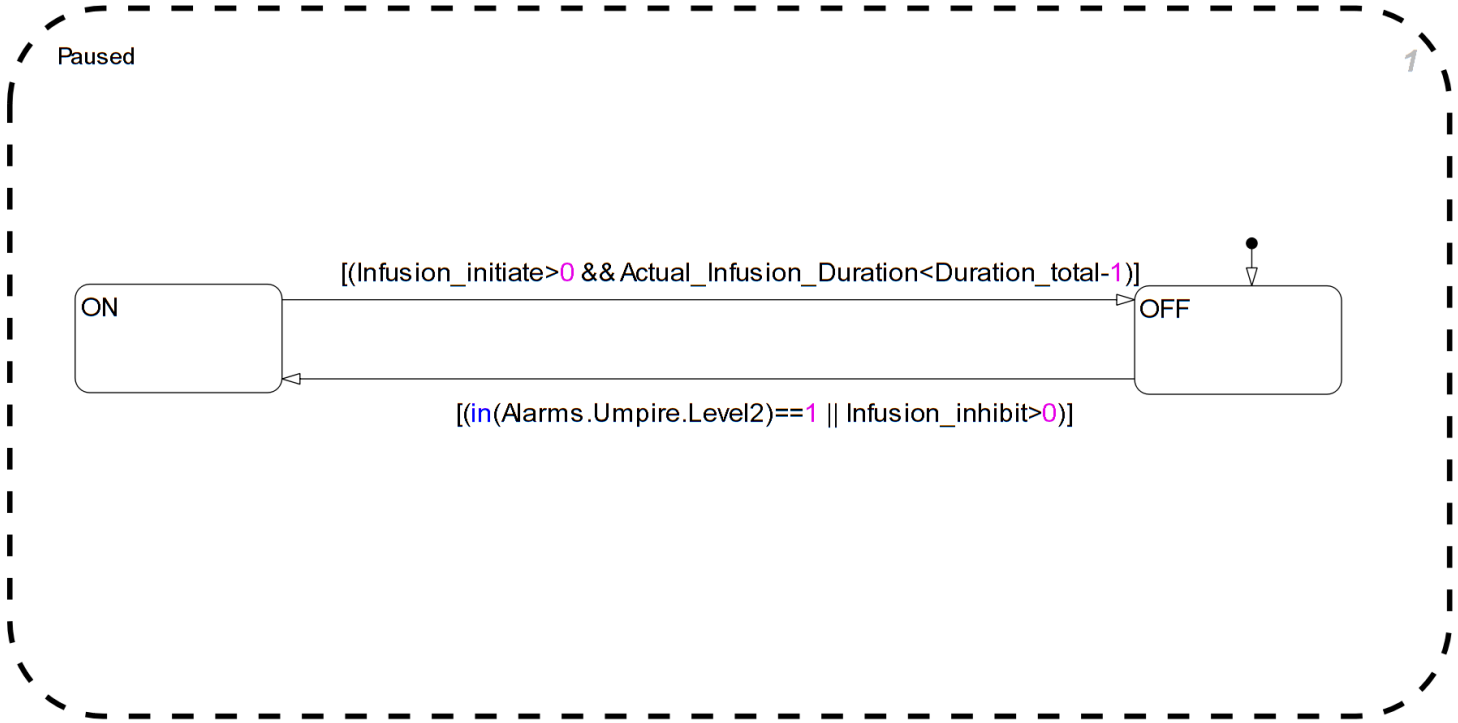


Figure A.16: System.ON.ModesHandler.Therapy.Paused

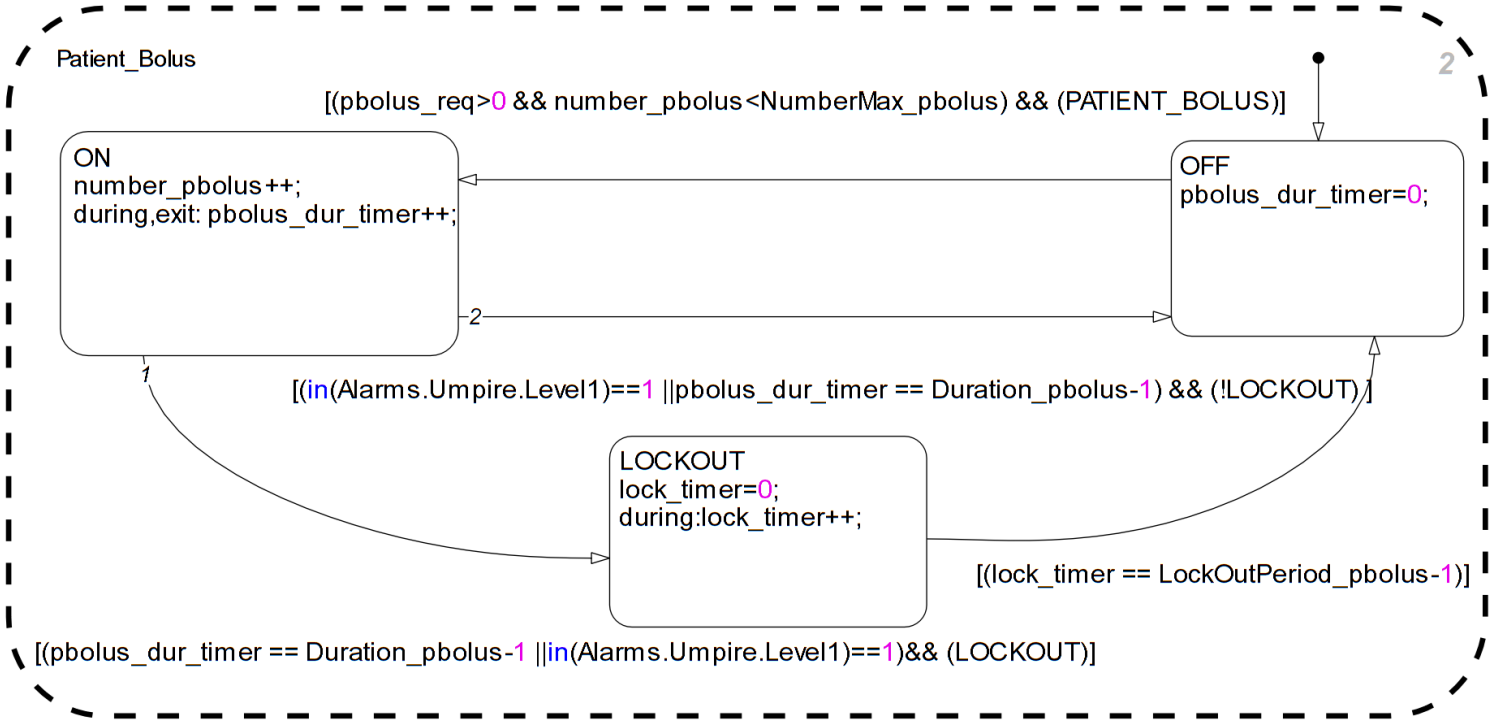


Figure A.17: