Federal University of Campina Grande
Informatics and Electrical Engineering Center

Towards a Test Generation Approach for Compositional Real-Time Systems

Adriana Carla Damasceno

Thesis submitted to Post-Graduation Coordination of Computer Science of Federal University of Campina Grande in partial fulfillment of the requirements for the degree of Doctor of Computer Science.

Research area: Computer Science
Research Interest: Software Engineering

Patricia Duarte de Lima Machado
(Advisor)
Wilkerson de Lucena Andrade
(Co-advisor)

Campina Grande, Paraíba, Brasil
©Adriana Carla Damasceno, March 6, 2015

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

D155t Damasceno, Adriana Carla.
Towards a test generation approach for compositional real-time systems. / Adriana Carla Damasceno. – Campina Grande, 2015.
129f. : il. color.

Tese (Doutorado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2015.

"Orientação: Prof.ª Dr.ª Patrícia Duarte de Lima Machado, Prof. Dr. Wilkerson de Lucena Andrade".
Referências.

1. Real-time systems. 2. Compositional testing. 3. Model-based testing. 4. Tioco. 5. Integration testing. I. Machado, Patrícia Duarte de Lima. II. Andrade, Wilkerson de Lucena. III. Título.

CDU 004.451(043)

"TOWARDS A TEST GENERATION APPROACH FOR COMPOSITIONAL REAL-TIME SYSTEMS"

ADRIANA CARLA DAMASCENO

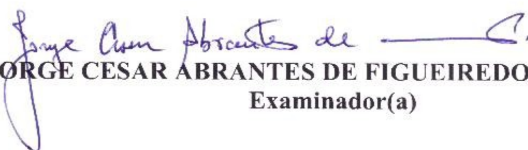
TESE APROVADA EM 06/03/2015

PATRICIA DUARTE DE LIMA MACHADO, Ph.D, UFCG
Orientador(a)

WILKERSON DE LUCENA ANDRADE, D.Sc, UFCG
Orientador(a)

ALEXANDRE CABRAL MOTA,
Examinador(a)

MARIA DE FÁTIMA MATTIELLO-FRANCISCO, Dra., INPE
Examinador(a)



JORGE CESAR ABRANTES DE FIGUEIREDO, D.Sc, UFCG
Examinador(a)

ADALBERTO CAJUEIRO DE FARIAS, Dr., UFCG
Examinador(a)

CAMPINA GRANDE - PB

// Abstract

We can find many examples of Real-time Systems (RTS) in critical applications such as patient monitoring, air traffic control and others. A failure in this kind of system can be catastrophic. For example, it can harm human lives or increase project budgets. Hence, the testing of real-time systems must be accurate. Models are used to perform this task, since they contain information about how the system behaves and when actions may happen. Due to the complexity of the available systems, most RTS are composed of subsystems that interact as part of a bigger system. These subsystems are combined through operators to model their specification behavior. However, works on the testing of compositional models for RTS are practically nonexistent. Among the available approaches to perform testing for non-compositional RTS models, the tioco conformance testing theory focuses on generating test cases based on implementation and specification models. Moreover, a conformance relation defines whether success in testing means conformance between an implementation and a specification. To express specifications and to represent implementations under test, we use Timed Input Output Symbolic Transitions Systems (TIOSTS). These models store symbolic data and clock variables, avoiding the state space and region explosion problems. Regarding the testing of compositional models, some questions may arise: If two subsystem implementations are tioco conformant to their specifications, is it correct to assume that the composition of the implementations is also tioco conformant to the composition of their specifications? In this case, how can operators be defined to work with TIOSTS and tioco? To answer these questions, this thesis proposes the sequential, interruption and parallel operators for the TIOSTS model. For each operator, we study how the tioco conformance relation behaves with respect to subsystems and the composed system. We present results towards properties of compositional operators when the subsystems are composed, as well as implementing them. Besides, we show three examples where each operator can be used and illustrate the applicability of our approach in two exploratory studies. The first models components of a aircraft specification and the second presents application level interruptions in an Android system.

Unshakable faith is only that which can meet reason face to face in every Human epoch.

—ALLAN KARDEC, *The gospel according to the Spiritism*

Acknowledgements

I want to thank God for letting me go beyond each difficulty to achieve this doctoral degree. Also, I express deep gratitude for my family, especially Damasceno, Gloria and Juan, for being extremely supportive, patient and kind; and Formiga e Maria Celia, for guiding me in the academic area since my childhood. I acknowledge the effort my advisors made in guiding me during this thesis, especially Patricia. Finally, I am grateful for all the friends who helped and supported me.

Contents

1 Introduction **1**

1.1 Problem Statement 3

1.2 Research Goals 5

1.3 Contributions 6

1.4 Methodology 7

1.5 Outline of the Document 8

1.6 Publications 8

2 Background **10**

2.1 Parallel Computing 10

2.2 Real-time Systems 11

2.3 Software Testing 12

2.3.1 Model-Based Testing 15

2.3.2 Conformance Testing 16

2.4 Concluding Remarks 30

3 Test Case Generation from Compositional Models **31**

3.1 Sequential Composition 32

3.2 Interruption Composition 38

3.3 Parallel Composition 48

3.4 Test Case Generation Process 53

3.5 Concluding Remarks 57

4 Algorithms **58**

4.1 Sequential Composition Algorithm 58

4.2 Interruption Composition Algorithm 62
4.3 Parallel Composition Algorithm 67
4.4 Concluding Remarks 70

5 Exploratory Studies 71
5.1 Avionics System 71
5.2 Cell Phone System 78
5.3 Concluding Remarks 87

6 Related Works 88
6.1 Model-based Interruption Testing 88
6.2 Model-based Compositional Testing 90
6.2.1 Bijl *et al.* 90
6.2.2 Daca *et al.* 91
6.2.3 Sampaio *et al.* 92
6.2.4 Briones 93
6.2.5 Aiguier *et al.* 93
6.2.6 Faivre *et al.* 94
6.2.7 Olderog and Swaminathan 95
6.2.8 Krichen and Tripakis 96
6.2.9 Bannour *et al.* 97
6.3 Concluding Remarks 98

7 Concluding Remarks 100
7.1 Main Results 100
7.2 Future Works 102

A Proofs 111
A.1 Theorem 2 111
A.2 Theorem 3 114
A.3 Theorem 4 116

List of Figures

1.1 $\mathcal{S}_1$ specification . . . 3

1.2 $\mathcal{S}_2$ specification . . . 3

1.3 Model $\mathcal{I}_1$ tioco $\mathcal{S}_1$ . . . 4

1.4 Model $\mathcal{I}_4$ $\neg tioco$ $\mathcal{S}_1$ . . . 4

1.5 Composition for $\mathcal{S}_1 \parallel \mathcal{S}_2$ . . . 5

1.6 Composition for $\mathcal{I}_1 \parallel \mathcal{I}_2$ . . . 5

2.1 General testing process . . . 14

2.2 Model-based testing approach (adapted from [Kic14]) . . . 16

2.3 TAIO for the *Mouse* subsystem . . . 18

2.4 TAIO for the *Screen* subsystem . . . 18

2.5 TAIO *Screen* conformant . . . 19

2.6 TAIO *Screen* not conformant . . . 19

2.7 Parallel composition for the *Mouse* and *Screen* specifications using the TAIO model . . . 20

2.8 Input-completion for the *Screen* TAIO specification . . . 22

2.9 *Mouse* subsystem . . . 25

2.10 *Screen* subsystem . . . 25

2.11 Model in conformance with *Screen* . . . 26

2.12 Model not in conformance with *Screen* . . . 26

2.13 Test case generation process . . . 29

3.1 Toy example for the sequential composition . . . 33

3.2 *Choose* subsystem . . . 34

3.3 *Pay* subsystem . . . 34

3.4 The sequential composition $Choose;_{send}$ Pay . . . 35
3.5 *Mouse* subsystem . . . 40
3.6 *Reset* subsystem . . . 40
3.7 Interruption composition for the *Mouse* and *Reset* subsystems . . . 40
3.8 Example of weak interruption composition . . . 42
3.9 Step 1 of the *sync* operator . . . 43
3.10 Step 2 of the *sync* operator . . . 44
3.11 Steps for the weak interruption composition . . . 46
3.12 Toy example for the parallel composition . . . 49
3.13 Parallel composition for *Mouse* and *Screen* specifications . . . 50
3.14 Input-complete *Mouse* spec . . . 51
3.15 Input-complete *Screen* spec . . . 51
3.16 The integration test case generation process . . . 53
3.17 Test purpose for the $Choose;_{send}$ Pay system . . . 54
3.18 Test cases for the $Choose;_{send}$ Pay system . . . 55
3.19 Test purpose for *Mouse* $\|$ *Screen* system . . . 55
3.20 Test cases for the *Mouse* $\|$ *Screen* system . . . 56
3.21 Test purpose for *Mouse Interrupted* and *Reset* subsystems . . . 56
3.22 Test cases for the *Mouse Interrupted* and *Reset* subsystems . . . 57

4.1 Toy example for the sequential composition . . . 59
4.2 Example of weak interruption composition . . . 62
4.3 Toy example for the parallel composition . . . 67

5.1 Tracking system of the generic avionics specification . . . 72
5.2 *Radar* specification . . . 73
5.3 *Target Designation* specification . . . 74
5.4 *Target Tracking* specification . . . 76
5.5 Sequential composition for the *Tracking* subsystem . . . 77
5.6 Test purpose for the *Target* system . . . 78
5.7 Test purpose for *Target* system $\|$ *Radar* . . . 78
5.8 *Cell Phone* system . . . 80

5.9 *Contacts* subsystem 82
5.10 *Message* subsystem 83
5.11 *Receive Call* subsystem 84
5.12 Test purpose (a) 84
5.13 Test purpose (b) 84
5.14 Test purpose (c) 85
5.15 Test purpose (d) 85
5.16 Test case 0 from test purpose (a) 86

A.1 Toy example for the sequential composition 112
A.2 Example of weak interruption composition 114
A.3 Toy example for the parallel composition 118

List of Tables

5.1 Test purposes and generation time of test cases 78
5.2 Test purposes and generation time of test cases 87

6.1 Summary of works 99

Chapter 1

Introduction

Real-time systems (RTS) are currently found in a number of embedded devices for monitoring and controlling activities such as patient monitoring, operation of aircrafts, business transactions, and so on. These systems have requirements that are constrained by time, that is, system tasks are performed successfully if a correct behavior can be observed within a time period [Lap09]. For instance, consider a robot movement controller that might act within $2s$ when an obstacle is detected in order to avoid a crash. Independently of the technology used, these systems are usually composed of a set of concurrent subsystems that may run on independent devices. Whether the system is critical or not, testing such systems is a complex task particularly at integration level. There is usually a number of interactions required as well as a number of assumptions of expected behavior that might be precisely defined in order to avoid false verdicts during testing execution.

To address these issues, a number of efforts have been carried out towards a conformance testing theory and practice for real-time systems [AM13; BJSK11; HLM+08]. Usually, the goal is to automatically generate valid test cases from an abstract model according to a testing criteria and test hypotheses in order to check conformance of a target implementation. This practice is known as Model-Based Testing (MBT) [UL07]. MBT has been extensively investigated as well as applied in industry. For instance, there are experience reports on the use of MBT for RTS by the SCARLETT [Sca14] and the DAIMLER/Verified Systems International GMBH [PHL+11] projects.

However, the state explosion problem [Val98] remains unsolved for a wide range

of models based on labeled transition systems. We can identify initiatives to solve this problem which are based on language specifications [DFM09] and symbolic models [Vea13]. Regarding conformance testing based on transition systems, symbolic models are a promising approach since both data and time can be abstracted in order to handle the complexity of RTS as well as to cope with the state space explosion problem [vSBS10; TR11]. Particularly, Andrade *et al.* [AM13] proposed a symbolic model, named Timed Input Output Symbolic Transition System (TIOSTS), devoted to real-time systems. The model can express both data and time symbolically with different types of deadlines. A test case generation process and a tool – SYMBOLRT – were also developed based on this model [AACM12]. However, the structure of the system under testing is not taken into account since the system is specified by a single model.

Conformance testing based on compositional models [BRT04; KT06] has addressed the problem of deciding the conformance of a system defined by a composition of specifications, when the corresponding subsystem implementations are in conformance to their specifications by testing them separately. For this, precise rules and constraints must be defined so that the composition of implementations preserves conformance to the composition of their corresponding specifications. Compositional testing can save integration testing effort because lessen system complexity to the level of subsystems [BRT04; DHKN14; SNMI14]. Moreover, test case generation from composite specifications can greatly contribute to the integration testing process by increasing reliability to test case definition and execution. Even though, there might be strong practical assumptions to be met on specifications and implementations when inferring compositional conformance by testing, some of them might be controlled by the test execution infrastructure.

The compositional conformance testing of real-time systems [KT06; BGAL13] can be achieved by two main strategies: i) infer conformance to the composition when subsystems are conformant or ii) generate tests from the composed specification to test the integration of the composed system. We follow and present both strategies along this work, studying difficulties and practical implications that may arise with them.

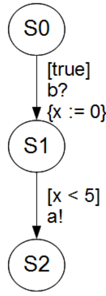


Figure 1.1: $\mathcal{S}_1$ specification

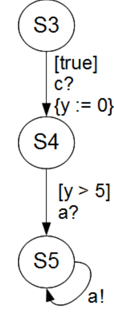


Figure 1.2: $\mathcal{S}_2$ specification

1.1 Problem Statement

Frequently, real-time systems requirements are modeled by Timed Automata with Inputs and Outputs (TAIO)[KT06]. Basically, this model is composed of a set of locations, a set of clocks that can be reset to zero, sets of input and output actions and a set of transitions where each element is a tuple comprised of the following elements: a guard over a clock, an action and an assignment that is related to the clock. In this section, we use the TAIO model to express the specification and implementation behaviors we want to check.

Figure 1.1 illustrates the TAIO for the $\mathcal{S}_1$ specification. From the S_0 location, the system communicates the *b?* input action, resets clock x and goes to location S_1. From the S_1 location, the system communicates the *a?* input action as long as $x < 5$ and goes to the S_2 location.

Figure 1.2 shows the $\mathcal{S}_2$ specification. Similarly to $\mathcal{S}_1$, it also has three locations. From the S_3 location, the system communicates the c? input action, resets the y clock and goes to the S_4 location. From location S_4, the system executes the *a!* output action if $y > 5$ and goes to the S_5 location. From the S_5 location, the system offers the *a!* output action and returns to the S_5 location.

We use the tioco (Timed Input-Output COnformance) to compare a specification $\mathcal{S}$ to an implementation model $\mathcal{I}$. In general terms, we say that $\mathcal{I}$ tioco $\mathcal{S}$ if, for every output action that $\mathcal{I}$ is capable of showing after a trace σ, there is a corresponding output action in $\mathcal{S}$. Hence, even if there is an input action present in $\mathcal{S}$ which is absent in $\mathcal{I}$, the tioco relation can be preserved.

Figure 1.3 shows the $\mathcal{I}_1$ implementation that is in tioco conformance to $\mathcal{S}_1$. If we

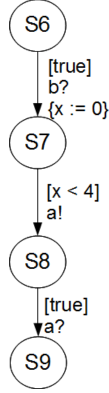


Figure 1.3: Model $\mathcal{I}_1$ tioco $\mathcal{S}_1$

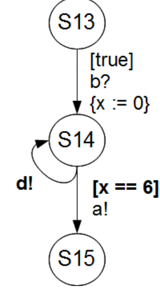


Figure 1.4: Model $\mathcal{I}_4$ $\neg$*tioco* $\mathcal{S}_1$

compare each trace from $\mathcal{S}_1$ to each trace from $\mathcal{I}_1$, we can notice that the given outputs are the same. Although the S7-S8 transition from $\mathcal{I}_1$ contains the $x < 4$ guard that differs from the $x < 5$ guard from the S1-S2 transition of $\mathcal{S}_1$, since the represented set from the $x < 4$ guard is contained in the represented set from the $x < 5$ guard, $\mathcal{I}_1$ tioco $\mathcal{S}_1$.

On the other hand, Figure 1.4 shows the $\mathcal{I}_4$ implementation which characterizes an example where the tioco conformance relation is not preserved. This happens because the S14-S14 transition allows the extra *d!* output action that is not present in the corresponding trace of the $\mathcal{S}_1$ specification. Furthermore, the S14-S15 transition contains the $x == 6$ guard that represents a set which is not contained in $x < 5$. In fact, these sets refer to different clock values, what forbids the preservation of the tioco conformance relation.

Unfortunately, the tioco conformance relation has some properties that stickles its usage when we test two isolated subsystems. Consider two specifications $\mathcal{S}_1$ and $\mathcal{S}_2$, their corresponding implementations $\mathcal{I}_1$ and $\mathcal{I}_2$ and a compositional operator **rel**. If $\mathcal{I}_1$ tioco $\mathcal{S}_1 \wedge \mathcal{I}_2$ tioco $\mathcal{S}_2$, we cannot always assume that $\mathcal{I}_1$ **rel** $\mathcal{I}_2$ tioco $\mathcal{S}_1$ **rel** $\mathcal{S}_2$. This happens because, although the tioco conformance relation allows the underspecification of inputs, it forbids the underspecification of outputs, leading to the generation of unpredictable outputs in the composed models.

For example, using the $\mathcal{S}_1$ and $\mathcal{S}_2$ specifications from Figures 1.1 and 1.2 and an ordinary parallel operator that allows subsystems to synchronize on common actions or interleave on distinct actions, we have the $\mathcal{S}_1 \parallel \mathcal{S}_2$ system in Figure 1.5. Considering

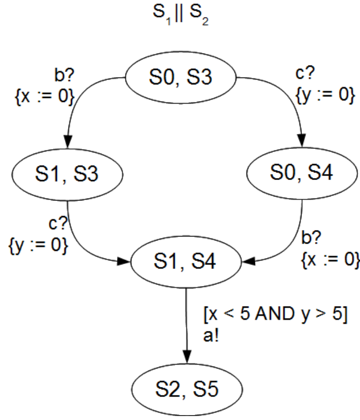


Figure 1.5: Composition for $\mathcal{S}_1 \parallel \mathcal{S}_2$

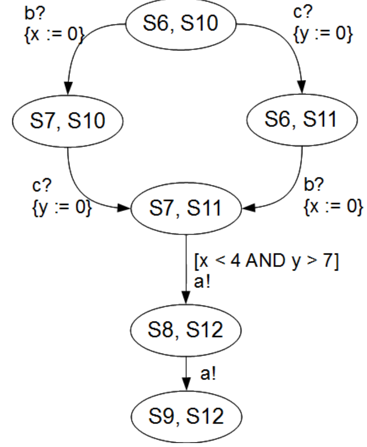


Figure 1.6: Composition for $\mathcal{I}_1 \parallel \mathcal{I}_2$

that implementation $\mathcal{I}_2$ is represented by the same model of specification $\mathcal{S}_2$, Figure 1.6 shows system $\mathcal{I}_1 \parallel \mathcal{I}_2$. It has the additional *a!* output action in the (S8,S12) - (S9, S12) transition which is generated during the synchronization process because of the additional *a?* input action of the S8-S9 transition contained in the $\mathcal{I}_1$ implementation.

There are many implications on the usage of the tioco conformance relation when testing compositional systems. Using the composed specification from Figure 1.5, we can observe that interleaved actions such as *b?* and *c?* must be tested and their interchangeable nature must be taken into account. In addition, the resulting composition and clocks resetting may lead the system that contain paths with unreachable conditions. Thus, if we take the two available paths to (S1, S4) location, the only one that makes the $x < 5$ *AND* $y > 5$ guard satisfied is by using the (S0, S4) location.

1.2 Research Goals

Facing the difficulties pointed out in Section 1.1, we define the following research questions that guides our work:

Research Question 1 *How symbolic models of real-time systems that abstract data and time can be composed?*

Research Question 2 *What are the main challenges to infer conformance of the composed system based on the conformance of subsystems?*

Research Question 3 *How can integration test cases be generated from composed models?*

These questions have been investigated in the scope of timed symbolic transition systems, more specifically, the TIOSTS model. We focus on the message-passing paradigm and investigate the sequential, interruption and parallel composition of subsystems with inputs and outputs that can be observed and modeled as independent components (Research Question 1). We examine constraints and applicability of these operators in accordance to the tioco conformance relation, guiding the test generation and execution results analysis (Research Question 2). Finally, from a practical point of view, integration testing is analyzed to allow test generation focused on subsystem interactions (Research Question 3).

To let our scope clear, we clarify that:

- We compose models intended to be used in a test case generation process;
- Test cases are derived from the compose model;
- We investigate theoretical implications of compositionality properties by using the tioco conformance relation;
- We focus on the generation of abstract test cases, excluding implementation issues of test cases and other architectural details;
- We take into consideration the composition of two subsystems at time;
- We do not take state and region explosion issues into account.

1.3 Contributions

To answer the research questions from Section 1.2, we propose a framework for composing real-time models. This is achieved by defining compositional operators that are suitable to the model and the conformance testing theory we intend to use. More specifically, we present the contributions:

- Define the parallel, sequential and interruption operators;

- Present and prove compositionality properties about conformance from subsystems to composed systems;
- Propose an integration testing strategy that uses these operators;
- Implement operators in a tool.

In general terms, our approach is composed by the following steps (Chapter 3): i) Compose subsystems; ii) Choose actions according to our test strategy; iii) Use a test generation strategy to generate test cases from the composed model . After that, we have test cases that cover interactions between subsystems. Moreover, we automate the composition of models and apply it two exploratory studies: i) a general avionics specification; and ii) a cell phone application. (Chapter 5).

We validade compositionality properties for each operator by using proofs that verify the preservation of the conformance relation when the subsystems are composed. Even so, we d not automate these compostionality results because of theoretical issues.

1.4 Methodology

We performed the following steps to develop our work:

- Execution of exploratory studies on the TIOSTS model and conformance theory described by [AM13]. Results showed that the definition of compositional operators improved practical usage of their work;
- Development of a systematic mapping to understand the main issues and difficulties regarding to compositional modeling of real-time systems. We analyzed 37 papers and compared them according to the adopted model, test generation approach, usage of a compositional strategy, evaluation method, and tool support. We report results in [Dam11];
- Definition of the sequential, parallel and interruption compositional operators;
- Implementation of the compositional operators in a tool;
- Execution of exploratory studies to improve operators definition;

- Improvements on the formal definition for each operator;
- Execution of further exploratory studies to evaluate the applicability of the approach;
- Definition of proofs about the compositional properties of operators when faced to the tioco conformance relation.

1.5 Outline of the Document

The remaining parts of this document are structured as follows:

Chapter 2 This chapter contains the basic concepts of real-time systems and the testing concepts we use, including a general view of model-based testing, conformance testing, symbolic model-based testing and compositional testing;

Chapter 3 We show the definition of the sequential, interruption and parallel operators. Moreover, we present proofs based on the compositionality properties of these operators. Finally, we show examples and our test case generation process;

Chapter 5 This chapter shows two exploratory studies that use the sequential, interruption and parallel operators. We implement and execute test case results for each system;

Chapter 6 This chapter presents the works related to our approach and compare them to our work;

Chapter 7 This chapter presents the answers to our research questions along with future work.

1.6 Publications

The following papers were produced from the work developed in this thesis, in this order:

1. DAMASCENO, A. C. ; ANDRADE, W. L. ; MACHADO, P. D. L. *Automatic Test Generation of Compositional Real-Time Systems.* In: 2nd Workshop on Theses and Dissertations of CBSoft (WTDSoft 2012), 2012, Natal - RN. Proceedings of the 2nd Workshop on Theses and Dissertations of CBSoft / 3rd Brazilian Conference on Software: Theory and Practice, 2012, 2012. v. 06. p. 60-66.

2. DAMASCENO, A. C.; MACHADO, P. D. L.; ANDRADE, W. L. *Testing Real-Time Systems from Compositional Symbolic Specifications*, submitted on July 15, 2014 to Software Tools for Technology Transfer Journal;

3. DAMASCENO, A. C. ; Machado, P. D. L. ; ANDRADE, W. L.; TORRES, W. N. M. *Symbolic Test Case Generation of Compositional Real-Time Systems Driven by Interruptions.* In: IEEE Symposium on Real-time Computing, 2015, AUCKLAND. Proceedings of the IEEE Symposium on Real-time Computing, 2015. v. 18, p 228-235.

Chapter 2

Background

This chapter provides the basic concepts for our work. Section 2.1 shows basic concepts on parallel computing. Section 2.2 presents characteristics of real-time systems. Section 2.3 shows general concepts in which software testing for real-time systems are based on, as well as a general background on model-based testing (Section 2.3.1) and conformance testing (Section 2.3.2.1).

2.1 Parallel Computing

Many key activities in interdisciplinary research such as climate modeling, protein folding and data analysis require high computer performance. As the computational power increases, other problems appear and require more performance. The computational power needed to perform these activities is achieved by an increasing density of transistors in the microprocessors that process activities [Pac11]. Additionally, Moore's Law assures that transistors density in a microprocessor doubles every 18 to 24 months [M+65].

The speed of integrated circuits may not follow the need from research problems due to limitations imposed by the technology associated to microprocessors [Tan07]. With this increasing necessity, sequential programs are rewritten to become parallel and the performance be increased accordingly. The writing of parallel programs uses coordination so that work is done by using several processors (or cores).

Parallel computing is based on processes and threads [CT05]. A *process* contains

the program instructions and the data used by the program. Processes do not share memory unless the operating system allows to do so. A *thread* is a unit of control within a process that executes a function in the program and is created by using commands from a programming language. *Multithreaded programs* allow the execution of more than one thread and hence the execution of multiple tasks.

A common way of building programs that use parallel computing is the *message passing* paradigm [Pac11]. Here, a system is composed by threads, named subsystems, that communicate data through messages from subsystems to communicate effects of the program execution. In the *synchronous message passing*, processes communicate through messages that are synchronized with no delay and forcing one process to wait for each other. With *asynchronous message passing*, it is possible for the process to receive messages while being busy. Consequently, messages are buffered so that processes are not postponed by others.

On the other hand, Parallel, distributed and concurrent computing are misleading concepts and there is no general agreement on these terms [Pac11]. This happens because they describe systems that perform more than one task at a time. *Concurrent computing* [CT05] denotes tasks of a program which interact over shared data, performing tasks simultaneously because this is part of a system functionality. For example, the same processor perform multiple tasks. *Parallel computing* [Pac11] allows tasks to run simultaneously to speed up computational results. For instance, multiple and possible related tasks are running in two processors at the same time. *Distributed computing* [BCDK12] focuses on using multiple physical parts connected by a network. Frequently, these parts are geographically dispersed and communication is constrained by network protocols. Although systems are broken into parts, they form a set that behaves as a single system. For example, tasks are performed on different machines in a network.

2.2 Real-time Systems

Currently, real-time systems have a range of applications. We can find examples in multimedia and health care monitoring systems, cell phones, engine controllers in cars

and washing machines. Their correctness depend on the system behavior and when tasks are executed [Kop11].

Consequently, these systems are based on the concept of time. The present point in time distinguishes between *past* and *future*. The *state* of a system unleashes its past and future behaviors, referring to its present point. A *digital clock* partitions the time into a sequence of equally and discrete spaced intervals, while an *analog clock* gives the notion of time continuity. *Time constraint* [Lap09] specifies the response time and the temporal behavior of real-time systems. A time constraint for a health monitoring system can be exemplified by a situation where the patient's heart stops beating for more than 5 seconds, and a result is that an emergency call is sent to doctors. These constraints come from design and safety decisions imposed by the systems developer.

The instant when a result must be produced is called a *deadline* [Li03]. The deadline is called *firm* if the result cannot be used after time passes by, otherwise it is called *soft*. If missing a deadline leads to severe consequences, it is named *hard*. For example, airplane systems are ruled by hard deadlines once that if an aircraft does not obey its route within a predefined time, there may be a crash. A real-time system that contains at least one hard deadline is called *hard real-time system*. If no hard deadline exists, the system is called *soft real-time system*.

Every real-time system may be able to react to external stimuli within time intervals. Hence, we call them *reactive*. These reactions are guided by the *real-time application*, which is a set of tasks that implements the system functionality. Frequently, complex real-time applications can be decomposed into simpler parts called *components* that interact through actions [CT05]. Components can be reused by taking into account their interface specifications without the need to understand the internal details.

2.3 Software Testing

The goal of software testing is to find and fix inconsistencies before delivering to the end user. This activity executes a model or program by using inputs to assess the software behavior. The test case is used to set the initial conditions and post-conditions for the

program execution, along with the performed steps.

The testing process comprises some fundamental concepts. Six of them are "fault" , "mistake", "error", "failure", "validation" and "verification" [DMJ07; Gal04]. A *fault* denotes an incorrect requirement in the implementation or the configuration of the system that produces an incorrect behavior. It can be a project or software development process inconsistency, as well as an incorrect data definition. A *mistake* is a human action that produces a fault. These two concepts are static since they do not depend on a specific program execution. The occurrence of a fault can lead to an *error* during a program execution, which is an unexpected system behavior. This situation can cause a *failure*, making the user observe an unexpected behavior at program execution. When opposed to errors, failures are easier to detect because they produce a behavior which is external to the system. *Validation* is to assess the degree to which a system meets its requirements in order to meet the user needs. This activity leads to the question: "Are we building the right product for the end user?". *Verification* is checking the conformance of an implementation with respect to a specification. We intend to answer the question: "Are we building the product right?". Consequently, verification is a check of consistency between two descriptions, in contrast to validation which compares a description against the user needs.

When focusing on the testing of real-time concurrent programs, we face two main issues: sequence feasibility, probe effect and timeliness [CT05]. A sequence of actions allowed by a program is named a *feasible sequence*. Testing concurrent programs involves determining if a sequence of actions is feasible or not. For instance, a failure may happen due to a synchronization problem. *Probe effect* is a change in the system behavior due to measuring that system and code instrumentation causes this undesirable change. Programs may fail when instrumentation code is removed and inconsistencies may disappear when debugging code is added. *Timeliness* intends to evaluate if the software meets time constraints.

Testing software is important because it is an evaluation technique that allows the examination of the implementation behavior in the environment of the real application. For instance, the system being tested may contain an unexpected behavior because of the misunderstanding of a system module or general requirements that are used by more

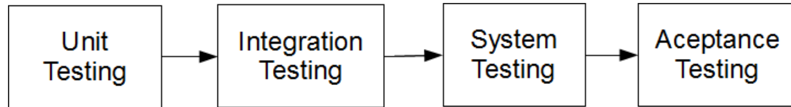


Figure 2.1: General testing process

than one part of a system. Each situation requires different ways of finding faults, since the testing of the isolated and integrated subsystems are different. Because of that, the testing phases are defined as unit, integration, system and acceptance testing [DMJ07]. Figure 2.1 shows this process.

The main goal of *unit testing* is to assess if each subsystem is correct with respect to its specification. We understand that a subsystem is the inner part of a system to be tested, that is, procedures, functions, methods or classes. Unit tests can be applied as soon as each subsystem is finished, no matter if other subsystems are still being developed.

We use the *integration testing* phase to assess the interaction between subsystems in their working groups. This phase evaluates the interfaces between subsystem and how they can be grouped. Some common types of integration testing approaches are big-bang, top-down, bottom-up and sandwich [Bin00].

In the *big-bang approach*, most of the developed subsystems are coupled together to form a complete software system for which the integration testing can be applied. Because of this, it is the least expensive and the fastest approach. The *top-down* approach begins by testing a top level subsystem and progressively adds lower level subsystems one by one. Generally, it simulates lower level subsystems by stubs. Conversely, the *bottom-up* approach integrates subsystems by making the lowest level components be tested first. After that, the testing of higher level components is performed. This process is repeated until the component at the top of the hierarchy is tested. The sandwich testing uses the bottom-up and top-down approaches at the same time.

The *system testing* phase starts after previous phases are performed. Its main goal is to evaluate if the features from the requirements document are implemented. In some cases such as the testing of embedded systems, we also test its non-functional requirements and how the program reacts to the embedded system.

Acceptance testing evaluates if a product works for the user. It consists in con-

ducting tests to determine if specification requirements are meet and the user accepts the solution. Generally, this phase intends to examine the application functionality without taking its internal structure into consideration.

During a software testing process, a great amount of time is used to figure out what the system might do. The reason is that it is difficult to discover if a result is correct or not. Thus, a model is very important because it can capture the system behavior. Moreover, it describes the structure of the input space, allowing testers to use information from software requirements and design.

2.3.1 Model-Based Testing

A model is an abstract description of the system and can be used to insert information about faults and other data that help in the test case generation process. *Model-based testing* uses models that abstract specific system behaviors to produce test cases that reveal divergences between the implementation and the specification models [DMJ07].

Usually, the tester receives the requirements document and produces a model from it. In the sequence, test cases that comprise inputs and expected outputs are generated. Inputs guide the implementation execution to produce results that are compared to the expected outputs of the test case. These outputs are produced by the *test oracle*, which is responsible for evaluating if the test passed or failed. If the test case is successful, there is no action to be taken. Otherwise, feedbacks to the implementation, model and requirements are produced in order to improve the whole system. Figure 2.2 shows a summary for this process.

Test cases can be executed using several approaches. Two common methods are online and offline testing [DNSVT07]. *Online testing* means that a model-based testing tool connects directly to an implementation and performs dynamic tests. *Offline generation* refers to a tool that generates test cases as computer instructions that are stored to be executed later.

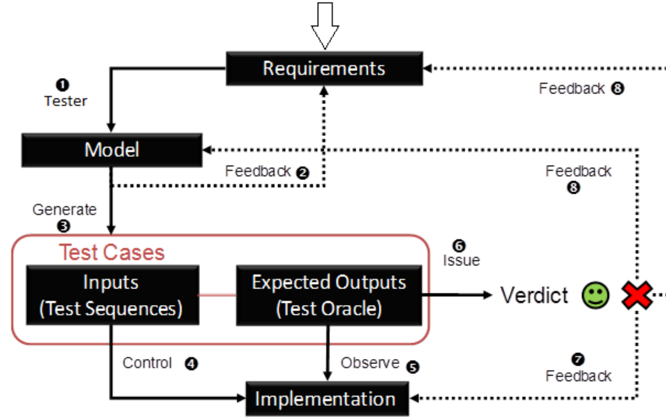


Figure 2.2: Model-based testing approach (adapted from [Kic14])

2.3.2 Conformance Testing

Implementations and specifications may be abstracted into models that are compared under some criteria. *Conformance testing* [Gar05] emerges to check that the implementation meets predefined requirements from the specification by means of testing. A *specification* $\mathcal{S}$ is a model of what the system should do and contains the goals to be checked on the testing process. An *implementation* $\mathcal{I}$ is described by using the same model used for the specification. Conformance testing has many applications. For instance, compilers (implementations) are extensively tested to determine if they contain deviations from programming language standards (specifications) and is guided by test derivation [bG14; 98914b; 98914a].

Treatmans [Tre99] describes the basics of conformance testing. He claims that the conformance testing process has two main phases: test generation and test execution. *Test generation* analyses and determines how the specification is going to be tested and develops test scripts. *Test execution* refers to a test environment which is developed to execute test scripts. This execution results in a verdict about the behavior of the implementation with regard to the specification.

Generally, conformance testing and formal methods are used to define frameworks. Each specification $\mathcal{S}$ belongs to a set of formal specifications $SPECS$, while an implementation IUT is contained in a set of implementations $IMPS$. In this way, conformance is defined as the relation **conforms-to** $\subseteq IMPS \times SPECS$ and $\mathcal{I}$ **conforms-to** $\mathcal{S}$ means that $\mathcal{I}$ is a correct implementation of $\mathcal{S}$.

Conversely to specifications, implementations are real pieces of hardware or software. This may cause difficulties on expressing a formal implementation behavior. To delimit the scope of specifications and implementations, the following assumption is made: any real implementation can be modeled by a formal object $\mathcal{I} \in MODS$, where $MODS$ is the universe of models. This assumption is the *test hypothesis*.

The test hypothesis assumes that implementations are formal objects. Thus, it is possible to express conformance by the formal relation between implementations, specifications and models. This relation, named *implementation relation*, is defined by **imp** $\subseteq MODS \times SPECS$. An implementation IUT $\in IMPS$ is correct with respect to $\mathcal{S} \in SPECS$, or IUT **conforms-to** $\mathcal{S}$, if and only if the model $\mathcal{I} \in MODS$ of IUT is **imp**-related to $\mathcal{S}$: $\mathcal{I}$ **imp** $\mathcal{S}$.

Conformance testing uses a wide range of models. Most of them are based on labeled transition systems and state machines [You08], but they are limited by the state explosion problem. On the other hand, the increasing system complexity requires that conformance relations allow composition operators usage. Symbolic models lessen the state explosion problem by preserving the behavior of infinite systems and corresponding values to symbolic expressions. Section 2.3.2.1 presents issues on conformance testing for composition subsystems and Section 2.3.2.2 shows an example of a model suitable for real-time systems.

2.3.2.1 Conformance Testing of Compositional Models

Due to the increasing complexity of real-time systems, they are divided into subsystems that interact through composition operators. The testing process complexity of the overall system follows the system complexity. Test generation of composed systems becomes an important task, as well as inferring properties from subsystems to the composed systems.

However, conformance relations that are based on inputs and outputs are not preserved from subsystems to the composed system [ABK12]. More specifically, given two specifications $\mathcal{S}_1$ and $\mathcal{S}_2$, two implementations $\mathcal{I}_1$ and $\mathcal{I}_2$, an operation **op**, and a conformance relation **conforms-to** based on inputs and outputs, if $\mathcal{I}_1$ **conforms-to** $\mathcal{S}_1$ and $\mathcal{I}_2$ **conforms-to** $\mathcal{S}_2$, can we infer that $\mathcal{I}_1$ **op** $\mathcal{I}_2$ **conforms-to** $\mathcal{S}_1$ **op** $\mathcal{S}_2$?

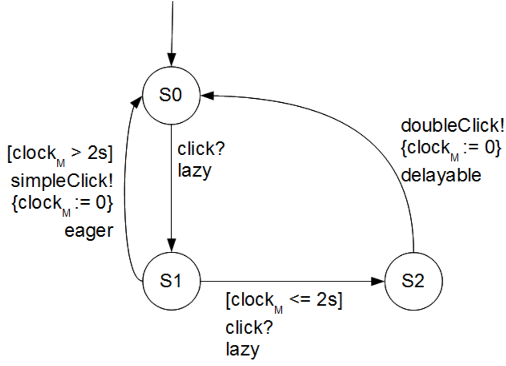


Figure 2.3: TAIO for the *Mouse* subsystem

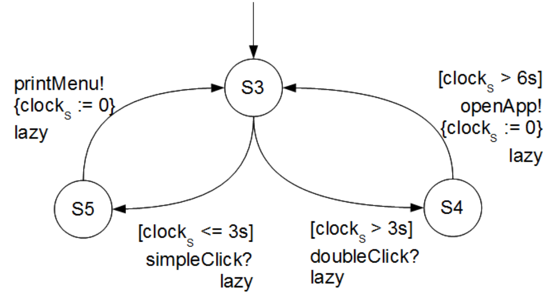


Figure 2.4: TAIO for the *Screen* subsystem

We exemplify this difficulty by using the TAIO model and tioco conformance relation presented by Krichen and Tripakis [KT06].

The authors adopted the TAIO (Timed Automata with Inputs and Outputs) model. Basically, it is a timed automata with a partitioned set of input, output and internal actions. In addition, it has sets of clocks, locations, transitions, actions and an initial state where the model starts. Figures 2.3 and 2.4 show two examples of the TAIO model which represent generic mouse and screen systems, in this sequence.

Transition deadlines impose the latest time by which a transition must be taken, assuming three possible values: i) *lazy*, imposes no urgency for a transition to be taken; ii) *delayable*, imposes that a transition must be taken before its condition becomes disabled, and iii) *eager*, imposes that a transition must be taken as soon as its condition becomes enabled. Definition 1 shows the formal concepts of the TAIO model.

Definition 1 (Timed Automata with Inputs and Outputs). *A Timed Automata with Inputs and Outpus (TAIO) is a tuple (Q, q_0, X, Ac_τ, E) where $Ac_\tau = Ac_{in} \cup Ac_{out} \cup \{\tau\}$, Q is a finite set of locations, X is a finite set o clocks and E is a finite set of transitions. Each transition is a tuple (q, q', ψ, r, d, a), where:*

- *$q, q' \in Q$ are the source and destination locations,*
- *ψ is the guard,*
- *$r \subseteq X$ is a set of clocks to reset to zero,*
- *$d \in \{lazy, delayable, eager\}$ is the deadline,*

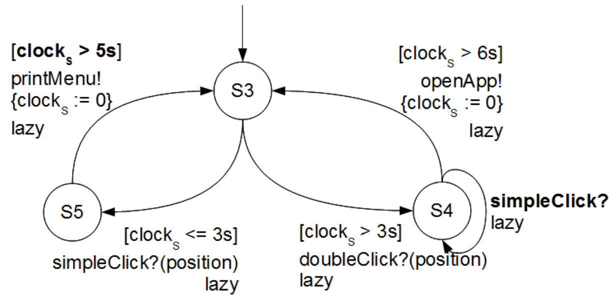


Figure 2.5: TAIO *Screen* conformant

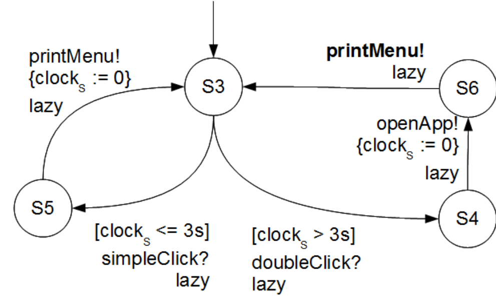


Figure 2.6: TAIO *Screen* not conformant

- *a is a conjunction of constraints of the form* $x\#c$*, where* $x \in X$*, c is an integer constant and* $\# \in \{<, \leq, =, \geq, >\}$.

The test assumptions are: the specification $\mathcal{S}$ of the system under test must be given as a non-blocking TAIO, that is, the system spends a discrete and finite period of time to execute any reachable action from the model. In addition, the implementation $\mathcal{I}$ is modeled as a non-blocking TAIO whose every location contains an output transition for every input action from this model. The tioco conformance relation can be used to test $\mathcal{I}$ against $\mathcal{S}$. In other words, the relation $\mathcal{I}$ tioco $\mathcal{S}$ expresses that $\mathcal{I}$ conforms to $\mathcal{S}$ if after any trace σ of the specification, every output action that $\mathcal{I}$ is capable of showing should be allowed by $\mathcal{S}$, as well as the time restrictions imposed by $\mathcal{S}$.

Figure 2.5 shows an example of implementation for the *Screen* specification from Figure 2.4 that is in tioco conformance to its specification. Although transition S4-S4 contains the additional *simpleClick?* input event, this omission is allowed by the tioco conformance relation. Furthermore, the $clock_s > 5s$ guard is contained in the true set of the corresponding trace of its specification. On the contrary, the implementation shown in Figure 2.6 is not tioco conformant to its specification because it has the additional *printMenu!* output action belonging to the S6-S3 transition.

Using the TAIO model, the parallel composition operator was defined. It interleaves actions which are uncommon to both models and synchronizes actions which belong to the set of input actions of one model and the set of output actions of the model at the same time. Figure 2.7 shows an example of parallel composition between the *Mouse* and *Screen* TAIO models. The subsystems synchronize on *simpleClick* and *doubleClick* actions, leading the (S1,S3)-(S0,S5) and (S2,S3)-(S0,S4) transitions to be replaced by

the τ internal action.

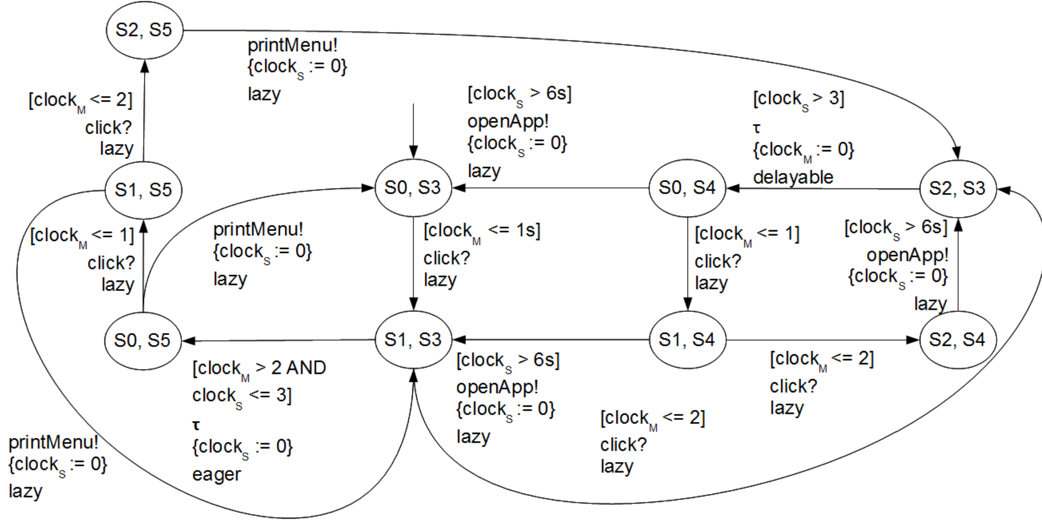


Figure 2.7: Parallel composition for the *Mouse* and *Screen* specifications using the TAIO model

Given two subsystems $\mathcal{S}_1$ and $\mathcal{S}_2$, the parallel composition $\mathcal{S}_1 \parallel \mathcal{S}_2$ is restricted by some rules. The subsystems must have disjoint sets of clocks, every input action from their synchronization sets must be associated to the deadline *lazy* (*lazy-input*) and they are input-complete with regard to this same set. The two TAIO synchronize on time and shared common actions that belong to the sets $Ac_{2\to1}$ and $Ac_{1\to2}$, being pairwise disjoint and, when connected to each other, the interaction between subsystems is assumed to be unobservable from the outside. Moreover, Ac^1_{in}, Ac^1_{out}, Ac^2_{in} and Ac^2_{out} are pairwise disjoint sets of input and output actions which do not belong to the synchronization set of the $\mathcal{S}_1$ and $\mathcal{S}_2$ models. Definition 2 presents the parallel operator.

Definition 2 (tioco Parallel Composition)**.** *Let $\mathcal{S}_i$ ($i \in \{1,2\}$) be two TAIO with the disjoint sets of unsynchronizable input and output actions Ac^i_{in} and Ac^i_{out} and $\mathcal{S}_i$ are both input-complete and lazy-input with regard to their synchronizable action set $Ac_{(3-i)\to i}$. The parallel composition $\mathcal{S}_1 \parallel \mathcal{S}_2 = (Q_1 \times Q_2, (q^1_0, q^2_0), X_1 \cup X_2, Ac_\tau, E)$ where $Ac_\tau = Ac_{in} \cup Ac_{out} \cup \tau$, $Ac_{in} = \bigcup_{i\in\{1,2\}} Ac^i_{in}$, $Ac_{out} = \bigcup_{i\in\{1,2\}} Ac^i_{out}$ and E is the smallest set such that:*

- *For $(q_1, q_2) \in Q_1 \times Q_2$ and $a \in Ac^1_{in} \cup Ac^1_{out} \cup \{\tau\}$:*

$$(q_1, q'_1, \psi_1, r_1, d_1, a) \in E_1 \Rightarrow ((q_1, q_2), (q'_1, q_2), \psi_1, r_1, d_1, a) \in E;$$

- *For* $(q_1, q_2) \in Q_1 \times Q_2$ *and* $a \in Ac^2_{in} \cup Ac^2_{out} \cup \{\tau\}$*:*

  $(q_2, q_2', \psi_2, r_2, d_2, a) \in E_2 \Rightarrow ((q_1, q_2), (q_1, q_2'), \psi_2, r_2, d_2, a) \in E$*;*
- *For* $a \in Ac_{1\to2} : (q_1, q_1', \psi_1, r_1, d_1, a) \in E_1 \wedge (q_2, q_2', \psi_2, r_2, d_2, a) \in E_2$

  $\Rightarrow ((q_1, q_2), (q_1, q_2'), \psi_1 \wedge \psi_2, r_1 \cup r_2, d_1, \tau_a) \in E$*;*
- *For* $a \in Ac_{2\to1} : (q_1, q_1', \psi_1, r_1, d_1, a) \in E_1 \wedge (q_2, q_2', \psi_2, r_2, d_2, a) \in E_2$

  $\Rightarrow ((q_1, q_2), (q_1, q_2'), \psi_1 \wedge \psi_2, r_1 \cup r_2, d_2, \tau_a) \in E$*;*

Unfortunately, the tioco confomance relation is not preserved from subsystems to the composed systems. If we consider the specifications $\mathcal{S}_1$ and $\mathcal{S}_2$ and their corresponding implementations $\mathcal{I}_1$ and $\mathcal{I}_2$, assuming that $\mathcal{I}_1$ tioco $\mathcal{S}_1$ and $\mathcal{I}_2$ tioco $\mathcal{S}_2$, $\mathcal{I}_1 \parallel \mathcal{I}_2$ tioco $\mathcal{S}_1 \parallel \mathcal{S}_2$ if both specifications are input-complete. This happens because the tioco conformance relation allows underspecification of inputs, but not the underspecification of outputs. Consequently, new transitions with output actions are generated in the resulting composed implementation if confronted to its corresponding composed specification. Theorem 1 presents these results, and its proof is available in [KT06].

Theorem 1 (Compositionality of tioco for the parallel operator)**.** *Let* $\mathcal{I}_1$*,* $\mathcal{I}_2$*,* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *be input-complete TAIO. If* $\mathcal{I}_1$ tioco $\mathcal{S}_1 \wedge \mathcal{I}_2$ tioco $\mathcal{S}_2$*, then* $\mathcal{I}_1 \parallel \mathcal{I}_2$ tioco $\mathcal{S}_1 \parallel \mathcal{S}_2$*.*

Krichen and Tripakis also presented an approach to perform the input-completion of TAIO models. However, the approach is limited to deterministic and *fully observable* (not having internal actions) specifications. Figure 2.8 shows an example of the input-complete version of the TAIO *Mouse* specification. The "don't care" location Q_{dc} is the destination of every transition that contains underspecified input actions in the original specification, as (S5-Q_{dc}) location exemplifies. Moreover, if the input action is present but the guard is not true, the negation of this guard with the same input action is added to the input-complete model. For instance, (S3-Q_{dc}) transition adds the transition with the *doubleClick?* input event associated to the guard $clock_s <= 3\ s$.

The input-complete process is done by adding edges covering the missing inputs and leading to a "don't care" location where all inputs and outputs are accepted. This process is formally presented in Definition 3. The proof is shown in [KT06].

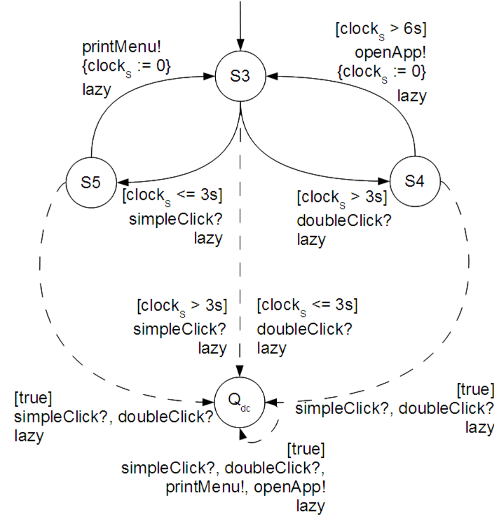


Figure 2.8: Input-completion for the *Screen* TAIO specification

Definition 3 (TAIO Input-completion Transformation). *Given a TAIO* $\mathcal{S} = (Q, q_0, X, Ac, E)$, *we build a corresponding input-complete TAIO* $\tilde{\mathcal{S}} = (\tilde{Q}, q_0, X, Ac, \tilde{E})$, *where* $\tilde{Q} = Q \cup \{q_{dc}\}$, $q_{dc} \notin Q$ *is the "don't care" location, and*
$\tilde{E} = E \cup \{(q_{dc}, q_{dc}), true, \emptyset, lazy, a) | a \in Ac\} \cup \{(q, q_{dc}), \neg\psi, \emptyset, lazy, a) | q \in Q \wedge a \in Ac_{in}\}$ *such that for each* $q \in Q$ *and each* $a \in Ac_{in}, \psi = \psi_1 \vee \psi_2 \vee \dots \psi_k$ *are the guards of the outgoing edges of* q *labeled with* a.

2.3.2.2 Conformance Testing of Symbolic Models

Symbolic model based-testing has its foundations on model-based testing, conformance testing and symbolic execution. *Symbolic execution* builds predicates that define conditions under which execution paths can be taken and the effects of the execution on program state can be verified [You08; Kin76]. This approach is useful to identify unfeasible paths or paths that lead the tester to a fail verdict.

We use Timed Input-Output Symbolic Transition Systems (TIOSTS) [AMJM11; AM13] as special kinds of symbolic transition systems where time constraints can be modeled by using a new type of variable to manage time. More specifically, a TIOSTS comprises an initial condition that must be satisfied prior to system execution along with a set of transitions containing the following elements: i) a source and a target location; ii) a condition that must be satisfied to allow the transition to be fired; iii) a deadline that imposes the latest time by which the transition can be executed; iv)

an input or output action that communicates one or more parameters and a set of assignments that change variable values. Definition 4 formalizes this model.

Definition 4 (TIOSTS)**.** *Timed Input-Output Symbolic Transition System is a tuple* $\langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$*, where:*

- V *is a finite set of typed variables;*
- P *is a finite set of parameters;*
- Θ *is the initial condition, a predicate with variables in* V*;*
- L *is a finite, non-empty set of locations and* $l^0 \in L$ *is the initial location;*
- $\Sigma = \Sigma^? \cup \Sigma^!$ *is a non-empty, finite alphabet, in which there is the disjoint union of the set* $\Sigma^?$ *of input actions and the set* $\Sigma^!$ *of output actions. For each action* $a \in \Sigma$*, its signature* $sig(a) = \langle p_1, \ldots, p_n \rangle$ *is a tuple of distinct parameters, where each* $p_i \in P (i = 1, \ldots, n)$*;*
- C *is a finite set of clocks with values in the set of non-negative real numbers, denoted by* $\mathbb{R}^{\geq 0}$*;*
- $\mathcal{T}$ *is a finite set of transitions. Each transition* $t \in \mathcal{T}$ *is a tuple* $\langle l, a, G, A, y, l' \rangle$ *where:*
  - $l \in L$ *is the origin location of the transition,*
  - $a \in \Sigma$ *is the action,*
  - $G = G^D \wedge G^C$ *is the guard, where* G^D *is a predicate with variables in* $V \cup set(sig(a))$[1,2] *and* G^C *is a clock constraint over* C *defined as a conjunction of constraints over* C *in the format* $\alpha \# c$*, where* $\alpha \in C$*,* $\# \in \{ <, \leq, =, \geq, > \}$ *and* $c \in \mathbb{N}$*,*
  - $A = A^D \cup A^C$ *is the assignment of the transition. For each variable* $x \in V$ *there is exactly one assignment in* A^D *using the format* $x := A^{D^x}$*, where*

[1] G^D is assumed to be expressed in a theory in which satisfiability is decidable.

[2] Let $set(j)$ be the function that converts the tuple j in a set.

A^{D^x} is an expression on $V \cup set(sig(a))$. $A^C \subseteq C$ is a set of clocks to be reset,

- *$y \in \{lazy, delayable, eager\}$ is the deadline of the transition,*
- *$l' \in L$ is the destination location of the transition.*

TIOSTS models include distinct sets of locations, variables, parameters and clocks. Since we are dealing with a symbolic model, the set of states is represented by locations. Also, clock constraints are represented by using zones, which is the maximum set of clock arguments satisfying a constraint [BY04]. Accordingly, variables and parameters refer to symbolic values, avoiding the state explosion problem, but they have one slight difference: their model scope. Parameters are used to communicate values from one model to another or to the environment. Moreover, parameters scope is restricted to a single transition. On the other hand, variables have global scope inside the model.

For the sake of simplicity, transition deadlines can be omitted from the graphical visualization of a TIOSTS and, in this case, we assume by default the lazy deadline for input actions and the delayable deadline for output actions.

Figure 2.9 presents an example of a TIOSTS that models a *Mouse system* with single and double click commands. In the S0 location, this subsystem can receive a *click?* with no urgency. When the transition fires, it stores this value in the $coordinate_M$ variable and goes to the S1 location. In the S1 location, either a *click?* input action can be received within $2s$ without urgency (S1 to S2 transition) or a *simpleClick!* output can be issued after $2s$ with eager urgency, leading to clock resetting (S1 to S0 transition). In the latter situation, the guard $coordinate_M = position$ communicates the $coordinate_M$ variable to other subsystems or the environment through the *simpleClick!* output action with eager deadline, forcing the parameter *position* to assume a value equal to the $coordinate_M$ variable value. In the S2 location, the $coordinate_M$ variable value is communicated to other subsystems or the environment through the *doubleClick!* output action with delayable deadline, also forcing parameter *position* to assume a value equal to the $coordinate_M$ variable value (see guard $coordinate_M = position$). Also, the clock is reset and the system goes to the S0 location.