System.ON.ModesHandler.Therapy.PatientBolus

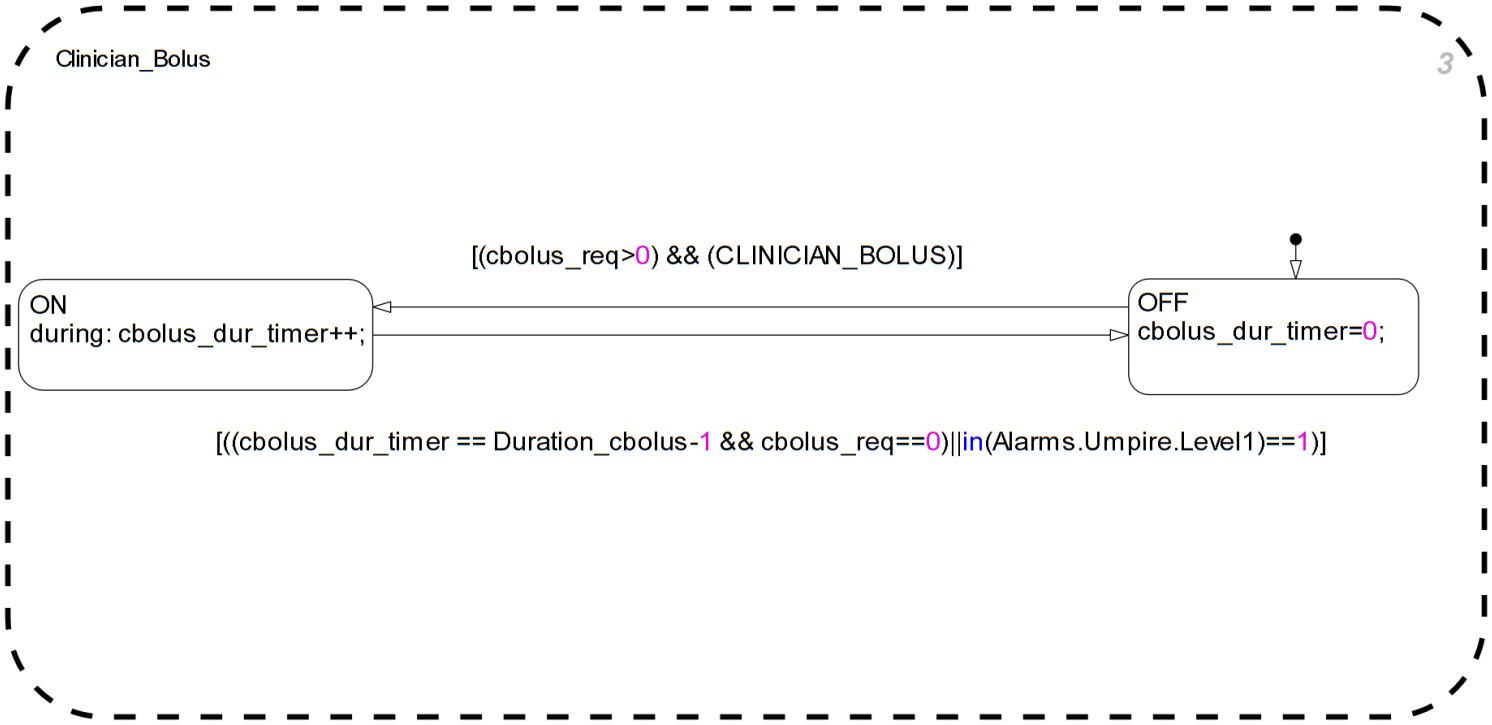


Figure A.18: System.ON.ModesHandler.Therapy.ClinicianBolus

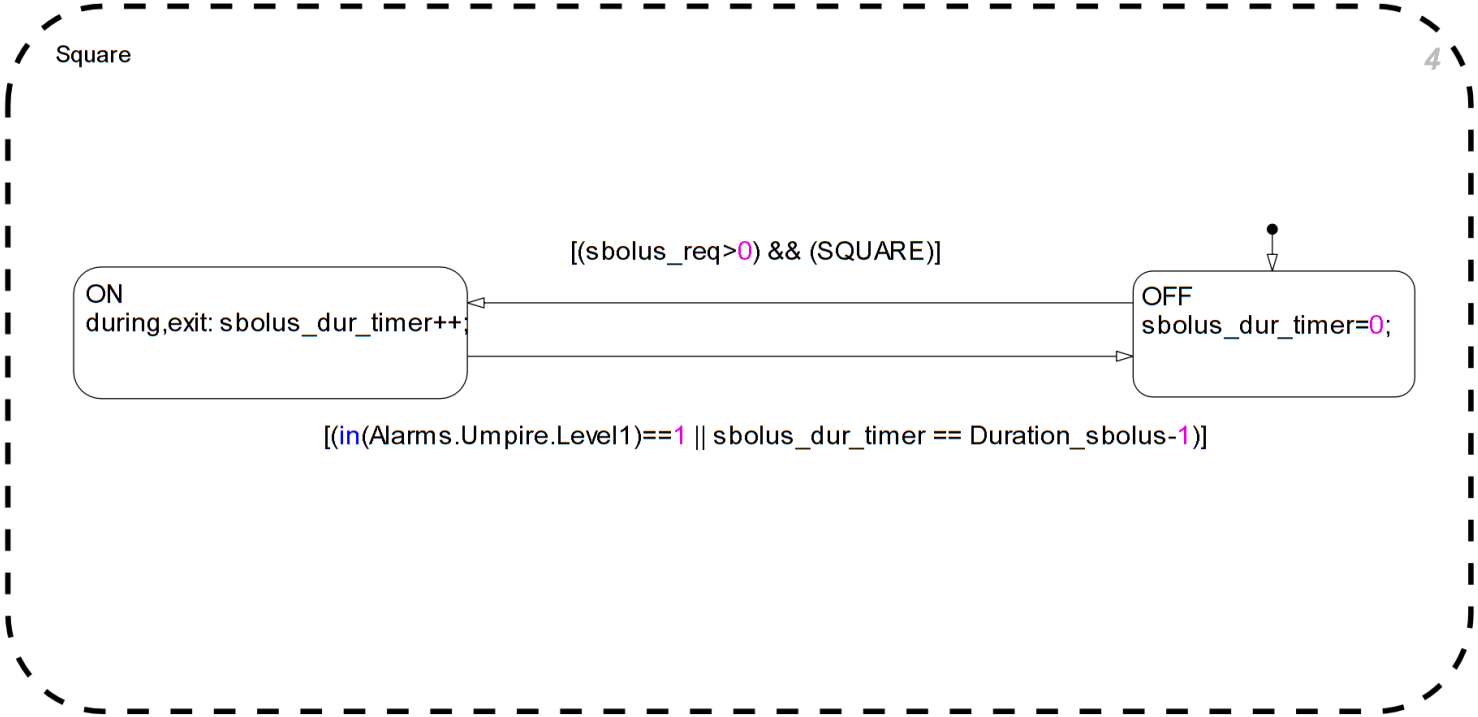


Figure A.19: System.ON.ModesHandler.Therapy.Square

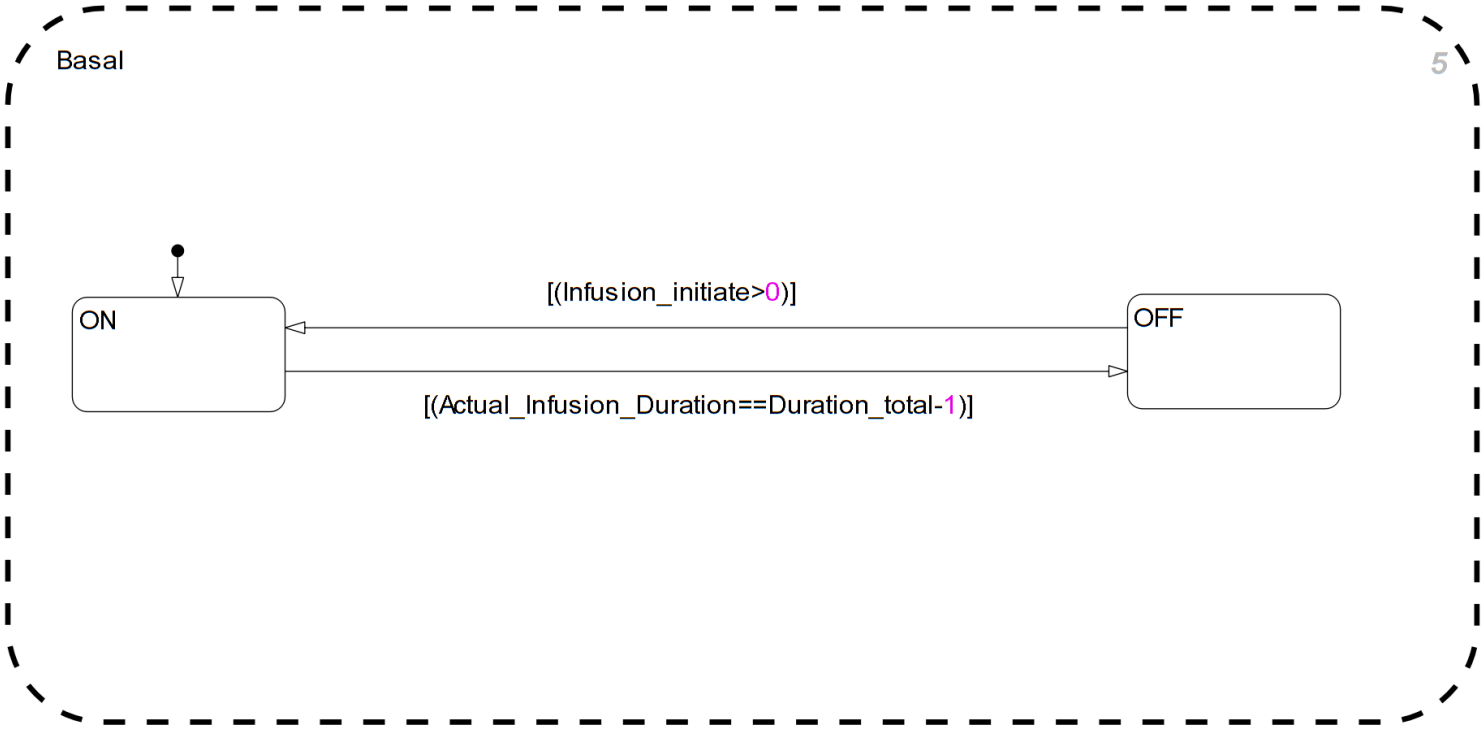


Figure A.20: System.ON.ModesHandler.Therapy.Basal