Additionally, consider the *Screen* specification (Figure 2.10) that is responsible for

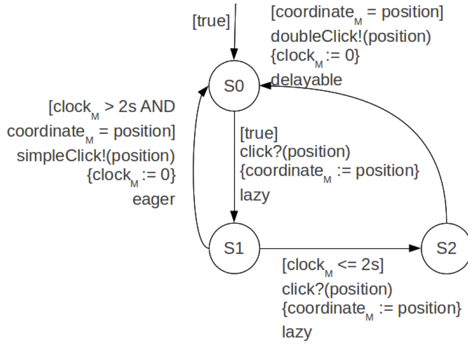


Figure 2.9: *Mouse* subsystem

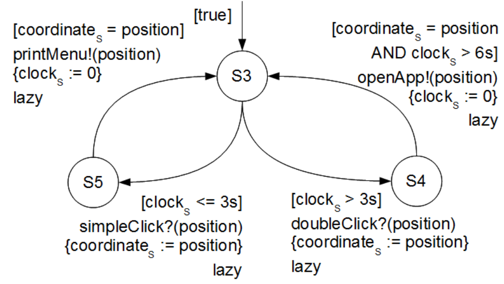


Figure 2.10: *Screen* subsystem

showing the consequences of the mouse commands on a general computer screen. In this sense, it should run in parallel with the *Mouse* application by synchronizing through the *simpleClick* and *doubleClick* actions. The *Screen* behavior has two main scenarios: 1) If the *Mouse* subsystem outputs the *position* value through the *simpleClick!* output action, the corresponding *simpleClick?* action from the *Screen* application inputs its *position* value within $3s$ and prints a menu by using the *printMenu!* output action; 2) If the *Mouse* subsystem outputs the mouse pointer position by using the *doubleClick!* output action, the *Screen* subsystem will respond to it by receiving this position through the *doubleClick?* input action and opening an application through the *openApp!* output action within $6s$.

Based on the tioco conformance relation [KT06], TIOSTS models can be used for conformance testing focusing on whether observable behavior of a given implementation is in conformance with a specification. We assume the testing hypothesis that the implementation can be abstracted as an implementation model, which will be referred simply as implementation further on (for simplicity). It is important to remark that the implementation model is not required to be constructed in a conformance testing process but rather assumed to exist [Tre99]. The examples of implementations provided in this section and throughout this paper are devoted solely for illustration of the theory. In practice, from a TIOSTS specification model, test cases are generated to test a real implementation.

The tioco conformance theory is exemplified by using the *Screen* subsystem specification from Figure 2.10 and two possible implementations in Figures 2.11 and 2.12. Since tioco allows underspecification of inputs, the implementation of Figure 2.11 con-

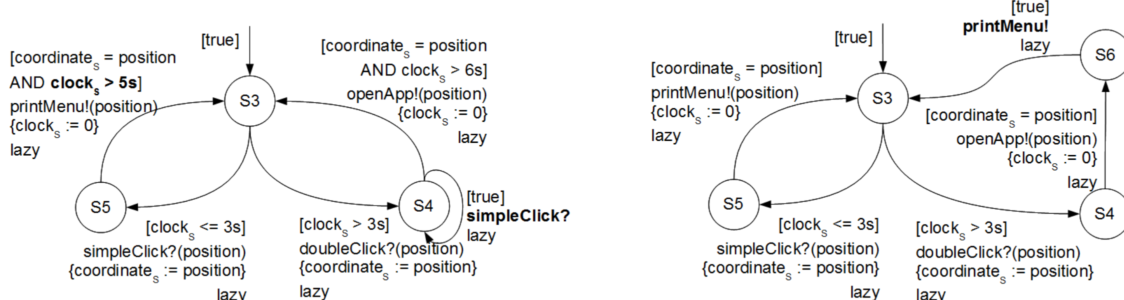


Figure 2.11: Model in conformance with *Screen*

Figure 2.12: Model not in conformance with *Screen*

forms to its specification even if presenting an additional transition from the S4 location (it contains the *simpleClick?* input action). Also, the clock guard from the S5 outgoing transition contains the $clock_S > 5s$ guard that represents a set that is contained in the set represented by the $G_c = true$ guard from the S5 outgoing transition of the model presented in Figure 2.10. Nevertheless, the implementation of Figure 2.12 does not conform to its specification because location S6 has an outgoing transition with the extra output action *printMenu!*. Besides, location S6 has an incoming transition with $G_c = true$, opposing to the $clock_S > 6s$ condition from the S4 outgoing transition of its specification model.

TIOSTS semantics is defined in terms of the TIOLTS model [AM13]. Basically, TIOLTS states instantiate the set of locations, valuations and clocks of a TIOSTS. The transition relation of the TIOLTS uses rules of inferences, where a preconditions are predefined to achieve a postcondition [BBJ02]. In this way, we need to understand elements from TIOLTS, presented in Definition 5.

Definition 5 (TIOLTS semantics of a TIOSTS)**.** *The semantics of a TIOSTS* $W = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T}\rangle$ *is a TIOLTS* $[\![W]\!] = \langle S, S^0, Act, T\rangle$*, defined as follows:*

- $S = L \times \mathcal{V} \times (C \to \mathbb{R}^{\geq 0})$ *is the set of states of the form* $s = \langle l, \nu, \psi\rangle$*, where* $l \in L$ *is a location from the set of locations* L*,* $\nu \in \mathcal{V}$ *is a valuation for all variables* $\mathcal{V}$*,* ψ *is a clock valuation from the set* C *of clock valuations.*

- $S_0 = \{\langle l^0, \nu, \psi\rangle | \Theta(\nu) = true, \overline{0}\}$ *is the set of initial states.*

- $Act = \Lambda \cup D$ *is the set of actions, where* $\Lambda = \{\langle a, \gamma\rangle \mid a \in \Sigma, \gamma \in \Gamma_{sig(a)}\}$ *is*

the set of discrete actions and $D = \mathbb{R}^{\geq 0}$ *is the set of time-elapsing actions.* Λ *is the union of the sets* $\Lambda^{?}$ *of input actions,* $\Lambda^{!}$ *of output actions and* $\Gamma_{sig(a)}$ *is the sequence of parameters associated to a transition action.*

- T *is the transition relation defined as the minimum set of the following rules:*
  - *Discrete actions:*

$$\frac{\begin{array}{ll} \langle l, \nu, \psi \rangle, \langle l', \nu', \psi' \rangle \in S & \langle a, \gamma \rangle \in \Lambda \\ t : \langle l, a, G, A, y, l' \rangle \in \mathcal{T} & G = true \\ \nu' = A^{D}(\nu, \gamma) & \psi' = \psi[A^{C} \leftarrow 0] \end{array}}{\langle l, \nu, \psi \rangle \xrightarrow{\langle a, \gamma \rangle} \langle l', \nu', \psi' \rangle}$$

  - *Time-elapsing actions:*

$$\frac{\begin{array}{l} d \in D \quad \langle l, \nu, \psi \rangle, \langle l, \nu, \psi + d \rangle \in S \\ t : \langle l, a, G, A, y, l' \rangle \in \mathcal{T} \\ y = eager \Rightarrow \psi \not\models G^{C} \\ y = delayable \Rightarrow \nexists d_1, d_2 \in D : \\ 0 \leq d_1 < d_2 \leq d \wedge \psi + d_1 \models G^{C} \wedge \psi + d_2 \not\models G^{C} \end{array}}{\langle l, \nu, \psi \rangle \xrightarrow{d} \langle l, \nu, \psi + d \rangle}$$

Let $s, s', s_i \in S$ be states; $a, a_i \in Act$ be actions; and $\sigma \in Act^*$ be a sequence over Act, where $\epsilon \in Act^*$ is the empty sequence. If $\sigma_1, \sigma_2 \in Act^*$, then $\sigma_1 \cdot \sigma_2$ is the concatenation of σ_1 and σ_2. We write $s \xrightarrow{a} s'$ for $(s, a, s') \in T$, $s \xrightarrow{a}$ for $\exists s' : s \xrightarrow{a} s'$. Let $s \xrightarrow{a_1 \ldots a_n} s' \triangleq \exists s_0, \ldots s_n : s = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n = s'$ be an execution. In addition, we use $s \xrightarrow{\sigma}$ for $\exists s' : s \xrightarrow{\sigma} s'$. Moreover, considering l and $l' \in L$, we write $l \xrightarrow{a} l'$ for $\langle l, a, G, A, y, l' \rangle \in \mathcal{T}$, omitting G, A and y to improve readability.

The set of fireable actions from s is defined by $\Omega(s) \triangleq \{a \in Act \,|\, s \xrightarrow{a}\}$. The set of all output actions (including time-elapsing actions) fireable from s is defined by $Out(s) \triangleq \Omega(s) \cap (\Lambda^{!} \cup D)$ and, when considering set of states, it is defined by $Out(P) \triangleq \bigcup_{s \in P} Out(s)$. By using these elements, Definition 6 presents *Traces*(s), which returns the set of sequences and time-elapsing actions fireable from a given location.

Definition 6 (Traces)**.** *Let Act^* be a set of sequences of actions that includes the empty sequence ϵ, σ be a sequence over Act^* and s be a state. Traces(s)* $\triangleq \{\sigma \in Act^* | \ s \xrightarrow{\sigma}\}$.

The set of sequences of actions fireable from the initial state of a TIOSTS W is given by $Traces(\text{W}) \triangleq Traces(S_0)$. In addition, s *after* $\sigma \triangleq \{s' \in S \mid s \xrightarrow{\sigma} s'\}$ is the set of reachable states from s after the execution of σ; for $P \subseteq S$, P *after* $\sigma \triangleq \bigcup_{s \in P} s$ *after* σ is the set of states reachable from the set P after the execution of σ and W *after* $\sigma \triangleq S_0$ *after* σ is the set of states reachable from S_0 after the execution of σ. Based on these concepts, the timed input-output conformance relation, denoted tioco, is formalized in Definition 7 by using the TIOLTS model to express specifications and implementations.

Definition 7 (tioco)**.** *Given a specification $\mathcal{S}$ and an implementation $\mathcal{I}$, $\mathcal{I}$* tioco *$\mathcal{S} \Leftrightarrow \forall \sigma \in Traces(\mathcal{S})$: Out($\mathcal{I}$ after σ) $\subseteq$ Out($\mathcal{S}$ after σ).*

Based on the tioco and TIOSTS theory, Andrade *et al.* [AM13] developed the testing framework we use. Their test generation process assumes some characteristics for the specification and implementation that can be handled. The specification $\mathcal{S}$ is a *non-blocking* TIOSTS. This means that the system spends a discrete and finite period of time to execute any reachable action from the model. The system is not supposed to block because an input action was not provided by the environment (the system cannot force input actions). Also, the implementation $\mathcal{I}$ is a software system running on a real-time environment which is represented by an *input-complete* model, meaning that it accepts any input action at any state. In addition, it must be non-blocking and have the same input and output actions with their signatures as in specification $\mathcal{S}$.

Besides, $\mathcal{S}$ and $\mathcal{I}$ are required to be deterministic in the theory and the tool we use. Given an action $a \in \Sigma_{\mathcal{S}}$ and three states s_1, s_2 and s_3, $\mathcal{S}$ is *deterministic* if $\forall \ s_1, s_2, s_3 \in S_{\mathcal{S}}, \ s_1 \xrightarrow{a} s_2 \wedge s_1 \xrightarrow{a} s_3$ then $s_2 = s_3$. In other words, specification results must be predictable and repeatable, meaning that: i) there is only one initial state mapped from its initial location, and ii) transitions originating from the same location that contain the same action must contain mutually exclusive guards. In general, formalisms like Timed Automata (TA) cannot be determinized [HLM+08] when the test case generation is performed apart from the test case execution (offline

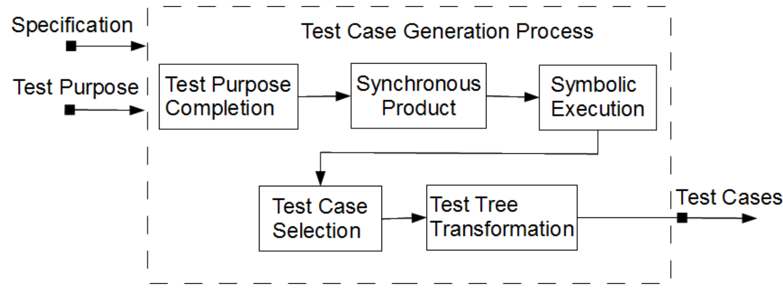


Figure 2.13: Test case generation process

test generation). Since the TIOSTS is related to TA and our tool would need to determinize models in the generation test cases process, we constrain subsystems to be deterministic.

The proposed test case generation process, presented in Figure 2.13, is guided by test purposes. A test purpose is a TIOSTS that contains the behavior we want to check in the implementation during test execution. It finishes with *Accept* or *Reject* locations, meaning that the behavior must be part of the test cases or a given scenario must be excluded from the set of test cases, respectively.

Initially, the test purpose completion step completes the definition of the test purpose provided as input. Then, the synchronous product generation phase performs a customized parallel product between the completed test purpose and the specification. This operation is defined by an algorithm that combines the test purpose and the specification model by synchronizing their actions and generating a new model that will be the basis for the next phase, focusing on the behavior specified by the test purpose.

The symbolic execution phase performs the execution of TIOSTS models by using a similar procedure adopted by the symbolic execution of programs. The goal is to provide a representation of infinite concrete sets of data and time by corresponding them to abstracted sets. However, both data and time parts are executed in distinct ways. The data part is accomplished by collecting transition guards of a path and checking them with a constraint solving. The time part involves zones to check the reachability of locations as it refers to time requirements. Consequently, a location is reachable if its path condition is satisfiable and its zone is not empty. The result is a symbolic execution tree that contains all allowed model traces.

In the selection of test cases, a sub-tree of the symbolic execution tree is chosen. A

test case is generated by selecting a trace that leads to a reachable *Accept* location in the symbolic execution sub-tree. Meanwhile, missing inputs allowed by the specification are added to the test case with *Inconclusive* verdicts.

Finally, the test tree transformation phase translates the selected test tree into a test case modeled by TIOSTS rules. Each test case leaf can finish with the *Inconclusive*, *Accept* or *Reject* verdicts. Besides, the *Inconclusive* verdict is an option if the paths that contain it lead to a behavior that conforms to the specification, but the actions from the test purpose are not presented. We highlight that the generation of the test cases is guided by the test purposes, so a test case scenario is exercised until the actions of the test purposes are found in the specification.

2.4 Concluding Remarks

This chapter presents the background for our work, divided into parallel computing, real-time systems and software testing. First, we present concepts on parallel computing, such as the message-passing paradigm and the differences between parallel, concurrent and distributed computing. In the sequence, we define some of the fundamental concepts of real-time systems, like clocks, components and deadline. In the sequence, we present a range of concepts of the software testing area: failure, fault, error, validation, verification and the software testing phases. We emphasize model-based and conformance testing, focusing on compositional properties of the ioco conformance relation and symbolic models for real-time systems that uses the tioco conformance relation.

Although conformance testing of symbolic models presents many gains during the test case generation process, there is no work that focuses on the study of problems of the tioco conformance relation and composition operators. Our work intends to contribute in this field.

Chapter 3

Test Case Generation from Compositional Models

This chapter presents the compositional operators defined for the TIOSTS model. We present the sequential, parallel and interruption operators. Their choice happens because they can be largely applied in real-time applications. Moreover, the interruption operator is an extended version of the sequential operator, lessening the effort to define it. In addition, the parallel operator is essential to parallel applications, which are frequently used nowadays.

Section 3.1 shows the sequential operator by introducing the *Choose* and *Pay* subsystems. In the sequence, Section 3.2 introduces the interruption operator through the *Mouse Interrupted* and *Reset* subsystems. Section 3.3 defines the parallel composition and applies it to the *Mouse* and *Screen* subsystems. Finally, we provide formal definitions for each operator and address implications of the tioco conformance from subsystems to the composed system. We present a test case generation process – which was developed in this work and integrated to a tool – showing test case results for each composed system (Section 3.4)[1].

[1] A complete presentation of test purposes and test cases with accept and inconclusive paths generated by using the SYMBOLRT tool as well as an implementation of the compositional operators are available at https://sites.google.com/site/compositionaltioco/

3.1 Sequential Composition

Sequential composition is applied when the interaction between two subsystems must be ordered and the first subsystem finishes before the second starts. We used the message passing paradigm to define this operator in a way that information is communicated from one subsystem to another through a single action which is present in both subsystems.

As an example, suppose that a system can be modeled by the sequential composition of the $\mathcal{S}_1$ and $\mathcal{S}_2$ TIOSTS models presented in Figure A.1. TIOSTS $\mathcal{S}_1$ has a location l_{c1} with a single incoming transition composed by the *lazy* deadline and the a_{c1} output action with parameter p_1. On the other hand, $\mathcal{S}_2$ has an input action with the same label of a_{c1} that receives parameter p_1 to be further used in $\mathcal{S}_2$ through the A_2^0 assignment. The sequential composition $\mathcal{S}_1\ ;_{a_{c1}}\ \mathcal{S}_2$ performs the synchronization of the a_{c1} action at the end of $\mathcal{S}_1$ and the beginning of $\mathcal{S}_2$.

Consequently, it allows that both subsystems be linked through the single transition $(l_{c1}^0, l_2^0) \xrightarrow{\mathrm{a_{c1}!p_1}} (l_{c1}, l_2'^0)$ that has the *lazy* deadline. This transition comprises the guard $G_{c1}\ AND\ \Theta_2\ AND\ G_2^0$, composed by $\mathcal{S}_2$ initial condition and $l_{c1}^0 \xrightarrow{\mathrm{a_{c1}!p_1}} l_{c1}$ and $l_2^0 \xrightarrow{\mathrm{a_{c1}?p_1}} l_2'^0$ transition guards, to meet system conditions. Also, we add clock resetting to A_2^0 in order to maintain the behavior of $\mathcal{S}_2$ clocks after the sequential composition, since every clock from the composite model starts in the beginning of $\mathcal{S}_1$. Finally, we replace transitions that contain l_{c1}^0 and $l_2'^0$ locations by new transitions with (l_{c1}^0, l_2^0) and $(l_{c1}, l_2'^0)$ locations.

Now consider a more practical example of a system that describes a choosing and payment process. We shall model the resulting system $\mathit{Choose};_{send}\ \mathit{Pay}$ from the two distinct subsystems that are composed by the sequential composition operator, since the *Pay* subsystem only starts after the *Choose* subsystem finishes. Therefore, we show the sequential composition operator behavior using the TIOSTS models *Choose* and *Pay* from Figures 3.2 and 3.3.

The *Choose* subsystem allows the user to select its payment method, first receiving the parameter *op* and storing it in the $option_C$ variable. If $option_C$ equals to check, cash or card and $clock_C$ is less than or equal to $10s$, the *performPayment!* output

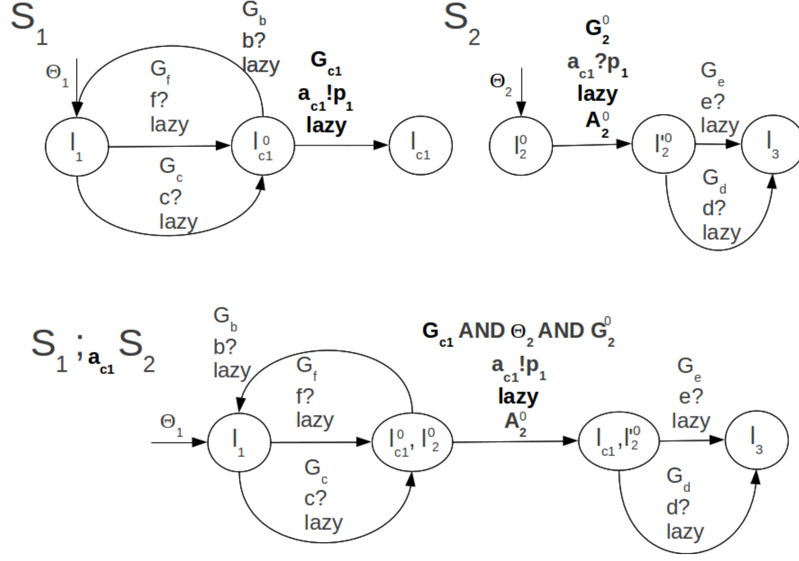


Figure 3.1: Toy example for the sequential composition

action must be executed when the transition fires. Similarly, if $option_C$ equals to *off* and $clock_C$ is less than or equals to $10s$, the *shutdown!* output action must be executed. Alternatively, if the clock value is more than $10s$, the *return!* output action must be executed and the clock resets to zero. Finally, in the S2 location, $option_C$ variable is communicated through the *send* action by using the *op* parameter and the *send!* output action.

The *Pay* subsystem models the user payment method. The payment alternatives are checked according to $option_P$ variable. If $option_P$ is check or cash and the $clock_P$ value is less than or equal to $20s$, the *pay!* output action must be executed. If $option_P$ is equal to card and the $clock_P$ value is less than or equal to $20s$, the parameter *info* is received through the *sign?* input action and the variable *signature* stores its value. Depending on the *signature* value, this subsystem must execute the *pay!* or *discard!* output actions, the last one with clock resetting. Finally, if $option_P$ stores the *off* value, the *finishSystem!* output action must be executed.

Now suppose these subsystems are part of a payment system, but they have been developed and/or are to be deployed separately. In this case, integration testing will be required. For this, we need to compose them both at specification level to allow test case generation. Notice that the *Choose* subsystem produces a value, stored in the $option_C$ variable, that is consumed by the *Pay* system and stored in the $option_P$

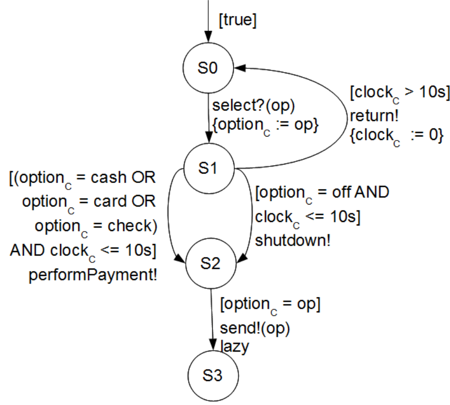


Figure 3.2: *Choose* subsystem

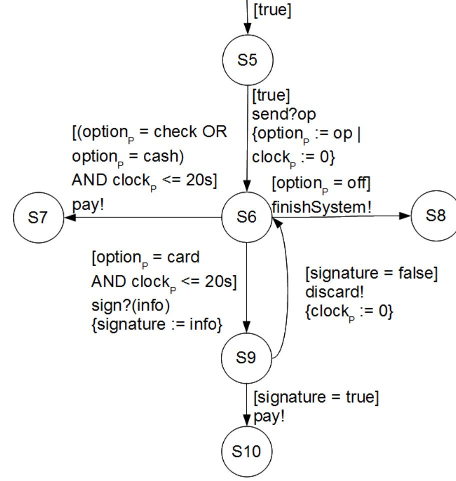


Figure 3.3: *Pay* subsystem

variable. Also this value is produced as output by *Choose* and received as input by *Pay*. In this case, as the execution of the subsystems is inherently sequential, they can be easily composed by the sequential composition operator: $Choose;_{send}\ Pay$ (Figure 3.4).

As a result of the composition, synchronization of the *send!* output action from transition $S2 \xrightarrow{send!op} S3$ to the *send?* input action from $S5 \xrightarrow{send?op} S6$ transition is performed. The new transition $(S2,S5) \xrightarrow{send!op} (S3,S6)$ that joins *Choose* and *Pay* subsystems has its guard built by the conjunction of the incoming transition of the S3 location from *Choose* subsystem, the guard of *Pay* first transition and the initial condition of *Pay* subsystem so that the set of initial values from *Choose* to *Pay* subsystem be preserved in the resulting composition. Moreover, we include the synchronizing *send!* output action to allow this parameter to be communicated (as a testing logging) to other subsystems during the integration testing phase. Also, we maintain assignments of the $S5 \xrightarrow{send?op} S6$ transition in lieu of assignments from the $S2 \xrightarrow{send!op} S3$ transition because there is no need to preserve them after the *op* parameter be communicated to the *Pay* subsystem, since we adopt a normal form that forces the assignments from the $S2 \xrightarrow{send!op} S3$ transition to be empty.

The sequential composition is not applicable to any pair of TIOSTS. A few requirements are needed for both models. To make them clear, we define a normal form that includes information about the final (composition) location of the first model and

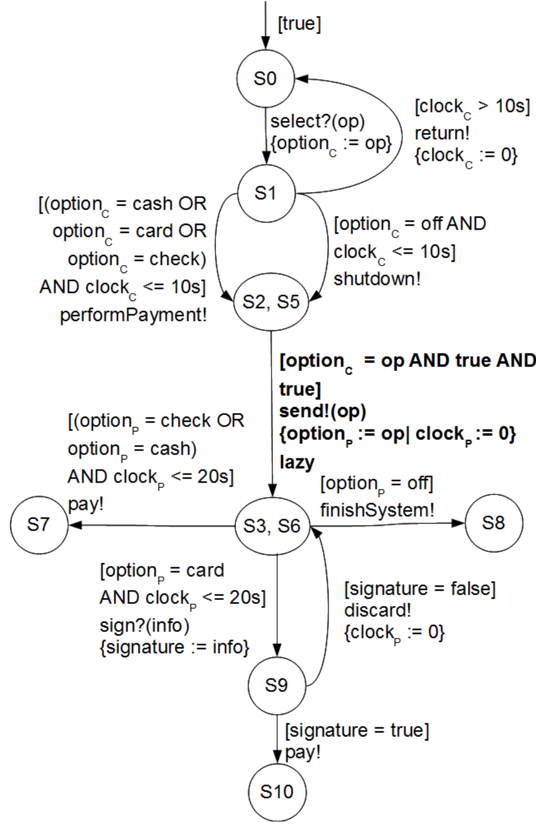


Figure 3.4: The sequential composition $Choose;_{send}\ Pay$

also about the composition (initial) location of the second model. Nevertheless, the sequential operator does not impose any additional test hypotheses on specifications and implementations besides the ones mentioned in Section 2.3.2.2, since this operator preserves the tioco conformance relation (Theorem 2).

The sequential composition $\mathcal{S}_1;_{a_{c1}}\mathcal{S}_2$ can be defined if the following conditions are met by $\mathcal{S}_1$ and $\mathcal{S}_2$. The $\mathcal{S}_1$ model must present: 1) A composition location l_{c1} with a single input transition named t_{c1}; 2) t_{c1} has an output action a_{c1} and no assignments. Moreover, the $\mathcal{S}_2$ model must present: 1) The initial location l_2^0 without input transitions and a single output transition t_2^0; 2) t_2^0 comprises the a_2^0 input action such that the list of parameters from a_{c1} is equal to a_2^0 (a_{c1} and a_2^0 have the same label and opposing output/ input actions); 3) The assignments A_2^0 such that every parameter communicated through the a_{c1} action corresponds to a variable from $\mathcal{S}_2$ and every clock from C_2 is reset (an example is presented in Figure A.1).

Given three locations l_0, l and $l' \in L_{\mathcal{S}_1}$, we consider $l_0 \rightarrow l$ and $l \rightarrow l'$, meaning

that l_0 is the source and l is the target of a transition, as well as l is the source and l' is the target of another transition. In addition, the function $\overline{a?} = a!$ is responsible for returning the opposite action with its label preserved. We formalize the Sequential Composition (SC) Normal Form in Definition 8.

Definition 8 (SC Normal Form)**.** *Let $\mathcal{S}_i = \langle V_i, P_i, \Theta_i, L_i, l_i^0, \Sigma_i, C_i, \mathcal{T}_i \rangle$, with $i \in \{1, 2\}$, be two TIOSTSs. $\mathcal{S}_1$ and $\mathcal{S}_2$ are in the SC normal form if:*

- *$L_1 \cap L_2 = V_1 \cap V_2 = C_1 \cap C_2 = \emptyset$;*
- *$\mathcal{S}_1$ has a special location named l_{c1} with a single incoming transition $t_{c1} = (l_{c1}^0, a_{c1}, G_{c1}, A_{c1}, d_{c1}, l_{c1})$ such that $a_{c1} \in \Sigma_1^!$, $d_{c1} = lazy$, a_{c1} does not appear in any other transition from $\mathcal{S}_1$, $sig(a_{c1}) = \langle p_{c1}^1, p_{c1}^2, \ldots p_{c1}^n \rangle$ is the list of parameters to be communicated to $\mathcal{S}_2$ and $A_{c1} = \emptyset$;*
- *$\mathcal{S}_2$ has the l_2^0 initial location with a single outgoing transition $t_2^0 = (l_2^0, a_2^0, G_2^0, A_2^0, d_2^0, l_2'^0)$, where $a_2^0 = \overline{a_{c1}}$, a_2^0 does not appear in any other transition from $\mathcal{S}_2$, $sig(a_2^0) = \langle p_{c1}^1, p_{c1}^2, \ldots p_{c1}^n \rangle$, $A_2^0 = A_2^{0D} \cup A_2^{0C}$ such that A_2^{0C} has one element for each clock resetting from C_2 and, for each parameter p from $sig(a_2^0)$ and variable $x \in V_2$, there is one element in A_2^{0D} using the format $x := p$.*

The conditions imposed by the normal form do not strictly narrow the operator usage. In practice, they can be met in a pipeline-style system, without shared memory, where the first subsystem produces a single result that is going to be consumed by the second one. For instance, model composition of Java threads defined by the *FutureTask* class is a potential scope of application. The *Future* interface provides a way of blocking a task until the result provided by another one is completed. In this sense, we may have a system where execution of $task_2$ starts only when a result is returned from the execution of $task_1$ [JP14] without no other needed synchronization than passing the result produced from one task to another. Moreover, sequential composition can be found at application level in the Android development platform [Dev14] to integrate an activity that is invoked for a result and a subsequent one consumes it.

From the SC Normal Form, the sequential composition operator can be defined as follows, where given two sets Σ_1 and Σ_2, $\Sigma_1 \backslash \Sigma_2$ returns the set of Σ_1 elements minus the set of Σ_2 elements.

Definition 9 (Sequential Composition). *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two TIOSTS in the SC Normal Form. The sequential composition $\mathcal{S}_1;_{a_{c1}}\mathcal{S}_2 = \langle V_1 \cup V_2, P_1 \cup P_2, \Theta_1, L, l^0, \Sigma, C_1 \cup C_2, \mathcal{T}\rangle$ where $L = L_1 \backslash \{l^0_{c1}, l_{c1}\} \cup L_2 \backslash \{l^0_2, l'^0_2\} \cup \{(l^0_{c1}, l^0_2), (l_{c1}, l'^0_2)\}$, $\Sigma = \Sigma_1 \cup \Sigma_2 \backslash \{a^0_2\}$ and $\mathcal{T}$ is the set of transitions such that:*

$$\mathcal{T}_1 \backslash (\{(l_1, a_1, G_1, A_1, d_1, l^0_{c_1}) | (l_1, a_1, G_1, A_1, d_1, l^0_{c_1}) \in \mathcal{T}_1\} \cup$$
$$\{(l^0_{c_1}, a_1, G_1, A_1, d_1, l_1) | (l^0_{c_1}, a_1, G_1, A_1, d_1, l_1) \in \mathcal{T}_1\} \cup \{t_{c1}\}) \cup \tag{3.1}$$
$$\mathcal{T}_2 \backslash (\{(l'^0_2, a_2, G_2, A_2, d_2, l_2) | (l'^0_2, a_2, G_2, A_2, d_2, l_2) \in \mathcal{T}_2\} \cup$$
$$\{(l_2, a_2, G_2, A_2, d_2, l'^0_2) | (l_2, a_2, G_2, A_2, d_2, l'^0_2) \in \mathcal{T}_2\} \cup \{t^0_2\}) \cup \tag{3.2}$$
$$\{(l_1, a_1, G_1, A_1, d_1, (l^0_{c1}, l^0_2) | (l_1, a_1, G_1, A_1, d_1, l^0_{c1}) \in \mathcal{T}_1\} \cup$$
$$\{((l^0_{c1}, l^0_2), a_1, G_1, A_1, d_1, l_1 | (l^0_{c1}, a_1, G_1, A_1, d_1, l_1) \in \mathcal{T}_1\} \cup \tag{3.3}$$
$$\{((l_{c1}, l'^0_2), a_2, G_2, A_2, d_2, l_2 | (l'^0_2, a_2, G_2, A_2, d_2, l_2) \in \mathcal{T}_2\} \cup$$
$$\{(l_2, a_2, G_2, A_2, d_2, (l_{c1}, l'^0_2) | (l_2, a_2, G_2, A_2, d_2, l'^0_2) \in \mathcal{T}_2\} \cup \tag{3.4}$$
$$\{((l^0_{c1}, l^0_2), a_{c1}, G_{c1} \wedge \Theta_2 \wedge G^0_2, A^0_2, lazy, (l_{c1}, l'^0_2))\} \tag{3.5}$$

The sequential composition combines the union of variables, parameters, locations and actions of TIOSTS. Furthermore, both TIOSTS are composed through the creation of the transition set $\mathcal{T}$, built in five steps during $\mathcal{T}$ definition: (3.1) Add $\mathcal{T}_1$ transitions excepting t_{c1} transition and those that have l^0_{c1} as their source or target location; (3.2) Add $\mathcal{T}_2$ transitions excepting t^0_2 transition and those that have l'^0_2 as their source or target location; (3.3) Add transitions excluded in step (3.1) with location l^0_{c1} replaced by location (l^0_{c1}, l^0_2); (3.4) Add transitions excluded in step (3.2) with location l'^0_2 replaced by location (l_{c1}, l'^0_2); (3.5) Add a new transition with the *lazy* deadline from locations (l^0_{c1}, l^0_2) to (l_{c1}, l'^0_2) that perform the communication between models $\mathcal{S}_1$ and $\mathcal{S}_2$.

In order to make certain under which conditions the sequential composition operator preserves the tioco conformance relation, we have the compositionality result of Theorem 2. Its proof is presented in Appendix A.

Theorem 2 (tioco Sequential Composition). *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be specifications and $\mathcal{I}_1$, $\mathcal{I}_2$ be implementations modeled by TIOSTSs that meet Definition 9. If $\mathcal{I}_1$* tioco *$\mathcal{S}_1 \wedge \mathcal{I}_2$* tioco *$\mathcal{S}_2$ then $\mathcal{I}_1;_{a_{c1}}\mathcal{I}_2$* tioco *$\mathcal{S}_1;_{a_{c1}}\mathcal{S}_2$.*

The sequential operator preserves the tioco conformance relation without requiring that the specification of the subsystems be input complete. This happens because it does not change the outputs after each trace from the two subsystems in the resulting system. In this way, if two subsystem implementations are in conformance to their specifications, the sequential composition of the implementations also will preserve conformance to the sequential composition of their specifications.

Analogously, it is not possible that non conforming subsystems leads to a composed implementation which is tioco conformant to its composed specification. This happens because there is no extra input or output actions which are added by the sequential operator when comparing the composed specification to the composed implementation.

In practice, we cannot always use that result when only a few tests have been performed to check I_1 against S_1 and I_2 against S_2. Therefore, it is crucial to develop a strategy to test the composition of I_1 and I_2. This is discussed in Section 3.4.

3.2 Interruption Composition

When composing subsystems by interruptions, one subsystem may interrupt the execution of another when shared resources are instantly required. For example, consider a phone call that arrives when the user is editing a document on a smartphone. This is an application level interruption and the effect is that the call subsystem is brought forward, sending the edition subsystem to the background.

After an interruption handling, the interrupted subsystem may or may not resume its execution [ZHHL11]. When the interrupted subsystem resumes its execution, from the point where the interruption occurred, we call it *weak interruption*. Otherwise, if execution finishes by occurrence of the interruption, we call it *strong interruption*. The former is suitable for dealing with resource sharing such as foreground execution, whereas the latter is suitable for recovery procedures where the interrupted system cannot proceed. In this work, we focus on weak interruptions only.

We use weak interruption composition for a situation where a subsystem interrupts another before it finishes. Additionally, the interrupted subsystem resumes after the execution of the interrupter subsystem. For this, we consider as possible points of

interruption the ones in which the systems can communicate by synchronizing actions that may carry parameters.

For example, consider the *Mouse* subsystem presented in Figure 3.5 as the interrupted subsystem and the *Reset* subsystem presented in Figure 3.6 as the interrupter subsystem. Also consider the *send* and *receive* actions that are present in both models either as input or output actions (the corresponding transitions are marked as dotted lines in order to be highlighted).

Figure 3.5 shows an example of TIOSTS through the *Mouse* subsystem, which is responsible for defining a simple mouse behavior. The *true* guard enables this subsystem, and the S0 location starts it, allowing the reception of the *click?* input action with no urgency and storing its value in the coordinate_M variable. From location S1, the system behavior is twofold: it can execute the *simpleClick!* output action after more than $2s$ and reset its clock with the eager deadline or execute the *click?* input action within $2s$ and no urgency, storing the pointer position in the coordinate_M variable. From location S2, the $coordinate_M$ variable value is communicated to other subsystems in the environment through the *doubleClick!* output action with delayable deadline, also forcing parameter *position* to assume a value equal to the $coordinate_M$ variable value (see guard $coordinate_M = position$). Also, the clock is reset and the system goes to the S0 location. Finally, the subsystem has location S3 with two transitions: i) an incoming transition that executes the *send!* output action and communicates the pointer position to other subsystems with lazy deadline; ii) an outgoing transition that contains the *receive?* input action and stores the pointer position in the $coordinate_M$ variable.

The *Reset* subsystem is responsible for resetting the *Mouse* subsystem state (Figure 3.6). It starts by receiving the current cursor position through the *send?* input action and stores it in the $coordinate_I$ variable. Depending on the $coordinate_I$ variable value, the system resets the hardware through the *resetHardware!* output action or it resets the software by using the *resetSoftware!* output action. Finally, the *receive!* output action emits the $coordinate_I$ value and indicates that the *Reset* subsystem is finished.

In Figure 3.7, the interruption composition $Mouse\ {}_{\text{send}}\triangle_{\text{receive}}\ Reset$ starts with the *Mouse* behavior, but the *send!* output action discontinues it, beginning the execution

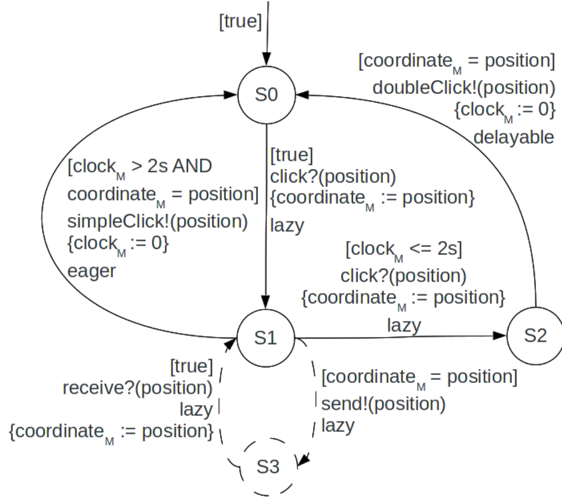


Figure 3.5: *Mouse* subsystem

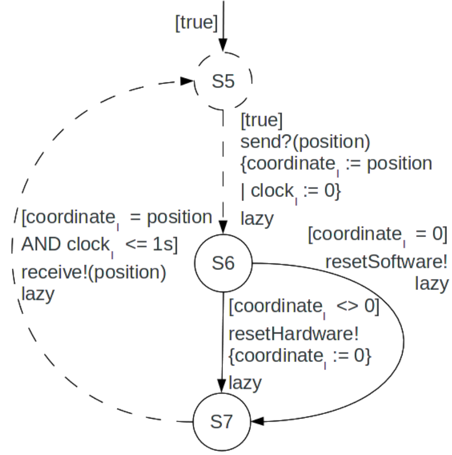


Figure 3.6: *Reset* subsystem

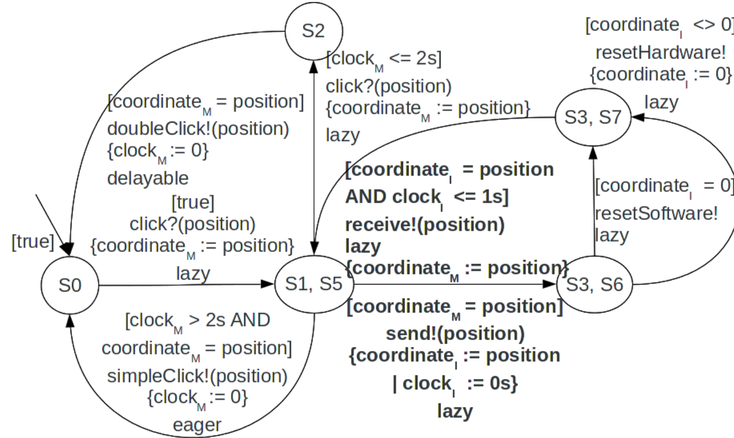


Figure 3.7: Interruption composition for the *Mouse* and *Reset* subsystems

of the *Reset* subsystem before the *Mouse* subsystem is finished. The system execution flow returns to the *Mouse* subsystem when the *receive* output action is executed and the $coordinate_I$ variable is reset.

We perform synchronizations of the *send* and the *receive* actions. This happens because there is a need to communicate the *Mouse* subsystem values at an interruption point delimited by the *send* action. After that, the *Reset* subsystem is executed until it finishes and the changed values are communicated back to the first subsystem through the *receive* action. From this point, the system follows its behavior.

From the example, we notice that the weak interruption operator requires the models to follow a pattern. First, the operator needs two synchronizing actions in each

model. Second, the interrupter system resets clock variables, before it starts, to preserve the same behavior specified before the composition. Therefore, it is mandatory to define a normal form for the interrupted and interrupter subsystems.

To introduce the normal form and the interruption operator, suppose that two TIOSTS models $\mathcal{S}_1$ and $\mathcal{S}_2$ become part of the interruption composition $\mathcal{S}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{S}_2$ (Figure A.2). In addition, we use the $\overline{a?} = a!$ operator to define the opposite action of $a?$ and its label remains the same. TIOSTS $\mathcal{S}_1$ represents the interrupted subsystem and it has the location l'_{c1} with a single incoming transition t_{c1} that comprises the a_{c1} output action, lazy deadline and parameters p_1. Additionally, l'_{c1} has a single outgoing transition t'_{c1} with lazy deadline, the $\overline{a_{c2}}$ input action, parameters p_2, and assignments A'_{c1} that store values of these parameters.

TIOSTS $\mathcal{S}_2$ represents the interrupter subsystem and it has the special location l_2^0 with the single outgoing transition t_2^0 composed by the $\overline{a_{c1}}$ input action that receives parameters p_1 from model $\mathcal{S}_1$, storing these values in the A_2^0 assignments. Also, location l_2^0 has a single incoming transition t_{c2} that comprises the a_{c2} output action with p_2 parameters and deadline lazy.

The weak interruption composition $\mathcal{S}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{S}_2$ performs the synchronization on the a_{c1} and a_{c2} actions, allowing $\mathcal{S}_1$ to be interrupted by $\mathcal{S}_2$. More specifically, the linking of the subsystems is twofold. The operator adds the $(l_{c1}, l_2^0) \xrightarrow{\mathrm{a_{c1}!p_1}} (l'_{c1}, l_2'^0)$ synchronizing transition that has the *lazy* deadline and the output action $a_{c1}!$ as an observation point for testing purposes. This transition contains the guard G_{c1} AND Θ_2 AND G_2^0 to maintain subsystems conditions. Also, we add clock resetting to the A_2^0 assignment, since we want to maintain $\mathcal{S}_2$ clock behavior after the beginning of the interruption. A similar process happens to the $(l'_{c1}, l_{c2}) \xrightarrow{\mathrm{a_{c2}!p_2}} (l_{c1}, l_2^0)$ transition.

In summary, WIC Normal Form requires that the $\mathcal{S}_1$ subsystem presents: 1) A composition location l'_{c1}; 2) A single incoming transition to l'_{c1} called $t_{c1} = (l_{c1}, a_{c1}, G_{c1}, A_{c1}, d_{c1}, l'_{c1})$ with an output action a_{c1} and no assignments; 3) A single outgoing transition to l'_{c1} called $t'_{c1} = (l'_{c1}, \overline{a_{c2}}, G'_{c1}, A'_{c1},\ d'_{c1}, l_{c1})$ with an input action $\overline{a_{c2}}$ and assignments that correspond variables to parameters sent to $\mathcal{S}_2$. Subsystem $\mathcal{S}_2$ presents: 4) Two composition locations l_2^0 and l_{c2}; 5) A single outgoing transition from l_2^0 called $t_2^0 = (l_2^0, \overline{a_{c1}}, G_2^0, A_2^0, d_2^0, l_2'^0)$ with an input action $\overline{a_{c1}}$ and assignments that correspond

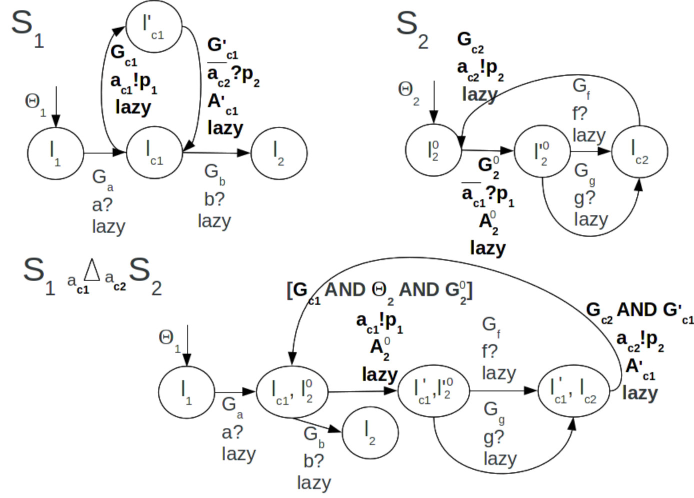


Figure 3.8: Example of weak interruption composition

variables to parameters sent to $\mathcal{S}_1$; 6) A single outgoing transition from l_{c2} called $t_{c2} = (l_{c2}, a_{c2}, G_{c2}, A_{c2}, d_{c2}, l_2^0)$ with an output action a_{c2} and no assignments. We force A_{c1} and A_{c2} assignments to be empty because the output actions associated to these transitions transmit values previously stored in the subsystem variables. We formalize the Weak Interruption Composition (WIC) Normal Form in Definition 10.

Definition 10. *(WIC Normal Form) Let* $i \in \{1, 2\}$ *and* $\mathcal{S}_i = \langle V_i, P_i, \Theta_i, L_i, l_i^0, \Sigma_i, C_i, \mathcal{T}_i \rangle$ *be two TIOSTS.* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *are in the WIC Normal Form if the following conditions are met:*

- $L_1 \cap L_2 = V_1 \cap V_2 = C_1 \cap C_2 = \emptyset$;

- $\mathcal{S}_1$ *must have: i) Two special locations* l_{c1} *and* l'_{c1}*; ii) The* l'_{c1} *location must have a single incoming transition* $t_{c1} = (l_{c1}, a_{c1}, G_{c1}, A_{c1}, d_{c1}, l'_{c1})$ *such that* $a_{c1} \in \Sigma_1^!$, $d_{c1} = lazy$, a_{c1} *does not appear in any other transition from* $\mathcal{S}_1$, $sig(a_{c1}) = \langle p_{c1}^1, p_{c1}^2, \ldots p_{c1}^n \rangle$ *is the list of parameters to be communicated to* $\mathcal{S}_2$ *and* $A_{c1} = \emptyset$*; iii) The* l'_{c1} *location must have a single outgoing transition* $t'_{c1} = (l'_{c1}, a'_{c1}, G'_{c1}, A'_{c1}, d'_{c1}, l_{c1})$ *such that* $a'_{c1} \in \Sigma_1^?$, $d'_{c1} = lazy$, a'_{c1} *does not appear in any other transition from* $\mathcal{S}_1$, $sig(a'_{c1}) = \langle p'^1_{c1}, p'^2_{c1}, \ldots p'^n_{c1} \rangle$ *is the list of parameters to be received from* $\mathcal{S}_2$ *and* $A'_{c1} = A'^C_{c1} \cup A'^D_{c1}$ *such that there is no imposed condition to* A'^C_{c1} *assignment and there is one element in* A'^D_{c1} *using the format* $x := p$ *for each parameter* p *from* $sig(a'_{c1})$ *and variable* $x \in V_1$.

- *$\mathcal{S}_2$ must have: i) Two special locations l_2^0 and l_{c2}; ii) The l_2^0 initial location has a single outgoing transition $t_2^0 = (l_2^0, a_2^0, G_2^0, A_2^0, d_2^0, l_2'^0)$, where , $d_2^0 = lazy$, $a_2^0 \in \Sigma_2^?$, $a_2^0 = \overline{a_{c1}}$, a_2^0 does not appear in any other transition from $\mathcal{S}_2$, $sig(a_2^0) = \langle p_{c1}^1, p_{c1}^2, \ldots p_{c1}^n \rangle$, $A_2^0 = A_2^{0D} \cup A_2^{0C}$ such that A_2^{0C} has one element for each clock resetting from C_2 and, for each parameter p from $sig(a_2^0)$ and variable $x \in V_2$, there is one element in A_2^{0D} using the format $x := p$; iii) The l_{c2} location must have a single outgoing transition $t_{c2} = (l_{c2}, a_{c2}, G_{c2}, A_{c2}, d_{c2}, l_2^0)$ such that $d_{c2} = lazy$, $a_{c2} \in \Sigma_2^!$, $a_{c2} = \overline{a'_{c1}}$, a_{c2} does not appear in any other transition from $\mathcal{S}_2$, $sig(a_{c2}) = \langle p_{c1}'^1, p_{c1}'^2, \ldots p_{c1}'^n \rangle$ is the list of parameters to be received from $\mathcal{S}_2$ and $A_{c2} = \emptyset$.* ◇

Besides requiring a normal form, the weak interruption composition operator uses the *sync* operation, responsible for defining the set of transitions from the composed model that do not synchronize. As input, this operation receives: i) the set of transitions $\mathcal{T}_1$ from $\mathcal{S}_1$; ii) the set of transitions $\mathcal{T}_2$ from $\mathcal{S}_2$; and iii) the set L_{sync} of synchronization locations. For example, the $\mathcal{S}_1 {}_{a_{c1}}\triangle_{a_{c2}} \mathcal{S}_2$ system from Figure A.2 comprises the (l_{c1}, l_2^0), $(l'_{c1}, l_2'^0)$ and (l'_{c1}, l_{c2}) for L_{sync} locations.

Basically, the *sync* operation is performed in two steps. First, we add each transition from $\mathcal{S}_1$ and $\mathcal{S}_2$ subsystems, excepting those that contain elements from L_{sync}. Since each transition contains a source or a target location that belongs to L_{sync}, no element is added to the set of transitions from the composed model. Figure 3.9 shows this step result. We use dotted lines to highlight transitions from subsystems excluded by this step.

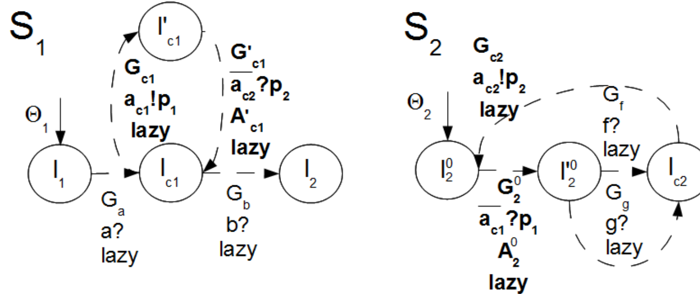


Figure 3.9: Step 1 of the *sync* operator

The second step adds transitions in the composed model whose source or target

locations do not belong to L_{sync}. Consequently, transitions $l_1 \xrightarrow{a?} (l_{c1}, l_2^0)$ and $(l_{c1}, l_2^0) \xrightarrow{b?} l_2$ fill these requirements. Figure 3.10 shows the result for this step. Although we do not add transitions whose source or target locations are $(l'_{c1}, l_2'^0)$ or (l'_{c1}, l_{c2}), we leave them to highlight that they do not take part in any transition from the composed model.

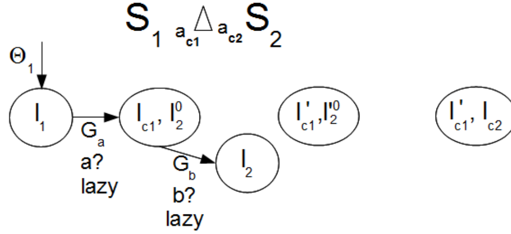


Figure 3.10: Step 2 of the *sync* operator

We present the formal rules of the *sync* operator in Definition 11. Step 1 comprises rules (3.6) and (3.7), while rules (3.8) and (3.9) from Definition 11 compose step 2. For the sake of simplification, we do not show elements $(l_{s1}, l_{s2}) \in L_{sync}$ in the transitions rule of the *sync* operator, but we restrict them in the beginning of this definition.

Also, given two sets Σ_1 and Σ_2, we use the notation $\Sigma_1 \backslash \Sigma_2$ to represent the relative complement of Σ_2 in Σ_1, or Σ_2 - Σ_1. For the sake of simplicity, we relax pertinence rules from set theory [Hal60; Jec78] by establishing conditions on the tuple components that compose a transition element outside the pertinence rule. In addition, we make explicit from which set an element belongs to. For example, if we want to represent the set of elements of the form $(l_1, a_1, G_1, A_1, d_1, l'_1)$ from Σ_1 such that $l_1 = l_{s1}$, instead of using $\Sigma_1 \backslash \{(l_1, a_1, G_1, A_1, d_1, l'_1) | (l_1, a_1, G_1, A_1, d_1, l'_1) \in \Sigma_1 \wedge l_1 = l_{s1}\}$, we present restrictions on l_{s1} and use $\Sigma_1 \backslash \{(l_1, a_1, G_1, A_1, d_1, l_{s1}) | (l_1, a_1, G_1, A_1, d_1, l_{s1}) \in \Sigma_1\}$.

Definition 11. *(sync Operation) Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two different sets of TIOSTS transitions and L_{sync} be a set of locations of the form (l_{s1}, l_{s2}) where $l_{s1} \in L_1$ and $l_{s2} \in L_2$. Assuming that $(l_{s1}, l_{s2}) \in L_{sync}$ and $l_1, l_2 \notin L_{sync}$, the operation $sync(\mathcal{T}_1, \mathcal{T}_2, L_{sync})$ returns the set obtained from the rule:*

$$
\begin{aligned}
&\mathcal{T}_1 \backslash (\{(l_1, a_1, G_1, A_1, d_1, l_{s1}) | (l_1, a_1, G_1, A_1, d_1, l_{s1}) \in \mathcal{T}_1\} \cup \\
&\{(l_{s1}, a_1, G_1, A_1, d_1, l_1) | (l_{s1}, a_1, G_1, A_1, d_1, l_1) \in \mathcal{T}_1\} \cup \\
&\{(l_{s1}, a_{s1}, G_{s1}, A_{s1}, d_{s1}, l'_{s1})\}) \cup && (3.6) \\
&\mathcal{T}_2 \backslash (\{(l_{s2}, a_2, G_2, A_2, d_2, l_2) | (l_{s2}, a_2, G_2, A_2, d_2, l_2) \in \mathcal{T}_2\} \cup \\
&\{(l_2, a_2, G_2, A_2, d_2, l_{s2}) | (l_2, a_2, G_2, A_2, d_2, l_{s2}) \in \mathcal{T}_2\} \cup \\
&\{(l_{s2}, a_{s2}, G_{s2}, A_{s2}, d_{s2}, l'_{s2})\}) \cup && (3.7) \\
&\{(l_1, a_1, G_1, A_1, d_1, (l_{s1}, l_{s2})) | (l_1, a_1, G_1, A_1, d_1, l_{s1}) \in \mathcal{T}_1\} \cup \\
&\{((l_{s1}, l_{s2}), a_1, G_1, A_1, d_1, l_1) | (l_{s1}, a_1, G_1, A_1, d_1, l_1) \in \mathcal{T}_1\} \cup && (3.8) \\
&\{(l_2, a_2, G_2, A_2, d_2, (l_{s1}, l_{s2})) | (l_2, a_2, G_2, A_2, d_2, l_{s2}) \in \mathcal{T}_2\} \cup \\
&\{((l_{s1}, l_{s2}), a_2, G_2, A_2, d_2, l_2) | (l_{s2}, a_2, G_2, A_2, d_2, l_2) \in \mathcal{T}_2\} && (3.9)
\end{aligned}
$$

$\diamond$

For subsystems $\mathcal{S}_1$ and $\mathcal{S}_2$ from Figure A.2, the sets of synchronization locations are $\{l_{c1}, l'_{c1}\}$, $\{l^0_2, l'^0_2, l_{c2}\}$, respectively. In this case, $sync(\mathcal{T}_1, \mathcal{T}_2, \{(l_{c1}, l^0_2), (l'_{c1}, l'^0_2), (l'_{c1}, l_{c2})\})$ returns: (3.6) $\mathcal{T}_1$ transitions, excluding those that comprise $l_{s1} \in \{l_{c1}, l'_{c1}\}$ as their source or target location and the $a_{c1}!$ and $\overline{a_{c2}}?$ actions; (3.7) $\mathcal{T}_2$ transitions, excluding those that have $l_{s2} \in \{l^0_2, l'^0_2, l_{c2}\}$ as their source or target location and $\overline{a_{c1}}?$ and $a_{c2}!$ actions; (3.8) transitions eliminated in step (3.6) with location l_{s1} replaced by location (l_{s1}, l_{s2}); (3.9) transitions excluded in step (3.7) with location l_{s2} replaced by location (l_{s1}, l_{s2}). In summary, the resulting set of transitions contains transitions with their locations updated by the L_{sync} set of locations, when applicable.