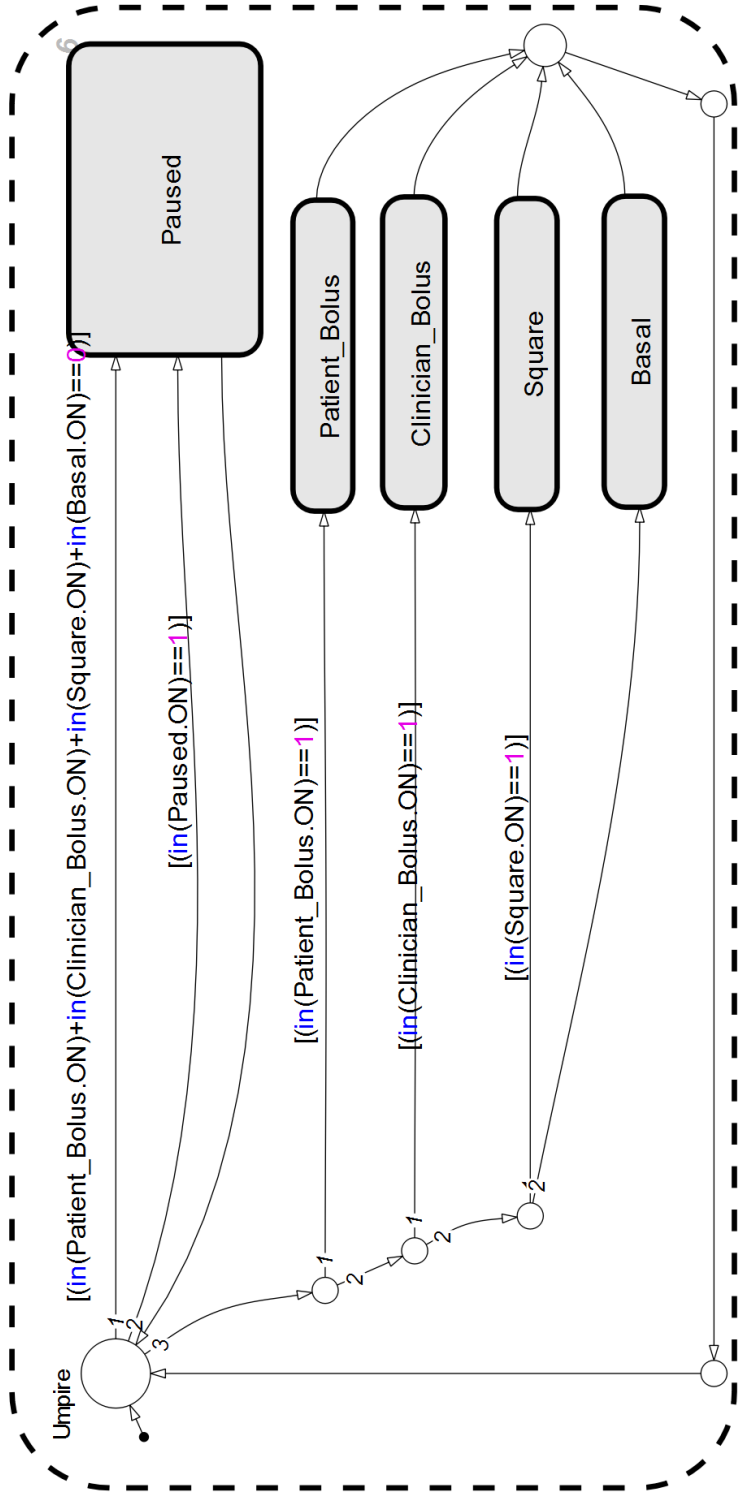


Figure A.21: System.ON.ModesHandler.Therapy.Umpire

Figure A.22: System.ON.ModesHandler.Therapy.Umpire.Paused