We apply the weak interruption composition operator to TIOSTS models $\mathcal{S}_i$, where $\mathcal{S}_i = \langle V_i, P_i, \Theta_i, L_i, l^0_i, \Sigma_i, C_i, \mathcal{T}_i \rangle$, $\Sigma_i = \Sigma^?_i \cup \Sigma^!_i$ and i = 1, 2. Definition 12 presents the formal rules for the weak interruption composition operator by using elements presented in Figure A.2.

Definition 12. *(Weak Interruption Composition) Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two TIOSTS in the WIC Normal Form. The weak interruption composition ${\mathcal{S}_1}_{\mathrm{a_{c1}}} \triangle_{\mathrm{a_{c2}}} \mathcal{S}_2$ is defined by $\langle V_1 \cup V_2, P_1 \cup P_2, \Theta_1, L, l^0, \Sigma, C_1 \cup C_2, \mathcal{T} \rangle$ where $L = L_1 \backslash \{l_{c1}, l'_{c1}\} \cup L_2 \backslash \{l^0_2, l'^0_2, l_{c2}\} \cup \{(l_{c1}, l^0_2), (l'_{c1}, l'^0_2), (l'_{c1}, l_{c2})\}$, $\Sigma = \Sigma_1 \cup \Sigma_2 \backslash \{a^0_2, a'_{c1}\}$ and $\mathcal{T}$ is the set of transitions such that:*

$$sync(\mathcal{T}_1, \mathcal{T}_2, \{(l_{c1}, l_2^0), (l'_{c1}, l_2'^0), (l'_{c1}, l_{c2})\}) \cup \tag{3.10}$$

$$\{((l_{c1}, l_2^0), a_{c1}, G_{c1} \wedge \Theta_2 \wedge G_2^0, A_2^0, lazy, (l'_{c1}, l_2'^0))\} \cup \tag{3.11}$$

$$\{((l'_{c1}, l_{c2}), a_{c2}, G_{c2} \wedge G'_{c1}, A'_{c1}, lazy, (l_{c1}, l_2^0))\} \cup \tag{3.12}$$

$$\{((l'_{c1}, l_2'^0), a_2, G_2, A_2, d_2, (l'_{c1}, l_{c2})) | (l_2'^0, a_2, G_2, A_2, d_2, l_{c2}) \in \mathcal{T}_2)\} \tag{3.13}$$

◇

The weak interruption composition does not allow intersections between the sets of locations, variables and clocks. We exclude actions a_2^0 and a'_{c1} from the set of actions of the composed model because we replace them by actions a_{c1} and a_{c2} during synchronization. Besides, the resulting model excludes locations which belong to the synchronization process. In summary, the transition set $\mathcal{T}$ includes: (3.10) transitions returned by the *sync* operation that receives the synchronizable action set and the transition set from $\mathcal{S}_1$ and $\mathcal{S}_2$, replacing the excluded locations by the new ones of the composed model (Definition 11); ((3.11) and (3.12)) two new transitions that link the isolated models by performing synchronizations in the composed model and preserving guard requirements through the conjunctions of transition guards from $\mathcal{S}_1$ and $\mathcal{S}_2$; (3.13) transitions from $\mathcal{S}_2$ that are between the new composed locations and do not synchronize. Figure 3.11 uses the $\mathcal{S}_1\,{}_{a_{c1}}\triangle_{a_{c2}}\mathcal{S}_2$ composed system from Figure A.2 to exemplify results for each step. Transitions associated to each step are grouped by color and line style.

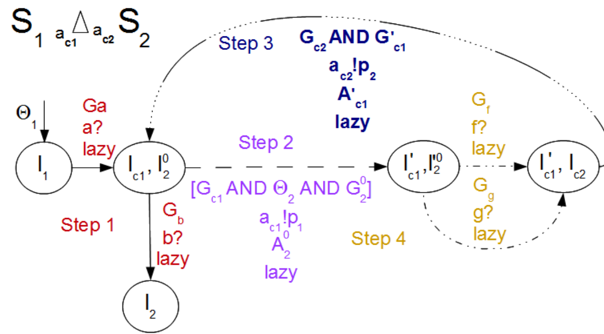


Figure 3.11: Steps for the weak interruption composition

Theorem 3 shows that the weak interruption composition preserves the tioco conformance relation from the subsystems to the composition result. The proof is presented

in Appendix A.

Theorem 3 (tioco weak interruption Composition). *Let $\mathcal{I}_1$, $\mathcal{I}_2$, $\mathcal{S}_1$ and $\mathcal{S}_2$ be four subsystems. If $\mathcal{I}_1$ tioco $\mathcal{S}_1$ and $\mathcal{I}_2$ tioco $\mathcal{S}_2$ then ${\mathcal{I}_1}_{\mathrm{ac1}}\triangle_{\mathrm{ac2}}\mathcal{I}_2$ tioco ${\mathcal{S}_1}_{\mathrm{ac1}}\triangle_{\mathrm{ac2}}\mathcal{S}_2$.*

The interruption operator preserves the tioco conformance relation from subsystems to the composed systems. This happens because it does not change outputs after each trace from the two subsystems in the resulting system. In this way, if two subsystem implementations are in conformance to their specifications, the interruption composition of the implementations also will preserve conformance to the interruption composition of their specifications.

Apart from tioco conformance, a test case generation process can use the composed model to generate interruption test cases. For this, we can follow the standard process and tool for TIOSTS presented in Chapter 2 having as input the composed TIOSTS and a choice of test purpose that covers the interruption scenario of interest. Since the process focuses on test purposes, it must comprise actions that belong to: i) the synchronization set of action from the isolated models; and ii) a finishing action from the first model. For example, the *send* and *receive* actions belong to the synchronization set of the $Mouse\ _{\mathrm{send}}\triangle_{\mathrm{receive}}\ Reset$ system. Additionally, the *simpleClick* indicates that the first subsystem finishes a path.

Despite the need to meet the WIC Normal Form, the interruption composition has a wide range of applications in real-time systems. Frequently, an executing subsystem interrupts another and the execution returns to the initial system, which describes the behavior of the weak interruption operator. For example, the Android Platform[Dev14] contains the Toast and Notification classes that provide notifications on an operation executed in the meanwhile. When a user is doing some work on the internet while a message warning appears, the developer can use the Toast Class to implement this behavior. If the user clicks on the notification, the *Message* application resumes the browser application.

3.3 Parallel Composition

Inspired by the message passing model of concurrency that can be found in specification formalisms such as CSP and programming languages such as Erlang, Scala and Go, we define a parallel composition operator. This operator can be applied to define a system that is composed by two communicating subsystems, without shared memory, that may either execute independently or may communicate to each other by using messages, defined as parameterized shared actions with the same label, but opposing input or output types. Also, during the synchronization process, an input action can be communicated to a single output action and vice-versa.

Figure A.3 shows a simple example for the parallel composition operator. TIOSTSs $\mathcal{S}_1$ and $\mathcal{S}_2$ synchronize on actions a and b, resulting in a conjunction of their original guards and output actions. However, they only synchronize if they are available in both models and in a complementary way (they have the same label with conjugated input/output types). For example, the first action of $\mathcal{S}_1$ is c and the first action of $\mathcal{S}_2$ is the synchronizing action $b?$, so c remains as the first action of the resulting system. Next, since the $\mathcal{S}_1$ subsystem performs the $b!$ output action and the $\mathcal{S}_2$ subsystem performs the $b?$ input action, $(l_1'^0, l_2^0) \xrightarrow{\text{b!}} (l_1^0, l_2'^0)$ transition from the resulting system shows a conjunction of guards. Similarly, the $(l_1^0, l_2'^0) \xrightarrow{\text{c}} (l_1'^0, l_2'^0)$ transition composes the resulting system because the independent action c and the synchronizing action $a?$ are the subsystem current actions. Independent actions can be executed in any order, so two possible paths can be executed between (l_1^0, l_3) and $(l_1'^0, l_2'^0)$ locations, differing on the order of c and f actions.

The parallel composition operator that is defined below comprises the function $op(x, y)$, which receives two deadlines and returns the most urgent between them, using the lazy < delayable < eager order. For instance, if x and y assume the delayable and eager values, the function returns eager. Also, we consider that $a!$ is the output action of a.

Definition 13 (Parallel Composition)**.** *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two TIOSTS and suppose that $L_1 \cap L_2 = C_1 \cap C_2 = V_1 \cap V_2 = \Sigma_1^? \cap \Sigma_2^? = \Sigma_1^! \cap \Sigma_2^! = \emptyset$. We define $\mathcal{S}_1 \parallel \mathcal{S}_2 = \langle V_1 \cup V_2; P_1 \cup P_2; \Theta_1 \wedge \Theta_2; L_1 \times L_2; (l_1^0, l_2^0); \Sigma; C_1 \cup C_2; \mathcal{T}\rangle$ where $\Sigma = \Sigma_1^! \cup \Sigma_2^! \cup (\Sigma_1^? \setminus \Sigma_2^!$*

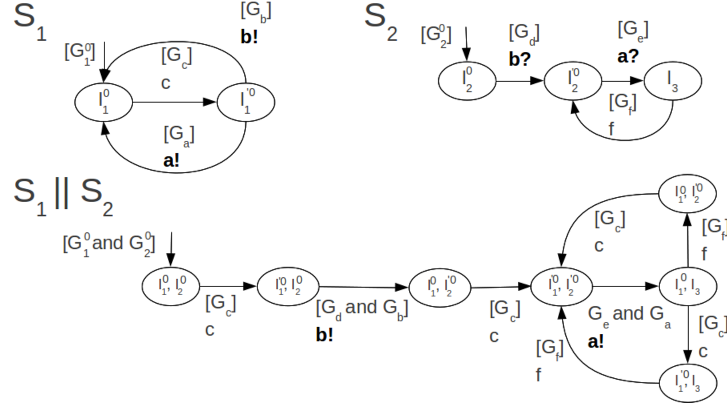


Figure 3.12: Toy example for the parallel composition

$) \cup (\Sigma^?_2 \backslash \Sigma^!_1)$ *and* $\mathcal{T}$ *is the set such that:*

$$For\ a \notin \Sigma_2: \tag{3.14}$$

$$If (l_1, a, G_1, A_1, d_1, l'_1) \in \mathcal{T}_1\ then$$

$$((l_1, l_2), a, G_1, A_1, d_1, (l'_1, l_2)) \in \mathcal{T}$$

$$For\ a \notin \Sigma_1: \tag{3.15}$$

$$If (l_2, a, G_2, A_2, d_2, l'_2) \in \mathcal{T}_2\ then$$

$$((l_1, l_2), a, G_2, A_2, d_2, (l_1, l'_2)) \in \mathcal{T}$$

$$For\ (a \in \Sigma^?_1 \cap \Sigma^!_2) \vee (a \in \Sigma^!_1 \cap \Sigma^?_2): \tag{3.16}$$

$$If (l_1, a, G_1, A_1, d_1, l'_1) \in \mathcal{T}_1 \wedge (l_2, a, G_2, A_2, d_2, l'_2) \in \mathcal{T}_2\ then$$

$$((l_1, l_2), a!, G_1 \wedge G_2, A_1 \cup A_2, op(d_1, d_2), (l'_1, l'_2)) \in \mathcal{T}$$

We do not constrain parameters because their scope are local to transitions in which they were used. Additionally, (3.14) and (3.15) are similar since they include no synchronizable actions in the resulting system, making them interleaved. On the other hand, (3.16) replaces synchronizing actions by output actions in their resulting transitions because we wanted to preserve their communication to other possible subsystems. To reflect this decision on the resulting input and output action set, we exclude input actions that belong to the synchronizing set of both subsystems.

Although we restrict locations, clocks and variables to have distinct sets, this restriction brings no prejudice to practical application of the parallel operator. Since the input action from one subsystem is synchronized to the output action from another

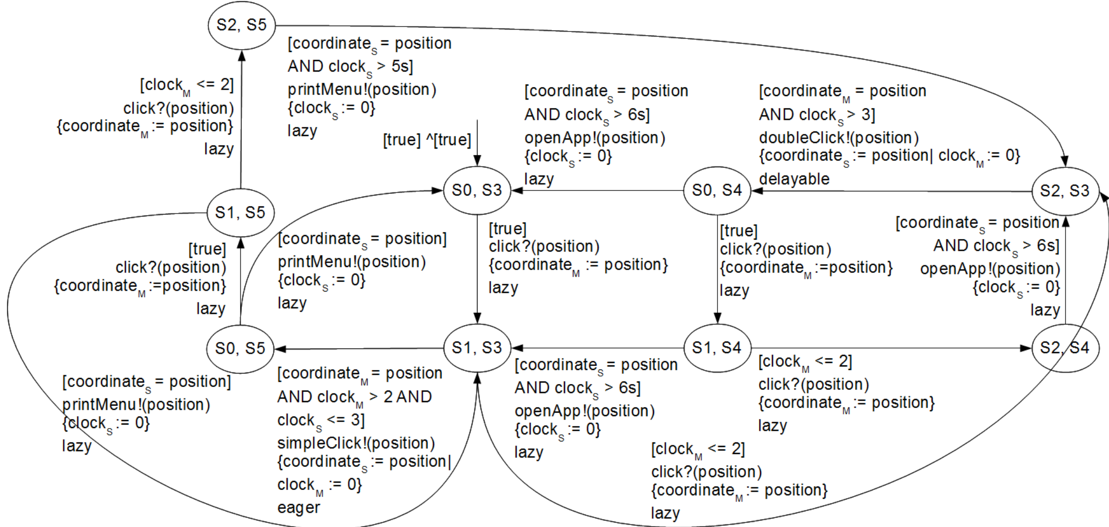


Figure 3.13: Parallel composition for *Mouse* and *Screen* specifications

one, we communicate variable values from one process to another by using the message passing paradigm. Besides, the change of location and clock names in a model does not affect its behavior.

We use the *Mouse* and *Screen* subsystems shown in Figures 2.9 and 2.10 to introduce a more complex example for the parallel composition operator. Figure 3.13 shows the composition for *Mouse* || *Screen* system. They synchronize on *simpleClick* and *doubleClick* actions, imposing a conjunction of the original transitions guards and a resulting output action. Actions such as *click?* from *Mouse* subsystem and *printMenu!* from *Screen* subsystem can be executed independently.

According to Krichen[KT06], the tioco conformance relation has some limitations regarding the parallel composition. So, if we compose one specification to another one in order to form a broader system specification, and their corresponding implementations are tioco conformant to their specifications, the resulting implementation composition is not guaranteed to be conformant according to the tioco theory. Consequently, the parallel composition operator does not always preserve the tioco conformance relation.

The non-conformance problem for the parallelism operator happens because the relation allows the underspecification of input actions, that is, it affords omission of input actions for a specification. In order to solve this issue, subsystems $\mathcal{S}_1$ and $\mathcal{S}_2$ must be input-complete, that is, every location has a transition that enables the ex-

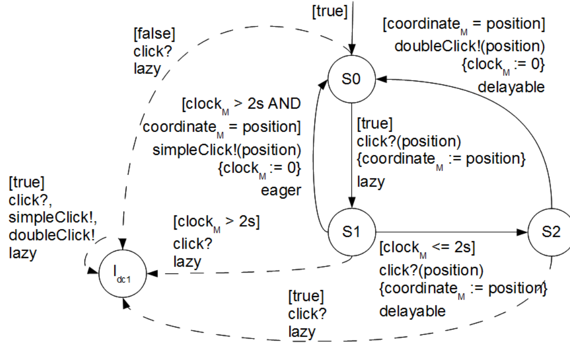


Figure 3.14: Input-complete *Mouse* spec

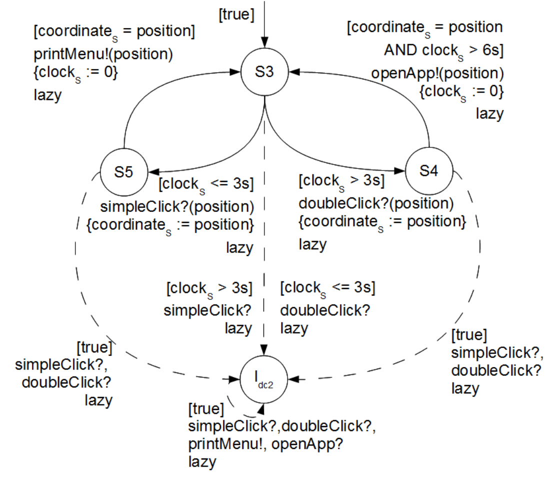


Figure 3.15: Input-complete *Screen* spec

ecution of every input action from the set of input actions. To make compositional subsystems input-complete and allow the testing of the isolated subsystems correct for their composition result, we propose an input-completion process based on [KT06].

Figures 3.14 and 3.15 show the *Mouse* and *Screen* input-complete specifications. We represent transitions added by the input-completion process with dotted arrows. Also, for the sake of simplification, transitions that contain more than one action correspond to single transitions for each action. Basically, we add transitions starting from each location to communicate each absent input action from each specification, leading each transition to the l_{dc} "don't care" location. Additionally, we add transitions with every input and output action to the l_{dc} location to conserve the chaotic behavior of the tioco conformance relation.

For example, the set of input actions from the *Mouse* subsystem is composed by the *click?* input action. Hence, each location from the input-complete *Mouse* specification must have an outgoing transition that contains the *click?* input action and leads this subsystem execution to location l_{dc1}. The S1 location has the outgoing transition S1 $\xrightarrow{\text{click?position}}$ S2 that contains the *click?* input action with the $clock_M <= 2s$ guard. To perform the input-completion process in this location, we need to insert the S1 $\xrightarrow{\text{click?}}$ l_{dc1} transition that contains the *click?* input actions with the $clock_M > 2s$ guard.

We formalize the input-complete system process in Definition 14. Following this input-completion process, there will be no extra input allowed by tioco conformance

relation and consequently no unpredicted output action can be generated in the implementation composition result when compared to the specification composition result.

Definition 14 (Input-completion)**.** *Let $\mathcal{S} = \langle V, P, \Theta, L, l_0, \Sigma, C, \mathcal{T} \rangle$ be a TIOSTS and $l_{dc} \notin L$. We define $IC(\mathcal{S}) = \langle V, P, \Theta, L_{IC}, l_0, \Sigma, C, \mathcal{T}_{IC} \rangle$, where $L_{IC} = L \cup \{l_{dc}\}$ and $\mathcal{T}_{IC} = \mathcal{T} \cup \{(l_{dc}, a, true, \emptyset, lazy, l_{dc}) \mid a \in \Sigma\} \cup \{(l, a, \neg G, \emptyset, lazy, l_{dc}) \mid a \in \Sigma^? \wedge l \in L\}$ such that for each $l \in L$ and each $a \in \Sigma^?$, $G = G_1 \wedge G_2 \wedge \ldots \wedge G_i$, where $G_1 \wedge G_2 \wedge \ldots \wedge G_i$ are the guards of the outgoing edges of l labeled with a. If there is no edge from l labelled with a, G assumes the $false$ value.*

Therefore, let $\mathcal{S}_1$, $\mathcal{I}_1$, $\mathcal{S}_2$ and $\mathcal{I}_2$ be four TIOSTSs such that for $i = 1, 2$, $\mathcal{S}_i$ and $\mathcal{I}_i$ meet conditions established in Definition 13. Thus, we have the compositionality result of Theorem 4, whose proof is presented Appendix A.

Theorem 4 (tioco Parallel Composition)**.** *Let specifications $\mathcal{S}_1$, $\mathcal{S}_2$ and implementations $\mathcal{I}_1$, $\mathcal{I}_2$ be input-complete TIOSTS models. Also $\Sigma_{S_1} = \Sigma_{I_1}$ and $\Sigma_{S_2} = \Sigma_{I_2}$. If $\mathcal{I}_1$ tioco $\mathcal{S}_1 \wedge \mathcal{I}_2$ tioco $\mathcal{S}_2$ then $\mathcal{I}_1 \parallel \mathcal{I}_2$ tioco $\mathcal{S}_1 \parallel \mathcal{S}_2$.*

Subsystems preserve the tioco conformance relation in the composed result because the parallel operator adds extra outputs if subsystems contain non-specified inputs that belong to the synchronization set of actions. If subsystems do not contain underspecified inputs or they are input-complete, the conformance relation is preserved. Consequently, non-conforming subsystems may lead to a conforming composed result if the non-conformance is caused by the synchronizing actions and subsystems are not input-complete.

If subsystems are not input-complete, the tioco conformance relation may not be preserved from subsystems to the composed result because this conformance relation allows underspecification of input actions, which may lead to extra outputs in the composed result that where not present in the subsystems. Consequently, if an extra output is produced in the composed implementation which is not present in the composed specification, the tioco conformance relation is not preserved in the composed system.

Although we restrict the usage of the parallel operator to a binary communication where an input action corresponds to a single output action, we consider that a system can be composed of a subsystem already composed of smaller components so that

output actions generated by subsystems synchronization can be synchronized to other subsystems. Consequently, if we have a subsystem that communicates to one or more subsystems, we maintain this behavior at some level. Besides, since we are dealing with the message-passing paradigm, we might also use this operator in the context of distributed systems. Furthermore, it is important to remark that even though the input-completion constraints on implementations are rather difficult to be met in practice, the parallel compositional operator can be extensively applied for the generation of critical test cases, as we discuss in Section 3.4.

3.4 Test Case Generation Process

This section presents a test case generation process for compositional real-time systems that can be applied as part of integration testing activities. The focus is on incremental integration because the binary compositional operators defined restrict models to be combined in pairs. Figure 3.16 shows the integration testing generation process.

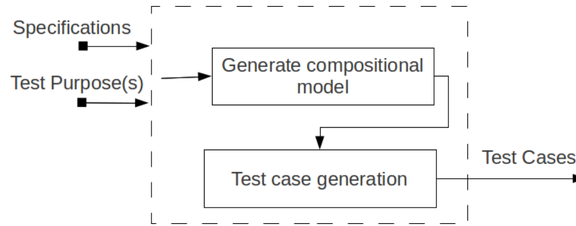


Figure 3.16: The integration test case generation process

The first step is to receive two specifications and a test purpose, modeled as TIOSTSs, that will guide the generation of test cases, where the specifications may be the result of previous composition steps. The second step is to perform the composition of the specification models, depending on the compositional operator(s) chosen, whereas the third step is the test case generation from the composed model by using the approach from Andrade *et al.* [AM13] that is implemented in the SYMBOLRT tool [AACM12].

The definition of the test purpose is essential to an effective test case generation, that is, selection of test cases that meet the testing objectives in a cost-effective way. Since, during integration testing, we intend to test interactions between the subsystems,

we need to define test purposes that allow the automatic generation of test cases that exercise these interactions.

For the sequential operator, it is important to guarantee that the integration actions of interest are traversed. For example, consider the *Choose* and *Pay* subsystems from Section 3.1. In this case, the *send!* action, which is the integration action, may be required in an integration test purpose to guarantee that the integration will be reached. Moreover, if we are interested in test cases that more completely executes the payment flow, we must add, to the test purpose, actions of the *Pay* system so that only traces that reach them are considered. Furthermore, for a finer selection of specific integration scenarios, one might add specific actions of the composed specification models.

Figure 3.17 shows a possible test purpose for the *Choose* $;_{send}$ *Pay* system from Figure 3.4. It allows the test case generation of scenarios where the *send!* and *finish-System!* output actions are presented in the specification, leading to the *Accept* verdict. Based on them, 2 test cases can be generated[2]. For the sake of simplification, we show only a part of the test cases generated by using this test purpose in Figure 3.18. As usual, inputs were changed to outputs and vice-versa to reflect inputs and output from the tester point of view. We can also generate test cases with a test purpose containing the single *send!* action, resulting in 4 test cases that correspond to the possible traces up to the *send!* action. Moreover, we can generate tests cases for the test purpose that comprises the *send!* and *pay!* actions, focusing on check and cash options, resulting in 2 test cases.

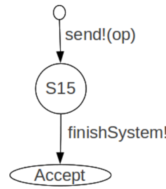


Figure 3.17: Test purpose for the *Choose*$;_{send}$ *Pay* system

Regarding the parallel operator, synchronization actions as well as other actions and parameter values may be required to shorten the number of generated test cases, since this operator produces rather complex models with several different combinations of

[2]In this case, we are considering that loops are traversed only once.

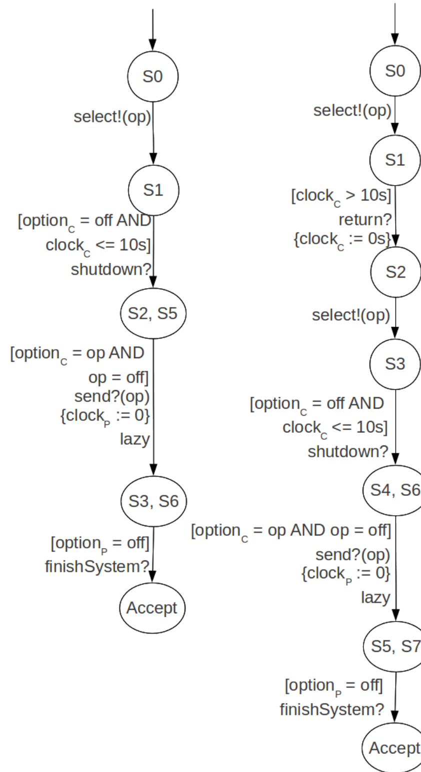


Figure 3.18: Test cases for the $Choose;_{send}\ Pay$ system

traces, particularly when the model contains cycles. For example, if we want to perform the integration testing of the $Mouse \parallel Screen$ system, we can choose a test purpose with a transition that contains either the *doubleClick!* or the *simpleClick!* output action. Figure 3.19 shows an example of the test purpose for this system. We generated 15 test cases from it. Figure 3.20 shows simplified versions of two of these test cases.

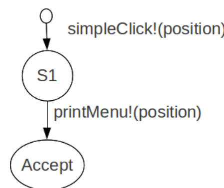


Figure 3.19: Test purpose for $Mouse \parallel Screen$ system

With the same system, we can use other test purposes to obtain different test cases. For example, the parallel composition allows us to acquire 6 test cases with the adoption of the single *doubleClick!* action for a test purpose, while the usage of a test purpose composed by the *simpleClick!* and *doubleClick!* actions lead us to the generation of 45 test cases. In addition, a test purpose containing the *doubleClick!* and *openApp!*

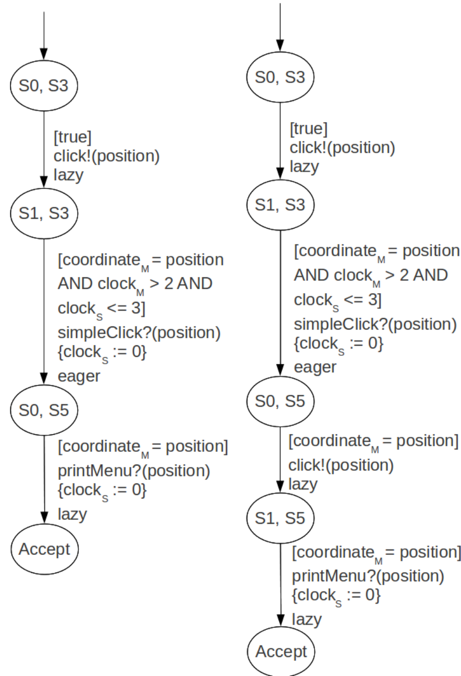


Figure 3.20: Test cases for the *Mouse* || *Screen* system

output actions lead us to the generation of 13 test cases.

The interruption composition follows a line of reasoning which is similar to the sequential operator. in this way, test purposes shall comprise actions that belong to: i) the synchronization set of action from the isolated models; and ii) a finishing action from the first model. For example, Figure 3.21 shows a test purpose where the *send* and *receive* actions belong to the synchronization set of the *Mouse Interrupted* $_{\text{send}}\triangle_{\text{receive}}$ *Reset* system. Additionally, the *simpleClick* indicates that the first subsystem finishes a path. By using the SYMBOLRT tool, we generated 24 test cases. Additionally, a test purpose composed by the *send*, *receive* and *doubleClick* actions produces 24 test cases.

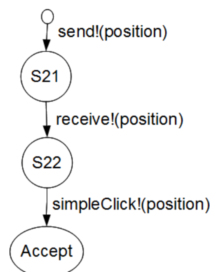


Figure 3.21: Test purpose for *Mouse Interrupted* and *Reset* subsystems

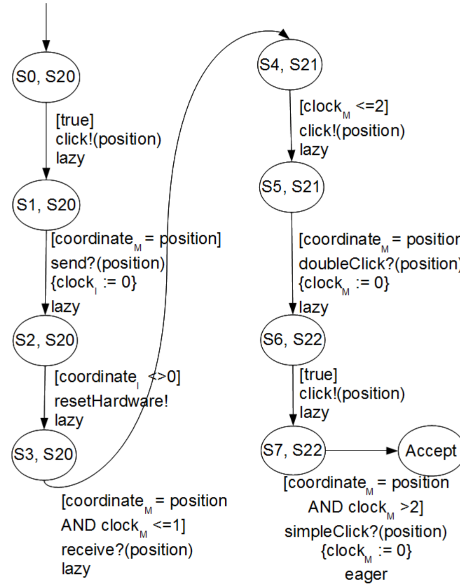


Figure 3.22: Test cases for the *Mouse Interrupted* and *Reset* subsystems

We highlight that the complexity of the generated test cases is related to the tester experience. Besides, a composed system which has an increasing number of loops increases the number of test cases accordingly because more paths are available.

3.5 Concluding Remarks

We introduced the sequential, parallel and interruption operators by using toy examples and applying them in simple applications. In the sequence, we presented their formal definitions and showed that the tioco conformance relation is preserved from subsystems to the composed systems when we use the sequential and interruption operator. Conversely, a system composed by the parallel operator preserves tioco if subsystems are input-complete. Finally, a test generation process that uses the compositional operator was proposed. The operators were implemented in a tool that allowed the generation of test cases for the composed systems we have shown.

Chapter 4

Algorithms

This chapter presents the algorithms used for the sequential, interruption and parallel operators. We use them with the SYMBOLRT tool and let them available in our site[1]. To implement them, we used the Java programming language and the TIOSTS class, which is a simple data class that store elements from the TIOSTS model (Definition 4), as well as gets and sets methods.

Every algorithmic description follows its operator definition. Thus, we used the toy examples previously presented to easy code explanations. Section 4.1 shows the algorithm for the sequential operator, while Sections 4.2 and 4.3 present codes for the interruption and parallel operators, in this order.

4.1 Sequential Composition Algorithm

The implementation for the sequential composition operator uses variables that represent elements from the operator normal form (Definition 8). Consequently, variables names from algorithm denotes their function during the composition process. Figure 4.1 repeats the toy example for the sequential operator to review these elements.

This operator implementation is performed in the method sequentialComposition, that receives two TIOSTS models and returns the model composed by the sequential operator. Basically, we perform four main steps: i) Variables startup; ii) If subsystems fill normal form, we create the transition from location (l^0_{c1}, l^0_2) to (l_{c1}, l'^0_2); iii)

[1] Available at https://sites.google.com/site/compositionaltioco/

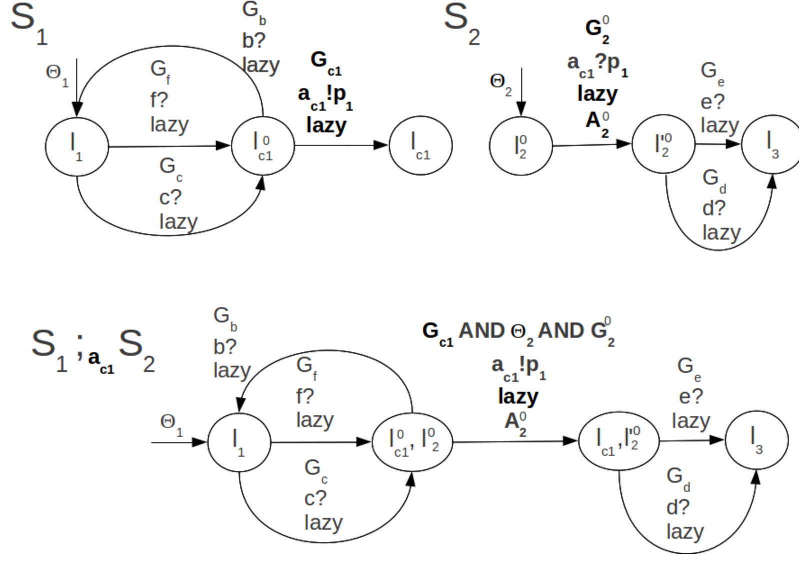


Figure 4.1: Toy example for the sequential composition

Adding $\mathcal{T}_1$ transitions, excepting t_{c1}; and iv) Adding $\mathcal{T}_2$ transitions, excepting t_2^0. Algorithm 4.1 shows the set of instructions to compose subsystems by using the sequential composition operator.

Algorithm 4.1: Sequential Composition

```
1  public TIOSTS sequentialComposition(TIOSTS model1, TIOSTS model2)
2    t02 = tiosts2.getInitialTransition();
3    compositionAction = t02.getActionName();
4    l02 = t02.getSource();
5    l02Line = t02.getTarget();
6    tc1 = getCompositionTransition(tiosts1, compositionAction);
7    lc = tc1.getTarget();
8    l0c1 = tc1.getSource();
9    actionlc = tc1.getAction();
10   actionl02 = t02.getAction();
11   if (isCompatible(tiosts1, tiosts2, lc)){
12     result = initialize(tiosts1, tiosts2, tc1, t02);
13     source = new Location(tc1.getSource()+ "," + t02.getSource());
14     target = new Location(tc1.getTarget()+ "," + t02.getTarget());
15     dataAssignments = tc1.getDataAssignments() + t02.getDataAssignments();
```

```
16    clockAssignments = tc1.getClockAssignments() + t02.getClockAssignments();
17    dataGuard = createDataGuard(tc1.getDataGuard(), t02.getDataGuard());
18    dataGuard = createDataGuard(dataGuard, tiosts2.getInitialCondition());
19    clockGuard = createClockGuard(tc1.getClockGuard(), t02.getClockGuard());
20    result.createTransition(source, dataGuard, clockGuard, actionlc, dataAssignments,
          clockAssignments, DEADLINE_LAZY, target);
21    for(Transition t:tiosts1.getTransitions()){
22      if (!t.equals(tc1)){
23        if (t.getTarget().equals(l0c1)){
24          result.createTransition(t.getSource(), t.getDataGuard(), t.getClockGuard(), t.
              getAction(), t.getDataAssignments(), t.getClockAssignments(), t.getDeadline
              (), source);
25  }else{
26    if (t.getSource().equals(l0c1)){
27      result.createTransition(source, t.getDataGuard(), t.getClockGuard(), t.getAction(), t.
          getDataAssignments(), t.getClockAssignments(), t.getDeadline(), t.getTarget());
28    }else{
29      result.createTransition(t.getSource(), t.getDataGuard(), t.getClockGuard(), t.
          getAction(), t.getDataAssignments(), t.getClockAssignments(), t.getDeadline(), t.
          getTarget());
30          }
31        }
32      }
33    }
34    for(Transition t:tiosts2.getTransitions()){
35      if (!t.equals(t02)){
36        if(t.getSource().equals(l02Linha)){
37          result.createTransition(target, t.getDataGuard(), t.getClockGuard(), t.getAction
              (), t.getDataAssignments(), t.getClockAssignments(), t.getDeadline(), t.
              getTarget());
38        }else{
39          if(t.getTarget().equals(l02Line)){
40            result.createTransition(t.getSource(), t.getDataGuard(), t.getClockGuard(), t.
```

```
            getAction(), t.getDataAssignments(), t.getClockAssignments(), t.
            getDeadline(), target);
41        }else{
42          result.createTransition(t.getSource(), t.getDataGuard(), t.getClockGuard(), t.
            getAction(), t.getDataAssignments(), t.getClockAssignments(), t.
            getDeadline(), t.getTarget());
43        }
44      }
45    }
46  }
47  return result;
48 }
```

The first step is the variables startup, which is performed in lines 2-10. Following the line order of the algorithm, we start t_2^0 value by receiving the initial transition of the $\mathcal{S}_2$, represented by model2 variable. In the sequence, we start the composition action, l_2^0, $l_2'^0$, l_{c1}^0 and l_{c1} by using get and set methods from the TIOSTS class and basing these choices on locations' positions. Since we cannot predict where the t_{c1} is, we start it by using the getCompositionTransition method and the label of the composition action, which is restricted by the normal form to belong to t_{c1}. Also, we store in variables action from t_{c1} and t_2^0 transitions in order to use them in the next steps.

The second step creates the composition transition, so that communication between subsystems $\mathcal{S}_1$ and $\mathcal{S}_2$ is performed (lines 11-20). To perform this task, we verify if subsystems are compatible to the operator normal form by using the isCompatible method. In the sequence, we initialize TIOSTS sets of variables, parameters, transitions, assignments and actions by using the initialize method. Thus, we start the source, target and other elements from this transition. In the sequence, we create the TIOSTS transition by using the createTransition method.

The third step is responsible for including transitions from subsystem $\mathcal{S}_1$ in the composed model (lines 21-33). We add each transition from $\mathcal{S}_1$, excepting t_{c1} transition (line 22). Besides that, a special treatment is devoted to transitions whose target or source location is l_{c1}^0: this location is replaced by the source location, recently created in

step ii). These set of instructions are performed in lines 23-27. Next, we add transitions that do not use the l^0_{c1} location as target or source locations (line 29).

In the last step, we add transitions from $\mathcal{S}_2$ in the composed model (lines 34-46). First, we separate transitions whose target location is l'^0_2 and replace them by the target location created in step ii), which belongs to the synchronization transition (lines 39-40). After that, we add transitions that do not contain the l'^0_2 location in the composed model (line 42). The result variable contains the composed model.

4.2 Interruption Composition Algorithm

The implementation of the interruption operator follows labels defined during the definition and presentation of its normal form, exemplified with the aid of a toy example. Figure 4.2 shows this example to review the names used for each element from the subsystems and the composed system. We suggest the reader to follow the implementation explanation by using the toy example.

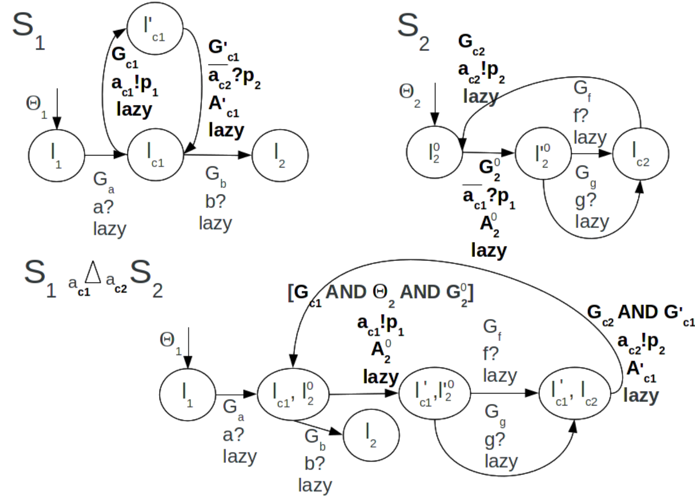


Figure 4.2: Example of weak interruption composition

The implementation of the interruption composition operator receives two models to be composed and returns the composed model. Algorithm 4.2 shows the code. First, we verify if subsystems $\mathcal{S}_1$ and $\mathcal{S}_2$ are compatible (line 2) in accordance to rules defined by the WIC normal form (Definition 10). Next, the intersectionActions method receives actions from both models and returns actions that are common to both sets (line

3). Since these actions labels re the same, we use them as the synchronizing action to be used in the resulting model. Thus, we define the starting values for elements from the resulting model in line 4. Finally, line 5 uses the sync method to update the synchronization transition in the resulting model.

Algorithm 4.2: Interruption Composition

```
1  TIOSTS interruptionComposition(TIOSTS model1, TIOSTS model2){
2    if (isCompatible(model1, model2)){
3      Collection eSync = intersectionActions(tiosts1.getActions(), tiosts2.getActions());
4      result = initialize(tiosts1, tiosts2, eSync);
5      result = sync(tiosts1.getTransitions(), tiosts2.getTransitions(), eSync);
6    }
7    return result;
8  }
```

The sync method receives two sets of transitions and a set of synchronization actions, returning a composed TIOSTS updated according to the sync operator rules (Definition 11). The implementation is divided into two steps: i) Creation of the (l'_{c1}, l_{c2}) and (l_{c1}, l_2^0) synchronization locations with the synchronization transitions and ii) Addition of the unsynchronizable transitions from the composed model. We show the sync operator code in Algorithm 4.3.

The creation of the synchronization locations is performed in lines 2-24. The process repeats for each action from the eSync set of action, which contains the set of synchronization actions (line 4). The first step is to recover the synchronization transitions from the sets of transitions $\mathcal{T}_1$ and $\mathcal{T}_1$ (lines 5 and 6). After that, we start a set of excluded locations with locations that belonged to synchronization location so that these locations do not belong to the composed system (line 7). Next, lines 8-10 initiate the source and target locations with a combination of labels from the excluded locations and add the newly created locations to the composed TIOSTS. In the sequence, we we initiate the remaining transition elements (lines 11-24). Lines 16-22 gives a special treatment to actions because it assures that an output action is seleced and included in the composing transition. After that, line 23 creates the transition with each element defined in previous lines.

The addition of unsynchronizable transitions from subsystems to the composed model is performed in lines 26-69. It happens for each action from the set of synchronizable actions (line 25). First, lines 29-47 adds transitions from $\mathcal{T}_1$. We add transitions in the composed systems by maintaining the previous transition elements of $\mathcal{T}_1$, but we replace locations l_{c1} and l'_{c1} by (l'_{c1}, l_{c2}) and (l_{c1}, l^0_2), in this order. Similarly, lines 48-67 replaces the l_{c2} and l^0_2 locations in $\mathcal{T}_2$ by (l'_{c1}, l_{c2}) and (l_{c1}, l^0_2) and add them to the resulting model.

Algorithm 4.3: sync operator

```
1  TIOSTS sync(Collection transitions1, Collection transitions2, Collection eSync){
2    excludedLocations = new Collection();
3    result = new TIOSTS();
4    for (Action actionSync:eSync){
5      t1Sync = recoverTransition(transitions1, actionSync);
6      t2Sync = recoverTransition(transitions2, actionSync);
7      excludedLocations.add(t1Sync.getSource(), t2Sync.getSource(), t1Sync.getTarget(),
           t2Sync.getTarget());
8      source = new Location(t1Sync.getSource() +”,”+ t2Sync.getSource());
9      target = new Location(t1Sync.getTarget() +”,”+ t2Sync.getTarget());
10     result.addLocation(source, target);
11     dataGuard = t1Sync.getDataGuard() + t2Sync.getDataGuard();
12     transitionl1 = t2Sync.getSource().getInTransitions().get(0);
13     dataAssignments = createAssignments(t1Sync.getDataAssignments(), t2Sync.
           getDataAssignments());
14     clockAssignments = createAssignments(t1Sync.getClockAssignments(), t2Sync.
           getClockAssignments());
15     clockGuard = createClockGuard(t1Sync.getClockGuard(), t2Sync.getClockGuard());
16     if(t1Sync.getAction().getType() == ACTION_OUTPUT){
17       action = result.recoverAction(t1Sync.getAction());
18       action.setParameters(t1Sync.getAction().getParameters());
19     }else{
20       action = result.recoverAction(t2Sync.getAction());
21       action.setParameters(t2Sync.getAction().getParameters());
```

```
22      }
23      result.createTransition(source, dataGuard, clockGuard, action, dataAssignments,
            clockAssignments, DEADLINE_LAZY, target);
24    }
25    for (Action actionSync:eSync){
26      tSync = recoverTransition(result.getTransitions(), actionSync);
27      t1Sync = recoverTransition(transitions1, actionSync);
28      t2Sync = recoverTransition(transitions2, actionSync);
29      for(Transition t:transitions1){
30        if (!eSync.contains(t.getAction())){
31          if(t.getTarget().equals(t1Sync.getSource())){
32            result.createTransition(t.getSource(), t.getDataGuard(), t.getClockGuard(), t.
                getAction(), t.getDataAssignments(), t.getClockAssignments(), t.getDeadline
                (), tSync.getSource());
33          }else{
34            if(t.getTarget().equals(t1Sync.getTarget())){
35              result.createTransition(t.getSource(), t.getDataGuard(), t.getClockGuard(), t.
                  getAction(), t.getDataAssignments(), t.getClockAssignments(), t.
                  getDeadline(), tSync.getTarget());
36            }else{
37              if(t.getSource().equals(t1Sync.getTarget())){
38                result.createTransition(tSync.getTarget(), t.getDataGuard(), t.
                    getClockGuard(), t.getAction(), t.getDataAssignments(), t.
                    getClockAssignments(), t.getDeadline(), t.getTarget());
39              }else{
40                if(t.getSource().equals(t1Sync.getSource())){
41                  result.createTransition(tSync.getSource(), t.getDataGuard(), t.
                      getClockGuard(), t.getAction(), t.getDataAssignments(), t.
                      getClockAssignments(), t.getDeadline(), t.getTarget());
42                }
43              }
44            }
45          }
```

```
46         }
47       }
48       for(Transition t:transitions2){
49         if (!eSync.contains(t.getAction())){
50           if(t.getTarget().equals(t2Sync.getSource())){
51             result.createTransition(t.getSource(), t.getDataGuard(), t.getClockGuard(), t.
                   getAction(), t.getDataAssignments(), t.getClockAssignments(), t.getDeadline
                   (), tSync.getSource());
52           }else{
53             if(t.getTarget().equals(t2Sync.getTarget())){
54               result.createTransition(t.getSource(), t.getDataGuard(), t.getClockGuard(), t.
                     getAction(), t.getDataAssignments(), t.getClockAssignments(), t.
                     getDeadline(), tSync.getTarget());
55             }else{
56               if(t.getSource().equals(t2Sync.getTarget())){
57                 result.createTransition(tSync.getTarget(), t.getDataGuard(), t.
                       getClockGuard(), t.getAction(), t.getDataAssignments(), t.
                       getClockAssignments(), t.getDeadline(), t.getTarget());
58               }else{
59                 if(t.getSource().equals(t2Sync.getSource())){
60                   result.createTransition(tSync.getSource(), t.getDataGuard(), t.
                         getClockGuard(), t.getAction(), t.getDataAssignments(), t.
                         getClockAssignments(), t.getDeadline(), t.getTarget());
61                 }
62               }
63             }
64           }
65         }
66       }
67     }
68     return result;
69   }
```

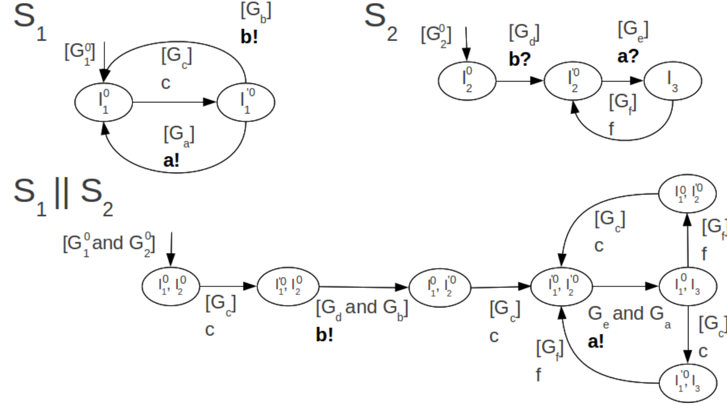


Figure 4.3: Toy example for the parallel composition

4.3 Parallel Composition Algorithm

The parallel composition composes subsystems by synchronizing actions with the same label and different input/output types, adding an output action to the composed system. Also, we perform interleaving to the non-synchronizable set of actions. Figure A.3 shows a toy example already presented in Chapter 3 to illustrate this operator behavior. We suggest the reader to follow this section explanation by using this toy example and Definition 13.

The implementation of the parallel composition operator receives two subsystems and returns a composed model. It uses the isCompatible method to verify if the intersection between the sets clocks, variables and locations from subsystems is empty. If it is true, we start the sets from the composed model. Also, we defined the sets of unsynchronizable actions from the Σ_1 and Σ_2 by using the minusSet method. In the sequence, we call the parComposition method in order to build the set of transitions and locations of the composed system. Algorithm 4.4 shows the theses steps.

Algorithm 4.4: Parallel Composition

```
1 TIOSTS parallelComposition(TIOSTS tiosts1, TIOSTS tiosts2){
2   if (isCompatible(tiosts1, tiosts2)){
3     result = initialize(tiosts1, tiosts2);
4     actionst1Minust2 = minusSet(model1.getActionNames(), model2.getActionNames());
5     actionst2Minust1 = minusSet(model2.getActionNames(), model1.getActionNames());
6     parComposition(tiosts1.getInitialLocation(), tiosts2.getInitialLocation(), result,
```

```
        actionst1Minust2, actionst2Minust1);
7     }
8   return result;
9   }
```

We show the parComposition method in Algorithm 4.5. It receives locations and actions from subsystems $\mathcal{S}_1$ and $\mathcal{S}_2$ and a partial version of the composed system. We recursively build the composed system pc from two subsystems. Because of that, line 2 provides a stopping criteria by assuring that the current locations from Σ_1 and Σ_2 were not visited. In the sequence, lines 2 and 3 perform a combination between each subsystems transitions of the non-visited locations to add the new transition in the composed model. From this point, a transition action follows two cases: i) The action belongs to the set of unsynchronizable actions from $\mathcal{T}_1$ or $\mathcal{T}_2$ and ii) The action belongs to the synchronization set of actions.

Algorithm 4.5: par composition method

```
1   void parComposition(Location loc1, Location loc2, TIOSTS pc, List actionsT1, List
        actionsT2){
2     if (isVisited(loc1,loc2)) return;
3     for (Transition t1 : loc1.getOutTransitions()) {
4       for (Transition t2 : loc2.getOutTransitions()){
5         action1 = t1.getAction();
6         action2 = t2.getAction();
7         if (actionsT1.contains(action1)){
8           source = new Location(loc1 + "," + loc2);
9           pc.addLocation(source);
10          target = new Location(t1.getTarget() + "," + loc2);
11          pc.addLocation(target);
12          pc.createTransition(source, t1.getDataGuard(), t1.getClockGuard(), t1.getAction(),
                t1.getDataAssignments(), t1.getClockAssignments(), t1.getDeadline(), target);
13          parComposition(t1.getTarget(), loc2, pc, actionsT1, actionsT2);
14        }
15        if (actionsT2.contains(action2)){
```

```
16        source = new Location(loc1 + ”,” + loc2);
17        pc.addLocation(source);
18        target = new Location(loc1 + ”,” + t2.getTarget());
19        pc.addLocation(target);
20        pc.createTransition(source, t2.getDataGuard(), t2.getClockGuard(), t2.getAction(),
              t2.getDataAssignments(), t2.getClockAssignments(), t2.getDeadline(), target);
21        parComposition(loc1, t2.getTarget(), pc, actionsT1, actionsT2);
22      }
23      if (canSynchronize(action1, action2)){
24        source = new Location(loc1 + ”,” + loc2);
25        pc.addLocation(source);
26        target = new Location(t1.getTarget()+ ”,” + t2.getTarget());
27        pc.addLocation(target);
28        dataAssignments = t1.getDataAssignments() + t2.getDataAssignments();
29        clockAssignments = t1.getClockAssignments() + t2.getClockAssignments();
30        dataGuard = t1.getDataGuard() + t2.getDataGuard();
31        clockGuard = t1.getClockGuard() + t2.getClockGuard();
32        parameters = action2.getParameters();
33        if (t1.getAction().getType() == ACTION_OUTPUT){
34          newAction = t1.getAction();
35        }
36        if (t2.getAction().getType() == ACTION_OUTPUT){
37          newAction = t2.getAction();
38        }
39        newAction.setParameters(parameters);
40        pc.createTransition(source, dataGuard, clockGuard, newAction, dataAssignments,
              clockAssignments, deadlineOperator(t1.getDeadline(), t2.getDeadline()), target
              );
41        parComposition(t1.getTarget(), t2.getTarget(), pc, actionsT1, actionsT2);
42      }
43    }
44  }
45 }
```

If the action belongs is not synchronizable, we perform an interleaving. If the action belongs to the set of actions from $\mathcal{S}_1$, we create the source, target locations and use the action from $\mathcal{T}_1$. After that, we create a transition from these elements in the composed system and a recursive call is performed to build the remainf transitions from this model (lines 7-14). The same happens to transitions added from $\mathcal{T}_1$ (lines 15-22).

On the other hand, if actions from both subsystems synchronize, we create new source and target locations and perform unions from data and clock assignments and parameters from transitions of $\mathcal{S}_1$ and $\mathcal{S}_2$ models (lines 24-32). In addition, lines 33-39 identifies from which subsystem the output action belongs to be further added to the composed system. In the sequence, the transition is created in the composed systems by using the deadline method that chooses the stricter deadline following the order eager $>$ delayable $>$ lazy. Finally, the parComposition method is called again to build the remaining model transitions (lines 40-41).

4.4 Concluding Remarks

This chapter presented the algorithmic implementations of the sequential, interruption and parallel compositional operators. We presented each method, explaining their detailed steps.

Chapter 5

Exploratory Studies

This chapter presents two exploratory studies that use the sequential, parallel and interruption operators in two different applications. First, we show the *Avionics* specification (Section 5.1). In the sequence, we detail the *Cell Phone* application (Section 5.2).

The main objective of this study is to evaluate the generation of test cases by using the sequential, interruption and parallel operators with respect to the applicability of our approach in the context of the SYMBOLRT tool. We give results about each model size and the time used to generate test cases.

For the sake of simplification, every transition from TIOSTSs of these exploratory studies have omitted deadlines, which implies that we assume the lazy value for input actions and delayable value for output actions. In addition, we implemented every model in a tool and presented our results and the generated code in a web page[1]. We generated test cases by using a computer with the following settings: Ubuntu 12.04 (64-bits), 12 GB RAM, 500 Gb HD, Intel Core i7-4770 processor (3.40 GHz), CVC 2.2 and UPPAAL DBM Library 2.0.7.

5.1 Avionics System

The complexity of avionics systems requires strict tests. A defect can cause a serious disaster if it is discovered when the system is in use. Among various examples, this

[1] https://sites.google.com/site/compositionaltioco/

section presents a few parts of a mission-critical software, particularly its tracking subsystem, which is a real-time subsystem in the avionics domain inspired by the Generic Software Avionics Specification Report [LVLG90].

Figure 5.1 shows a block diagram for the tracking subsystem composed of three subsystems: i) the *Radar Control* subsystem is used to display a radar view of the terrain in order to detect and identify possible targets and detailed information about them; ii) The *Target Designation* subsystem allows the designation of a target by the aircrew; and iii) the *Target Tracking* subsystem tracks the target, depending on the selected mode of the target designation subsystem. The last two subsystems are sequentially composed: the *Target Designation* subsystem passes the mode and target identification values through the *finishTargetDesignation* action that determines if the *Target Tracking* subsystem is tracked by the *Radar Control* or *HUD* subsystems (which we do not specify in this exploratory study). Similarly, the *Radar* communicates to the *Tracking* subsystem through the *targetPosition* and *track* actions. We assume that the three subsystems are developed separately.

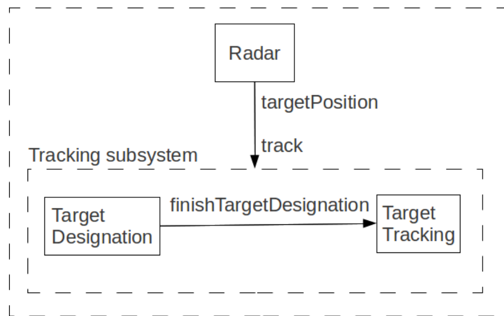


Figure 5.1: Tracking system of the generic avionics specification

Figure 5.2 shows the *Radar* subsystem which is responsible for detecting new targets. In the S36 location, the subsystem can receive an option through $select_R?$ action and store it in $mode_R$ variable. From location S37, the subsystem allows the execution of two branches: 1) if the $mode_R$ variable equals to *groundMap* option, it inputs the *receive?* action and stores its value in *terrainView* value or 2) if $mode_R$ equals to *groundSearch* option, it receives a value through *range?* input action and stores it in *rangeContacts* variable. From the S38 location, the subsystem outputs the *info* parameter through the *display!* output action if it is equal to *terrainView* variable. If the *rangeContacts*

variable value is bigger than 10 contacts and the clock is less or equal to $82ms$, from location S39, the subsystem is allowed to receive a value using the *detect?* input action and stores it in the *rangeContacts* variable. From the S40 location, the subsystem can receive the target coordinates through the *targetPosition?* input action and stores them in $azimuth_R$, $elevation_R$ and $range_R$ variables. From the S41 location, the subsystem receives the target being tracked and stores it in the *contactID* variable by using the *track?* input action. From the S42 location, the subsystem shows the number of the target being tracked to other subsystems by using the *display!* output action.