Figure A.23: System.ON.ModesHandler.Therapy.Umpire.PatientBolus

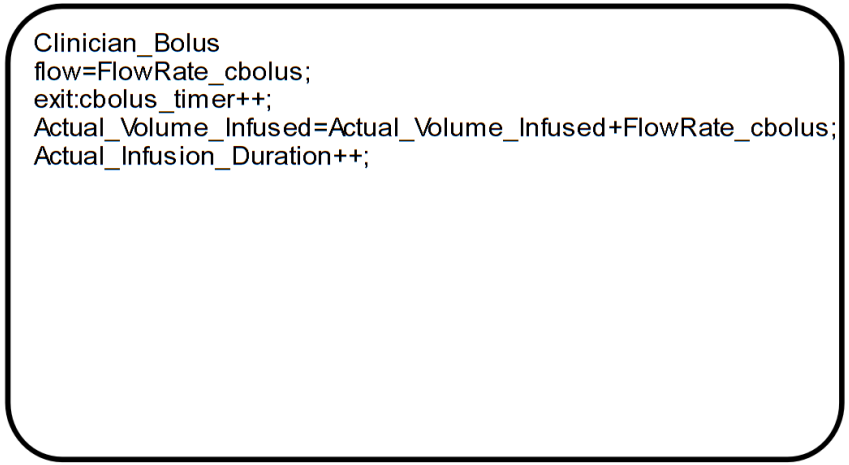


Figure A.24: System.ON.ModesHandler.Therapy.Umpire.ClinicianBolus