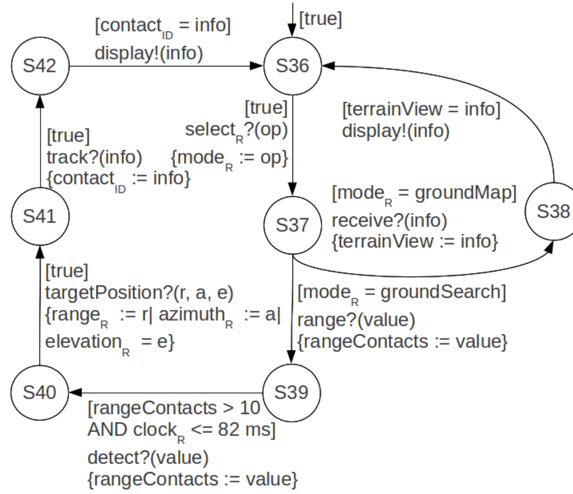


Figure 5.2: *Radar* specification

The *Target Designation* subsystem is shown in Figure 5.3. It can work under three modes: *HUDDesignation*, *RadarDesignation* and *undesignationTarget*. From the S9 location, this subsystem allows the selection of a mode and a target identifier through the $select_{TD}$*?* input action, storing their values in the $mode_{TD}$ and $target_{TD}$ variables and goes to the S10 location. In the S10 location, the subsystem can: 1) undesignate a target through the *undesignate!* output action if $mode_{TD}$ equals to *undesignationTarget* and returns to the S10 location; 2) emit the *changeModeHUD!* output action to other subsystem modules if the $mode_{TD}$ variable equals to *HUDDesignation* and goes to the S15 location and 3) output the *changeModeRadar!* action to other subsystem modules if the $mode_{TD}$ variable equals to *radarDesignation* and goes to the S11 location. In the S11 location, the subsystem can receive the location aircraft coordinates through the *aircraftPosition?* input action, storing their values in $range_{TD}$, $azimuth_{TD}$ and

$elevation_{TD}$ variables and goes to the S12 location. In S12 location, the subsystem communicates the target position using the *targetPosition!* output action and goes to the S13 location. In the S13 location, the subsystem designates a target through the *designate!* output action if this subsystem clock equals to $200ms$ and goes to the S17 location. In the S17 location, the subsystem communicates the $mode_{TD}$ and $target_{TD}$ variable values through the *finishTargetDesignation!* output action to other subsystems and goes to the S18 location. In the S14 location, the subsystem can update the target location using the reticle coordinates through the $reticlePosition_{HUD}?$ input action and stores their value in the $azimuth_{TD}$ and $elevation_{TD}$ variables, going to the S15 location. In the S15 location, the subsystem can designate a target by using the *designate!* output action if the $clock_{TD}$ variable equals to $200ms$ and goes to the S16 location. In the S16 location, the subsystem communicates the coordinates by using the $reticlePosition_{HUD}!$ output action and goes to S17 location.

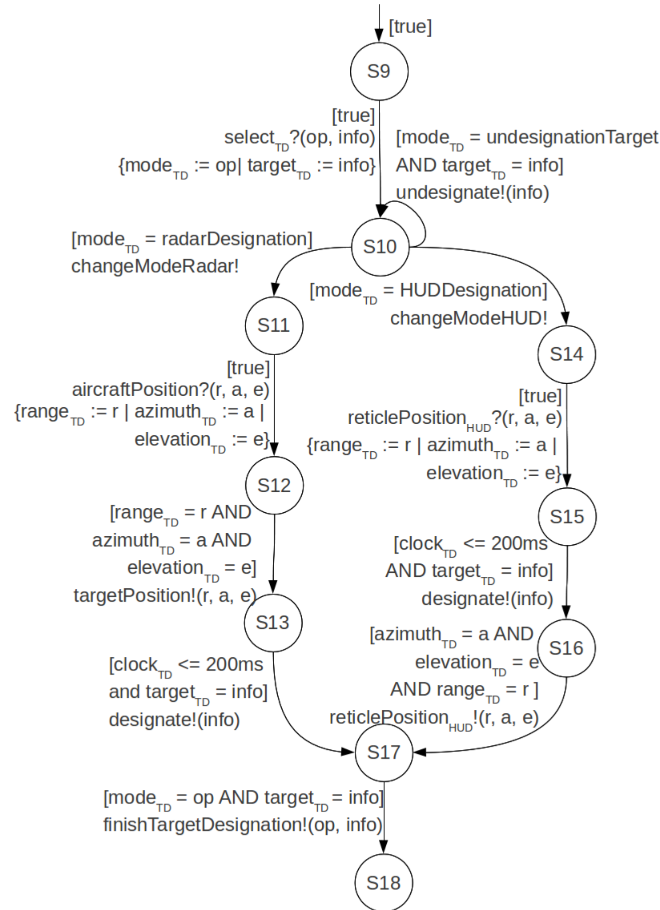


Figure 5.3: *Target Designation* specification

Figure 5.4 shows the *Target Tracking* specification. It also operates under the *HUD-Designation* and *RadarDesignation* modes, previously set during the target designation execution. From the S21 location, the subsystem can receive the operation mode and target identification values through *finishTargetDesignation?* input action and stores them in $mode_{TT}$ and $target_{TT}$ variables, going to the S22 location. From the S22 location, the subsystem can perform one of the following: 1) communicate its operation change to radar mode through the *changeModeRadar!* output action if the $mode_{TT}$ variable value equals to *RadarDesignation* and goes to the S23 location or 2) inform the subsystem change to HUD mode through the *changeModeHUD!* output action if the $mode_{TT}$ variable value equals to *HUDDesignation* and goes to the S24 location. From the S23 location, the subsystem receives the aircraft coordinates and stores them in the $range_{TT}$, $azimuth_{TT}$ and $elevation_{TT}$ variables and goes to the S25 location. From the S24 location, the subsystem can receive the target and aircraft locations through the $reticlePosition_{HOTAS}$*?* input action and stores them in the $azimu\text{-}th_{TT}$, $elevation_{TT}$ and $range_{TT}$ variables, going to S25 location. In the S25 location, the subsystem informs to other subsystems that the radar is tracking a target identified by the $target_{TT}$ variable through the *track!* output action and goes to the S26 location. From the S26 location, the subsystem communicates the later coordinates through the $reticlePosition_{HUD}$*!* output action and goes to the S27 location. From the S27 location, the same coordinates are updated to other modules using the *update!* output action if the clock is less or equal to $40ms$ and goes to the S28 location. Finally, from the S28 location, the subsystem finishes through the *finishTargetTracking!* output action and goes to the S29 location.

We show the sequential composition of the *Target Designation* and *Target Tracking* subsystems in Figure 5.5. We replaced the S17, S18, S21 and S22 locations by the (S17, S21) and (S18, S22) locations. The (S17, S21) $\xrightarrow{\text{finishTargetDesignation!(op, info)}}$ (S18, S22) transition allows the communication of $mode_{TD}$ and $target_{TD}$ variables from the *Target Designation* subsystem to $mode_{TT}$ and $target_{TT}$ variables from the *Target Tracking* subsystem. Thus, the added transition communicates the $target_{TD}$ and $mode_{TD}$ variables through the *finishTargetDesignation!* output action and stores their values in the $mode_{TT}$ and $target_{TT}$ variables to be further used in the *Target Tracking* subsystem.

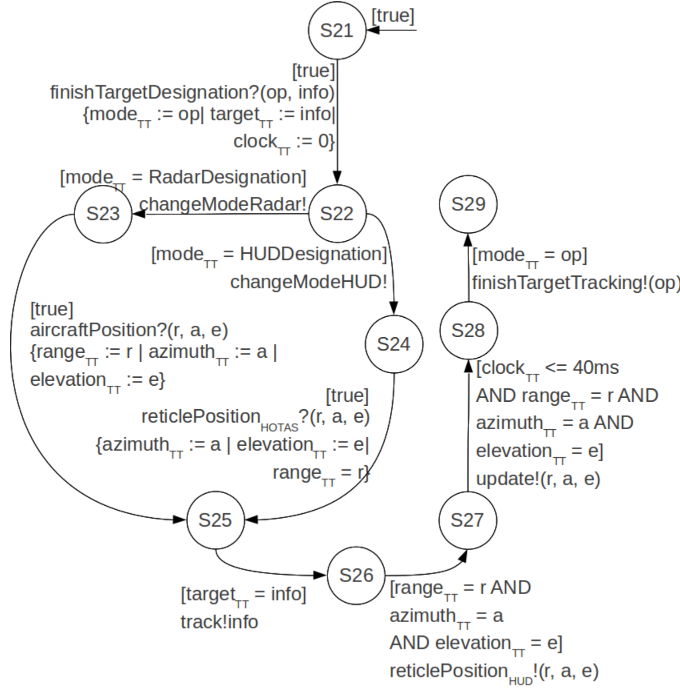


Figure 5.4: *Target Tracking* specification

The parallel composition (*Target Designation* $;_{\text{finishTargetDesignation}}$ *Target Tracking*) $\|$ *Radar* was automatically generated using the parallel operator definition from Section 3.3 and comprises 91 locations and 170 transitions. This number of system transitions and locations is increased with regard to their subsystems because there is only the *targetPosition* and *track* synchronization actions common to both of them. Consequently, unsynchronized actions repeat their transitions with different locations along the resulting system. For example, although the *undesignate!* output action occurs between S10 $\xrightarrow{\text{undesignate!info}}$ S10 transition of the *Target Designation* subsystem, the composed system offers the same action in transitions where source and target locations are (S10,S36); (S10,S37); (S10,S38); (S10,S39) or (S10,S40).

Due to the definition of the sequential and parallel composition operators, we are able to generate integration level test cases that automatically cover different combinations of behaviors from the composed models. Figures 5.6 and 5.7 present the test purposes considered for test generation of our sequential and parallel examples. They intend to extract test cases that would not be generated if each subsystem were tested separately. The test purpose of Figure 5.6 guided the generation of 2 test cases which start in the *Target Designation* subsystem and end at *Target Tracking*, using the output

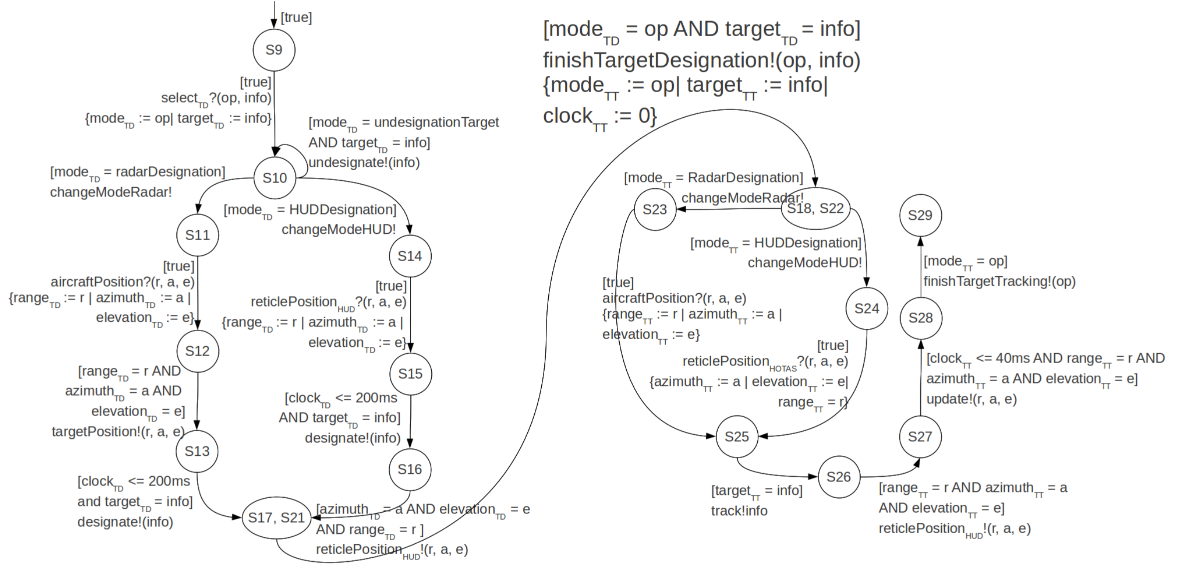


Figure 5.5: Sequential composition for the *Tracking* subsystem

action *finishTargetDesignation!* common to both of them and performing an interface between both subsystems. This generation spent 2s. The usage of another test purpose with the single *finishTargetDesignation!* output action also gave us 2 different test cases after 1s of system processing.

Similarly, the test purpose of Figure 5.7 intends to generate test cases that interleave actions from our sequential composition result and the *Radar* subsystem by using the $select_{TD}?$, $select_R?$ and *undesignate!* actions. This depicts a scenario where a target is undesignated in the *Target* system and interleaved by the $select_R?$ input action from the *Radar* subsystem. Since we wanted to lessen the number of test cases, some actions lead to the Reject location. With this test purpose, we acquired 14 test cases within 21s. Another test purpose composed by the *targetPosition!*, *track!* and *finishTargetDesignation!* output actions gave us 7 test cases, which were generated in 22s.

Following the same strategy, we generated 4193 test cases for the parallel composition system by using a test purpose composed by the *targetPosition!*, *track!*, *finishTargetDesignation!* and *finishTargetTracking!* output actions. In order to lessen the number of test cases, this test purpose also reject some actions through the Reject location. This increased number of test cases and their execution generation time is proportional to the size of the parallel composed system, to paths that contain loops

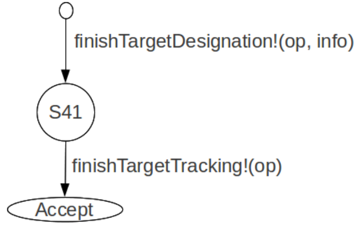


Figure 5.6: Test purpose for the *Target* system

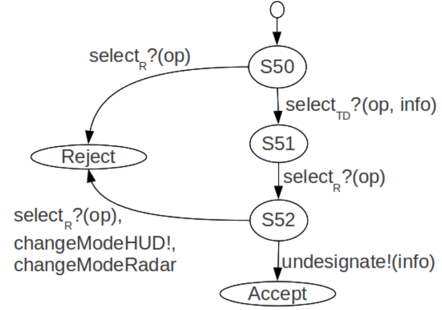


Figure 5.7: Test purpose for *Target* system || *Radar*

Table 5.1: Test purposes and generation time of test cases

Site Identifier	# Test Cases	Time
FinishTargetDesignation	2	$2s$
SelectUndesignate	14	$21s$
SelectUndesignateFinish	7	$22s$
TargetPositionTrackFinish	4193	$2.11h$

and are reached by the test purpose we are using and at the same time do not finish in a Reject location. The integration tests were generated in 2.11 hours.

Table 5.1 shows a summary of the generated test cases fro the Avionics system. We present the used identifiers in our site to each test purpose, the number and time to generated test cases. We observed that the time used to generate test cases increased with the size of transitions and locations of the model. Also, it lasts longer to generate test cases from a test purpose that contains an action belonging to paths with a big number of transitions.

5.2 Cell Phone System

Real-time systems are usually an important part of cell phones, where applications are often composed of features that may interact. Eldh *et al.* [EPHJ07] shows that more than 38% of software faults presented by a large complex telecommunication industry

middleware system come from unclear specifications, happening for the first time at integration or system level. Additionally, Lorentsen [LTX01] cites three categories of cell phone feature interactions that are hard to test: i) feature use interaction; ii) shared limited resources; and iii) when one feature affects another by making it unavailable.

In this section, we present an exploratory study to illustrate our approach to generate interruption testing for a cell phone system, composed of 3 features of the smartphone Nexus 5 with Android operating system version 4.4.3. To compose the features, we apply the interruption operator, proposed in this paper, as well as the sequential composition operator. Generally, the sequential operator communicates information from one subsystem to another through a single action present in both subsystems. It requires an ordering of interactions so that the first subsystem finishes before the second starts.

Our goal is to extract test cases that we would not consider if we test each subsystem separately. Therefore, we create test purposes composed by actions that cover the integration actions of the whole model, which are *sendMessageSelected*, *receiveInterruption* or *sendInterruption*. Moreover, we include actions that finish each scenario of interest in order to allow the selection of a complete integration scenario as test case (otherwise, test cases would resume immediately after the integration action execution). Since the composed model is big, we need to use the *Reject* location in every test purpose to limit the number of test cases selected. To exemplify our approach, we generate test cases by using a computer with the following settings: Ubuntu 12.04 (64-bits), SYMBOLRT 1.3, 12 GB RAM, 500 GB HD, and Intel Core i7-4770 processor (3.40 GHz).

The Cell Phone system comprises the following items: i) the *Contacts* feature that manages a cell phone agenda; ii) the *Message* feature whose function is to send and show messages; iii) the *Receive Call* feature that manages receiving incoming calls. To compose the system, we apply: i) the sequential composition between the *Contacts* and *Message* features through the *sendMessageSelected* action; ii) the sequential composition between *Receive Call* and *Message* features through the *sendMessageSelected* action; iii) The weak interruption composition between the resulting compositions of i) and ii), by *sendInterruption* and *receiveInterruption* actions. Figure 5.8 shows the

Cell Phone system composition structure.

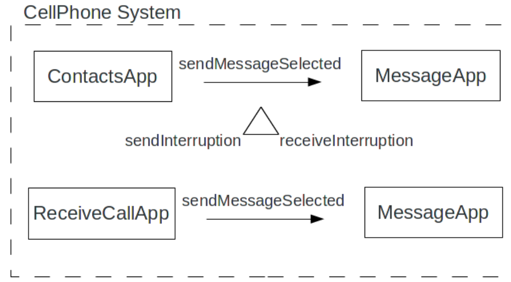


Figure 5.8: *Cell Phone* system

The TIOSTS models of the features and the composed system cannot appear in this paper, but they are available at our web site[2] along with all other artifacts of this exploratory study. Overall, the composed system has 28 locations and 34 transitions.

Because the compositions use the sequential and interruption operators, normal forms shall be applied to them. To highlight transitions that fill these requirements, we use dotted lines.

The first model is the *Contacts* application (Figure 5.9). It models the behavior of an ordinary contacts application, comprising the functionalities to add, delete, edit and send message to a contact as follows.

- From location S0, the system inputs the option to be executed through the *select?* input action and stores its value in $option_{contacts}$ variable with no urgency.

- From the S2 location, there can be two ways: the system executes the *searchContactSelected!* output action if the $option_{contacts}$ is equal to searchContact or it emits the *addContactSelected!* output action if $option_{contacts}$ is equal to addContact.

- From S3 location, the contact name and number are stored in the $contactName$ and $number_{contacts}$ variables through the *insert?* input action.

- The S4 location allows the termination of the contacts' addition be communicated to other subsystems through the $done_{contacts}!$ output action.

[2]https://sites.google.com/site/compositionaltioco/

- In the S6 location, the system can input the name and number of the user, storing these information in the $contactName$ and $number_{contacts}$ and resetting this subsystem's clock.

- From the S7 location, the output action *display!* emits a contact name within $1s$.

- In location S8, the input action *select?* stores another value for the $option_{contacts}$ and goes to location S9.

- The S9 location allows three different behaviors: i) if $option_{contacts}$ equals to sendMessage, the system outputs the *messageAppSelected!* action with this option and the current contact number and goes to location S16; ii) if $option_{contacts}$ is equal to deleteContact, the system executes the *deleteContactSelected!* output action and goes to the S13 location; iii) if $option_{contacts}$ is equal to editContact, the system emits the *editContactSelected!* output action and goes to the S10 location.

- The S13 location receives the *confirm?* input action and stores it in the $answer$ variable. If $answer$ equals to yes, from location S14, the *delete* input action is executed.

- From location S15, the system can confirm that the contact was deleted by executing the $done_{contacts}!$ output action with the deleteContact value.

- The S10 location allows the storage of information in the $option_{contacts}$ and $contactName$ variables through the *insert?* input action, also resetting this subsystem clock.

- From the S11 location, the contact number is shown through the *display!* output action if the clock is less or equal to $1s$.

- Finally, from location S12, the edition termination of the current contact is communicated to other subsystems through the $done_{contacts}!$ output action.

The *Message* application offers the features of showing a message or typing data to be sent later (Figure 5.10).

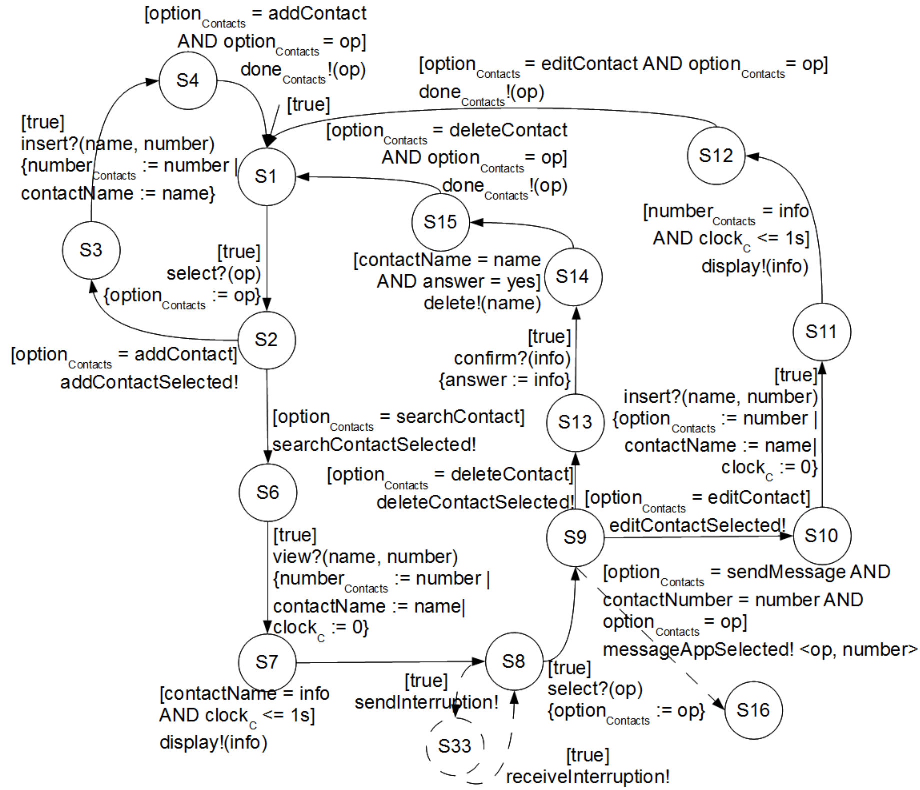


Figure 5.9: *Contacts* subsystem

- In the initial S17 location, the *messageAppSelected?* input action receives the option and phone number, storing these data in the *choice* and *phoneNumber* variables and resets the clock.

- From location S18, there are two possible ways: i) if *choice* equals to searchMessage, the phone number is sent to other subsystems through the *showMessage!* output action, going to location S22 or ii) if *choice* equals to composeMessage, the *type?* input action receives data and stores in the *message* variable, going to the S19 location.

- From location S22, the end of the searchMessage process is communicated to other subsystems through the $done_{Message}!$ output action within $2s$.

- The S19 location allows the message content and the phone number be sent through the *transmit!* output action, resetting the system clock.

- Finally, from location S20, the system communicates that the message was sent

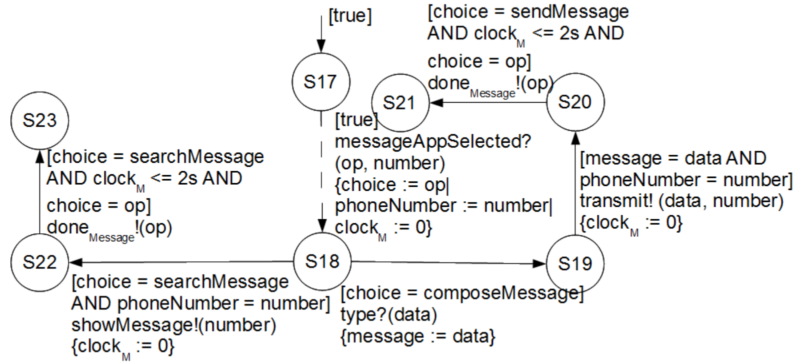


Figure 5.10: *Message* subsystem

to other subsystems.

The *Receive Call* application is responsible for allowing a cell phone to receive a call as follows.

- The S24 initial location allows the reception of the *numberCall* and $option_{call}$ variables through the *callSelected?* input action.

- From location S25, the system outputs the *callReceived!* action if $option_{call}$ equals to receiveCall.

- The S26 location allows the system to receive an option within 1s and stores this value in $option_{call}$ variable. From location S27, the system can: i) reject a call through the *reject!* output action if $option_{call}$ is equal to rejectCall and goes to location S28; ii) start a conversation through the *talk*! output action if $option_{call}$ equals to acceptCall and goes to location S30 and iii) activate the *Message* application through the *messageAppSelected!* and inform the option sendMessage and the phone number being called.

- The S30 location communicates the acceptCall option to other subsystems through the $done_{Call}!$ output action.

- From location S28, the system informs to other subsystems that the rejectCall process was finished.

To illustrate the possible test case scenarios that the tester can create from the composed model, we describe 4 examples of test purposes with the corresponding generated

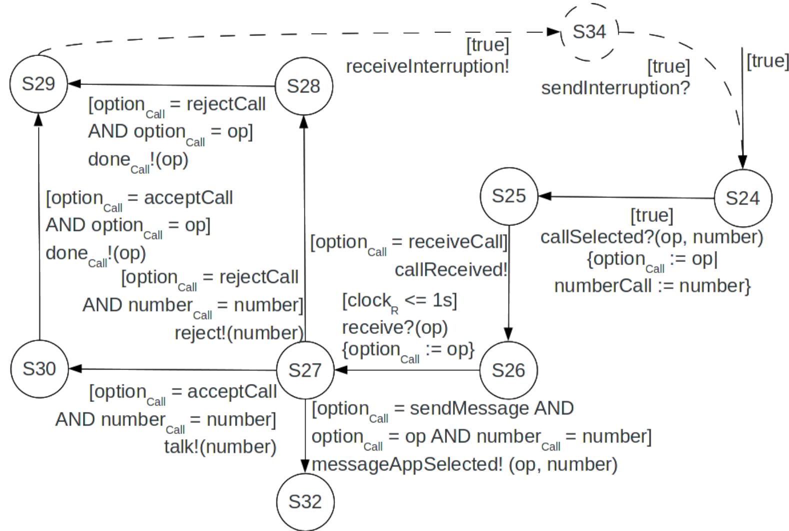


Figure 5.11: *Receive Call* subsystem

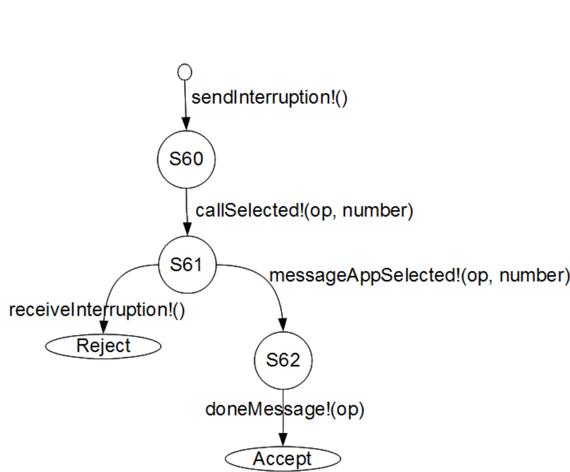


Figure 5.12: Test purpose (a)

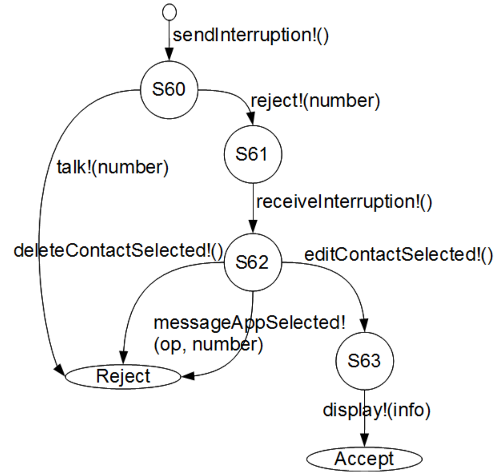


Figure 5.13: Test purpose (b)

test cases in our site. For each test purpose, the *Reject* location leads to situations we want to ignore and the *Accept* location indicates that test cases following that path shall be generated. For example, the test purpose (a) from Figure 5.12 leads the generation of test cases that contain the sequence of output actions *sendInterruption!*, *callSelected!(op, number)*, *messageAppSelected!(op, number)* and *doneMessage!(op)*. On the other hand, this same test purpose ignores the generation of test cases that comprise the sequence of output actions *sendInterruption!*, *callSelected!* and *receiveInterruption!*. In this way, we test the scenario where the system lets the user accept the call and send a message to this calling number.