Figure A.25: System.ON.ModesHandler.Therapy.Umpire.Square

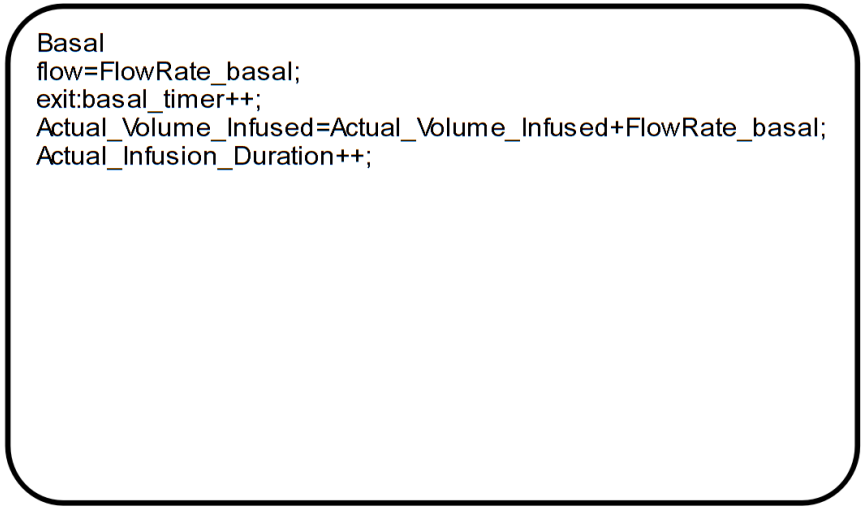


Figure A.26: System.ON.ModesHandler.Therapy.Umpire.Basal