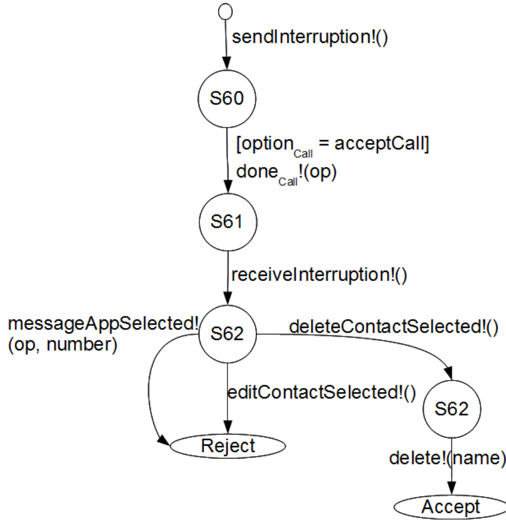


Figure 5.14: Test purpose (c)

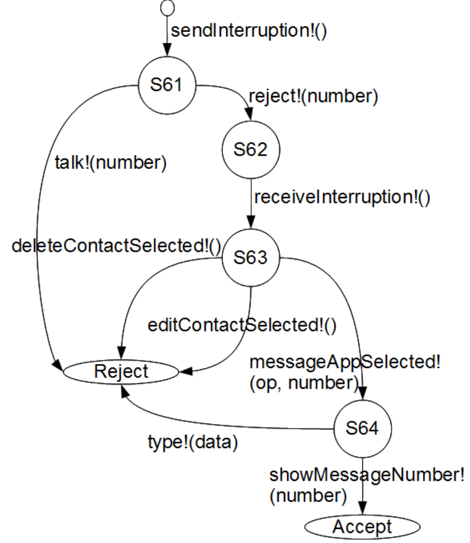


Figure 5.15: Test purpose (d)

The test purpose (b) ignores each test case that leads to the sequences: i) *sendInterruption!* and *talk!(number)*; ii) *sendInterruption!*, *reject!(number)*, *receiveInterruption!* and *deleteContactSeleted!*; iii) *sendInterruption!*, *reject!(number)*, *receiveInterruption!* and *messageAppSelected!(op, number)*. Meanwhile, this test purpose guides the generation of test cases that contain the sequence *sendInterruption!*, *reject!(number)*, *receiveInterruption!*, *editContactSeleted!* and *display!(info)*. Hence, we test scenarios where the user rejects the call and edits the contact after that. Figure 5.13 shows the corresponding TIOSTS model.

We show test purpose (c) in Figure 5.14 that guides the generation of test cases where the user accepts a call and deletes a contact, in this order. It allows the generation of test cases that comprise the sequence of actions *sendInterruption!*, $done_{call}$ when the option acceptCall is selected, *receiveInterruption!*, *deleteContactSeleted!* and *delete!(name)*. It also ignores two sequences of actions: i) *sendInterruption!*, $done_{call}$ when the option acceptCall is selected, *receiveInterruption!* and *messageAppSelected!(op, number)*; and ii) *sendInterruption!*, $done_{call}$ when the option acceptCall is selected, *receiveInterruption!* and *editContactSeleted!*.

The test purpose (d) allow the generation of test cases where the user rejects a call and send a message to that number after that (Figure 5.16). This TIOSTS guides the generation of test cases that contain the sequence *sendInterruption!*, *re-*

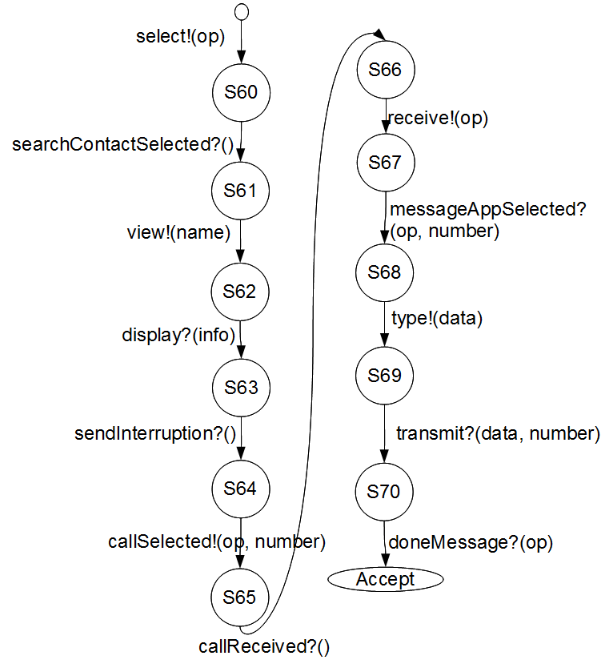


Figure 5.16: Test case 0 from test purpose (a)

ject!(number), *receiveInterruption!*, *messageAppSelected!(op, number)* and *showMessageNumber*. Meanwhile, it ignores test case generation that contain four sequences of actions.

We cannot show complete test case examples because of the space required for its presentation. However, we present a partial description of test case 0 generated from test purpose (a). Inputs were changed to outputs and vice-versa to reflect inputs and output from the tester point of view. The full model of this test case is available at our site.

We present the identification of each test purpose (used in our site) along with the number of generated test cases and the computation cost in Table 5.2. In the first scenario, the SYMBOLRT tool generates 12 test cases in 44s. The second scenario returns 12 test cases in 38s. The third scenario generates 12 test cases in 37s. Finally, SYMBORLT tool generates 4 test cases in 9s for the fourth scenario.

We implement the system in the Android platform. Moreover, we automate the execution of test cases by using the Junit framework [Fra14a] and the Robotium framework [Fra14b] under the Eclipse platform. During test case execution, from a startup activity, Contact and Receive Call activities start. Junit assertions, along with usual

Table 5.2: Test purposes and generation time of test cases

Test Purpose	Site Identifier	# Test Cases	Spent Time
(a)	sendInterruptionDoneMessage	4	$9s$
(b)	interruptionRejectDelete	12	$37s$
(c)	interruptionEditDisplay	12	$38s$
(d)	interruptionShowMessageNumber	12	$44s$

conditions checking, checks also if the correct activity starts at the expected time of composition. Robotium captures user clicks and searches for strings in the screen to capture message and context changing between applications. Robotium also simulates phone call receiving.

5.3 Concluding Remarks

We applied the sequential, parallel and interruption operators through realistic examples. The *Avionics* system combines subsystems through the sequential and parallel operators. The *Cell Phone* system composes subsystems using the interruption and sequential operators. Along with test cases generated, we introduced hints on the selection of test cases for these systems and exemplified how we use them in complex applications.

These exploratory studies intended to evaluate the applicability of our integration testing approach to real-time systems. We showed results towards the generation of integration test cases. The time used to generate test cases increases with the number of transitions and locations of a composed model. In addition, big test cases last longer to be generated than small ones.

Chapter 6

Related Works

To base our research, we performed a systematic review on compositional models [Dam11] and identified few works focusing on the conformance testing for compositional real-time systems. As far as we know, there is no study that aims to generate test cases from compositional and timed models. Since these systems comprise symbolic timed requirements, the Section 6.2 included conformance testing of untimed and compositional models.

Moreover, because we did not identify works that focus on symbolic untimed systems composed by interruptions, we focus on works that do not take composition into account but address the outcomes of interruptions in reactive systems (Section 6.1).

6.1 Model-based Interruption Testing

Andrade *et al.* published some papers towards model-based testing of reactive systems. Their main feature is that they react to external events. In their first work, Andrade and Machado [AM08] propose a symbolic transition system that stores inputs and outputs. The Input/Output Symbolic Transition Systems (IOSTS) is composed by the following sets: input, output and internal actions; locations; typed variables; and parameters that are communicated through actions.

The test process is guided by test purposes that have the main function of reducing the generation of the test cases. Test cases are IOSTS models that leads execution to verdicts, which assume the values *Pass*, *Fail* or *Inconclusive*. These verdicts have the

same meaning as in our work.

Also, they use conformance testing to verify if a specification is in accordance to an implementation by executing test cases. They name their conformance relation as ioco. In this way, an implementation is in conformance to a specification if for all traces of the specification, the set of output actions of the specification contains the set of output actions of the implementation. Interrupt-driven reactive systems is achieved by adding actions with the label "interrupt" in a test purpose to generate test cases. This approach was used with the STG tool to perform a case study that uses a subsystem of a cell phone application.

In addition, Andrade and Machado [AM12; AM09] present the Annotated Labeled Transition System (ALTS). It is a kind of LTS with specific labels called annotations. These labels delimits the start and the end of an interruption. The evolution of this work is threefold: i) presentation of algorithms to translate high level specifications into the ALTS models, ii) instantiation of a CSP formal model to ALTS test cases, and iii) presentation of a detailed case study.

Another extension of Andrade's work [AMAA09] present the TIOSTS model, that adds clocks to IOSTS. In the sequence, they adapt their strategy to comprise the notion of time, including the definition of test cases and the tioco conformance relation. They generate, implement and run test cases by using a real-time operating system. Additionally, they present some rules to transform the sequence, component and the state machine diagrams from UML to TIOSTS models.

The interruption testing approach used by these works start from an untimed model which evolves to a timed symbolic transition system. If we take the timed model into account, Andrade's work differs from our work because their model include a set of internal actions. Also, subsystems cannot use an operator. Their interrupt-driven strategy is based on rules that represent interruptions in a single model. In spite of our work, they do not study compositional issues and consequences about the conformance relations on interruptions.

6.2 Model-based Compositional Testing

We split compositional model-based testing approach in untimed (Sections 6.2.1 to 6.2.6) and timed (Sections 6.2.7 to 6.2.9).

6.2.1 Bijl *et al.*

Bijl *et al.* [BRT04] work on compositional properties for conformance testing of untimed systems. They use the Input-Output Transition System (IOTS), which is a input-complete LTS. This model considers internal actions by using the τ action. Also, it allows quiescence, meaning that it contains states where no outputs are enabled and the system is forced to wait until an input is provided.

The composition of IOTS is performed by using the hiding and parallel operators. The hiding operator replaces each action from a predefined set by the τ action. A consequence from this composition is that hidden actions cannot be seen by subsystems which are outside the composition. Oh the other hand, the parallel operator follows the same line of reasoning of other common operators, where uncommon action to both systems are interleaved by a Cartesian product and common actions are synchronized, resulting in an output action in the composed system. They restrict the operator to a binary parallel composition, forcing the correspondence between synchronizing actions to happen in pairs. Initial results show that even if subsystems are livelock-free, the composed system may not be.

The authors present the ioco conformance relation that is suitable for IOTS, proving some properties for the compositional operators. They point out that the hiding and parallel operators do not preserve the conformance relation from subsystems to the composed system because ioco allows underspecification of inputs. Nevertheless, this limitation is surpassed by restricting subsystems in two ways: i) use an input-complete process to avoid underspecification of inputs (named demonic completion) or ii) define a conformance relation weaker than ioco to preserve conformance from subsystems to the composed system. Following the first idea, they suggest an input-complete process that leads non-specified inputs to states that do not belong to the original subsystems. Once that the underspecification of inputs is avoided, the ioco conformance relation is

preserved from subsystems to the composed system.

From this perspective, they conclude that inferring ioco conformance from subsystems is suitable to prove properties on the composed systems, reducing efforts and costs. They suggest that systems can be composed by the parallel operator, and a second step is to apply the hiding operator to isolate actions from other systems. The hiding of actions delimits system components, which simplifies the testing process of complex systems.

When compared to our work, Bijl *et al.* present a model that is not symbolic and does not store time requirements, so the state explosion problem remains unsolved and real-time systems cannot be tested. Although they contribute with a testing approach, they do not implement it in a tool. However, their approach can be adapted to ours by creating a hiding operator to TIOSTS models.

6.2.2 Daca *et al.*

Daca *et al.* [DHKN14] propose a new approach to preserve the ioco conformance relation from subsystems to the composed system in the context of IOLTS systems. They use the IOLTS model, which is an LTS that comprises internal actions, quiescence and input and output actions.

They define new versions of the parallel and hiding operators inspired by contract-based design and interface theories. These operators are suitable for subsystems that allow underspecification of input actions and work by suppressing inputs and outputs that lead to incompatible interactions between subsystems. With this, they show that the their operators preserves the ioco conformance relation from subsystems to the composed system. Besides that, they formalize their approach, applied in a case study and compare with the demonic input-completion proposed by Bijl *et al.* [BRT04]. Results show that they generated composed systems with fewer transitions and states, lessening the effort to generate test cases.

When compared to our approach, Daca *et al.* work present similar issues to Bijl *et al.*. The restriction of the untimed and non-symbolic model limits its usage to real-time systems with the state explosion problem. However, we believe that their approach to preserve conformance from subsystems to the composed can be adapted to our strategy.

6.2.3 Sampaio *et al.*

Sampaio *et al.* [SNM09] use the CSP language that combines processes by using compositional operators. They establish compositional properties for the cspio conformance relation based on traces model of the CSP language. However, if we take semantics into consideration, input and output actions do not differ.

The authors define the I/O process as the basic element of a new algebra based on CSP. They compare the ioco conformance relation presented in [BRT04] and cspio. They map LTS to CSP processes and show that ioco is equivalent to cspio. The mapping preserves LTS traces and quiescent states. In the sequence, they define the I/O parallel, I/O hiding and I/O choice operators that belong to this algebra and define trace semantics of some CSP operators. Finally, they prove that I/O operators preserve the cspio conformance relation from input-complete subsystems to the composed system.

An extension of this work is presented in [SNMI14]. They define the I/O hiding operator and state general compositionality properties for each operator by requiring input-completeness of composites. Regarding the I/O choice operator, they relax this property when an implementation conforms to a set of partial specifications. In addition, they mechanize proofs by using and automatic theorem prover. At last, they evaluate test cases effectiveness by using a cell phone application and fault-based testing. They show that the ioco and cspio approaches achieve similar performance in terms of time.

Because Sampaio *et al.*'s work focuses on the ioco conformance relation, we observe some similarities to our work. They define a set of compositional operators for I/O processes, including the parallel operator. Yet, they relate their theory to conformance testing, use mechanized proofs and demonstrate practical implications of their theory. Nevertheless, our operators are suitable for timed symbolic models. Since they use CSP refinement to test systems and we generate test cases from models, we consider both works different.

6.2.4 Briones

Most authors require that subsystems be input-enabled to infer ioco conformance from subsystems to the composed system. Nevertheless, Briones [Bri10] presents a solution which focuses on assume-guarantee reasoning. Knowing that subsystems follow a pattern, this work uses the divide and conquer approach to conclude some behavior on the composed system.

The author gives assumptions on the parallel and hiding operators when used with IOTS models. However, if subsystems are strongly convergent (they do not have infinite sequences of actions), the composed system may not be. Regarding the parallel operator, given four IOTS models A, S, i_1 and i_2 under some restrictions, if $i_1 \parallel A$ ioco $S \wedge i_2$ ioco A, then $i_1 \parallel i_2$ ioco S. As a consequence, if A is provided, we can test i_1 and i_2 in isolation and assume that $i_1 \parallel i_2$ ioco S. Also, considering a set of actions V and *hide* to be the hiding operator, if *hide* V *in* $i_1 \parallel A$ ioco $S \wedge i_2$ ioco A, then *hide* V *in* $i_1 \parallel i_1$. This allow the testing of interfaces between components without performing big changes in the system.

When compared to our work, Briones uses a different conformance relation that does not comprise time requirements and symbolic models. Although she considers a case study, she does not implement her approach in a tool and does not show test cases.

6.2.5 Aiguier *et al.*

Aiguier *et al.* [ABK12] model software components which are independent of any computational structure. They use a type of algebra to model each one, comprising determinism and non-determinism. Moreover, they define two integration operators: Cartesian product and feedback, that can be used to build other systems by composition.

The Cartesian product operator follows the same reasoning of other operators from Mathematics. It generates a composition where components are executed simultaneously when they are matched in pairs. The feedback operator is a compositional operator where inputs and outputs from components are linked. This link can be

simultaneous or not, splitting this operator in two kinds. The first is the relaxed feedback, happening when a previous input depends on a current input. The second one is the synchronous feedback that matches inputs when both are available.

These two integration operators compose more complex operators: sequential, synchronous product and concurrent composition. The sequential operator connects two components sequentially disposed and the second one needs the outputs of the first to begin. The synchronous product results in a composition where components can be executed independently or jointly and linked by input and output actions. The concurrent composition adds the execution of other component after the synchronization is performed.

Besides defining components and operators, the authors propose the cioco conformance relation that is suitable for components and is based on ioco. They show that cioco is preserved from subsystems to the composed system. However, the input-completeness of specifications is need for the feedback operator. They define a test framework to be used with components and cioco which is guided by test purposes. Finally, they present detailed proofs towards their results.

Despite defining a testing framework, the authors do not implement their theory in a tool. In addition, they do not show test cases generated from models. They also defined the sequential operator and variations of the parallel operator, but no definition of the interruption operator is found. In spite of our work, they are based on components that do not handle symbolic data, focusing on a different scope.

6.2.6 Faivre *et al.*

Faivre et *al.* [FGG07] proposes an approach to deal with component-based specifications modeled by the Input/Output Symbolic Transition Systems (IOSTS). This model embodies quiescence and internal actions. In addition, they present a conformance testing theory for components with the hiding and renaming operators.

The authors show how to use components in a ioco conformance testing process that uses symbolic execution to compute the behavior of subsystems and test purposes to narrow test generation. Besides *Fail*, *Pass* and *Inconclusive*, they add the *weakpass* verdict that means that the specification behavior belongs to the test purpose and

to a path which is not in the test purpose. This theory is exemplified in a system representing a slot machine.

This work uses symbolic models, but time requirements are not taken into account because the ioco conformance relation is used. Besides, to the best of our knowledge, the authors do not present proofs nor implemented a tool to be used with their approach.

6.2.7 Olderog and Swaminathan

Olderog and Swaminathan [OS10] studied distributed real-time systems that are executed in multiple platforms and each action depend one of another within time. To model system behavior, they use a version of Timed Automata. This model stores sets of clocks, locations, actions and transitions. Besides, the model embraces sets of invariants, which are mappings from locations to zone clocks that have the upper bounds $\leq$ and $<$.

In order to compose subsystems, the authors define three kinds of compositional operators: sequential, parallel and layered. The sequential and parallel operators follows the CSP style, so the sequential operator allows the execution of actions from the first model and in the sequence includes actions from the second one. The parallel operator lets common actions to both subsystems synchronize and disjoint actions follows a Cartesian product, leading to interleaving.

The layered operator modifies the parallel operator by including the concept of independent actions during interleaving. Independent actions fill the enabledness and commutativity conditions. Commutativity means that two different execution order of actions lead to the same state of the distributed system. Enabledness implies that one action does not prevent the execution of another. Each set of independent actions composes a layer, and the next layer is executed after the current one finishes. Besides synchronizing actions, the layered operator takes dependent and independent actions into account.

The authors use a real-time system that specifies an audio/video collision avoidance protocol to perform a case study by using the parallel and sequential operators. They use the UPPALL tool to model the protocol as a network of timed automata, resulting in a model with 7000 locations. They reduce the number of locations from the model

in 300 locations by using the operators and the approach they develop.

This work is similar to our work because it defines the sequential and parallel compositional operators for models that stores time. Nevertheless, they have a different scope, since they focus on distributed systems and the shared memory paradigm. Also, their model do not store data and do not allow test case generation from it. Besides, it obligates the usage of a final location, which may be inconvenient to some systems as there are distributed systems that continually executes.

One advantage is that their version of the sequential operator does not constrain subsystems to fit a normal form. This happens because the final location of the first subsystem and the first location of the second subsystem are replaced by a new location in the composed model that links the two subsystems. On the other hand, the clocks of the second system are not reset during the composition, changing the behavior of the second subsystem, since the composed system force the overall clocks to start functioning from the beginning of the first subsystem.

6.2.8 Krichen and Tripakis

Krichen and Tripakis [KT06] use the TAIO (Timed Automata with Inputs and Outputs) to study the conformance testing of timed systems. This model embraces sets of: states, clocks, inputs actions, output actions and transitions.

This work aims at presenting some characteristics of the tioco conformance relation. Among many properties, the authors show that tioco is compositional under the parallel and hiding operators if specifications are input-complete and the intersection between clocks is empty.

We understand that our work has a practical appeal when compared to Krichen and Tripakis's work because we develop case studies and an approach implemented in a tool. In addition, the TAIO model derives from finite automata, while the TIOSTS model derives from Transition Systems [AD94; Mil99; BK+08]. Transition systems characterize the notion of observation and interaction. They have syntax, that give support for modeling, and semantics, which bases calculation. When compared to finite automata, transition systems: i) do not contain a set of final states; ii) may have a countable and finite set of actions and states; iii) may contain an infinite set of

transitions and iv) have the set of actions that may be subject to synchronization.

6.2.9 Bannour *et al.*

Bannour *et al.* [BGAL13] study compositionality properties for the tioco conformance relation by using the parallel operator. They use TIOLTS, a timed version that extends the LTS model. Besides, they include the notion of *durations*, an isomorphic set to strictly positive real numbers that represent clock values. In this sense, this model is a labeled transition system over actions and *durations*.

The compositionality of tioco is studied under a version of the parallel operator that synchronize on common actions and perform a Cartesian product on uncommon actions. It requires that the intersection between the set of input actions from both models be empty, as well as the set of output actions. Besides, the set of durations must be common to them.

The local consistency of a composed system $\mathcal{S}_1 \parallel \mathcal{S}_2$ states that if there are traces which are similar in the subsystems, the system will continue these traces with the same input and outputs, or allow for the same amounts of time to elapse. Assuming two implementations $\mathcal{I}_1$ and $\mathcal{I}_2$, omitting some terms about time requirements from the original definition and the local consistency of $\mathcal{S}_1 \parallel \mathcal{S}_2$, the authors assures that $\mathcal{I}_1$ tioco $(\mathcal{S}_1 \parallel \mathcal{S}_2) \wedge \mathcal{I}_2$ tioco $(\mathcal{S}_1 \parallel \mathcal{S}_2) \Rightarrow (\mathcal{I}_1 \parallel \mathcal{I}_2)$ tioco $(\mathcal{S}_1 \parallel \mathcal{S}_2)$. Besides, if $(\mathcal{I}_1 \parallel \mathcal{I}_2)$ $\neg\mathsf{tioco}$ $(\mathcal{S}_1 \parallel \mathcal{S})$, it is assumed that a subsystem implementation is not tioco conformant to its composed specification.

An algorithm to check local consistency property is defined and the theory is illustrated in an example. Also, the authors present some steps towards symbolic execution of compositional systems and how this algorithm can be used to check local consistency.

The TIOSTS model from Bannour work is similar to our version, but they add the τ internal action. However, we did not identify the usage of parameters that communicate variable values between subsystem, implying that they use data variables to perform this task. Their assumption about preservation of the tioco conformance relation for the parallel operator is different from ours because they need a common set of clocks between subsystems. Besides, there is no work towards the sequential and interruption operator. Finally, we did not identify the usage of a tool or test cases generated from

their approach.

6.3 Concluding Remarks

Model-based testing of compositional untimed systems has been extensively addressed in the literature. Most works intend to verify conformance of the composed implementation by assuring properties on the subsystems. The main advantage of this approach is to avoid building the composed system, lessening costs and computational effort.

Table 6.1 shows a summary of related works. We compare them according to the following items: model, conformance relation, compositional operator(s) and if the approach uses or not a tool. Most works use the ioco conformance relation, symbolic models, and few of them use tools integrated to their approach. Moreover, all the works that make use of compositional operators also adopt versions of the parallel operator.

Frequently, the preservation of conformance relations from subsystems to the composed system is assumed to be correct under restricted conditions. Some works assume specifications to be input-complete to avoid a common problem of conformance relations based on ioco: the underspecification of inputs, leading to an unpredictable behavior. The solution to this problem is threefold: i) the specification is changed to an input-complete process, ii) inputs are pruned or iii) subsystems must fill some conditions to be part of a compositional system.

Some research has been devoted to model-based testing of compositional real-time systems. Nevertheless, since tioco is an extension of ioco, the underspecification of inputs is a problem from tioco which is inherited from ioco. Hence, applications of these proposed solutions in the tioco theory needs further investigation.

To the best of our knowledge, there are three works that study compositional properties of the tioco conformance relation: Olderog and Swaminathan, Bannour *et al.* and Krichen and Tripakis. All of them use the parallel operator to study the outcomes of tioco under compositionaly, but they used different pre-conditions for subsystems. While Krichen and Tripakis and Olderog and Swaminathan's work require the set of clocks from both subsystems to be empty, Bannour *et al.* assume the same set of clocks.

Table 6.1: Summary of works

Work	Model	Conformance relation	Compositional operator(s)	Tool
Andrade *et al.*	IOSTS	ioco	No	Yes
Bijl *et al.*	IOTS	ioco	Parallel and hiding	No
Daca *et al.*	IOLTS	ioco	Parallel and hiding	No
Sampaio *et al.*	I/O process	cspio	Parallel, hiding and choice	Yes
Briones	IOTS	ioco	Parallel and hiding	No
Aiguier *et al.*	Components	cioco	Feedback and Cartesian product	No
Faivre *et al.*	IOSTS	ioco	Hiding and renaming	No
Olderog and Swaminathan	TA	tioco	Sequential, parallel, layered	No
Krichen and Tripakis	TAIO	tioco	Parallel and hiding	No
Bannour *et al.*	TIOSTS	tioco	Parallel	No

Chapter 7

Concluding Remarks

This chapter sums up our main contributions (Section 7.1) and suggests future works (Section 7.2).

7.1 Main Results

The main goal of this research is to provide a framework for composing timed and symbolic models that represent real-time systems. The composed specifications are used to generate test cases. Subsystems use the TIOSTS model, which is a symbolic transition system that stores data and clock information [AMJM11]. To compare implementations and specifications, we use the tioco conformance relation. We define the sequencial, interruption and parallel operators and restrict our study to real-time systems that use the message-passing communication paradigm. Moreover, we infer compositional properties on these operators and provide an integration testing strategy suitable to the composed systems.

In this context, we answer the research questions from Section 1.2 by using the results presented in Chapters 3 and 5 as follows:

Research Question 1 *How symbolic models of real-time systems that abstract data and time can be composed?*

We presented the parallel, sequential and interruption operators for the TIOSTS model and introduced them by examples. The sequential operator resembles system

level composition of subsystems that represent independent activities where one subsystem uses a final result produced by another. The parallel operator can be applied to compose subsystems whose execution is independent (in terms of resource sharing) but communication is required based on synchronization of input-output actions. The interruption operator is used when the interrupted system resumes its execution after interruption handling finishes.

Research Question 2 *What are the main challenges to infer conformance of the composed system based on conformance of composites?*

We used the tioco conformance relation to study implications of conformance from subsystems to the composed systems. The preservation of the conformance relation depends on the compositional operator. The sequential and interruption operators preserve the conformance relation as long as subsystems fill predefined normal forms. Conversely, the parallel operator preserves the conformance relation if subsystems are input-complete and implementations and specifications have the same set of outputs.

Research Question 3 *How can integration test cases be generated from composed models?*

We use the test generation process described by Andrade *et al* [AM13] to generate test cases. It requires two main inputs: a specification and a test purpose. We generate specifications by using our compositional operators and giving advices on how to choose test purposes that lead to integration test cases.

Since we are restricting systems to the message-passing paradigm, we identified subsystems interactions by determining the synchronizing actions they have in common. So, if an action from different subsystems has the same label and conjugated input/output action types, we assume that these subsystems synchronize on them. Hence, these actions are used to build the test purposes that guide the generation of the interaction test cases.

Unfortunately, the tioco conformance relation is not preserved from its isolated subsystems to the composed system. Nevertheless, this does not mean that composition is useless, since the purposes of composition for testing are twofold: inferring conformity of the composed implementation with regard to the composed specification and

generating integration test cases. The latter can be always performed in spite of the former.

And so we performed composition of subsystems to be used in a test case generation process. This work provides fundamental background towards an approach to test compositional real-time systems. It contributes to improve the applicability of the TIOSTS model in the context of integration testing, focusing on interruption, sequential and parallel compositions. In addition, it defines compositional properties of the tioco conformance relation from subsystems to the composed system. Finally, it presents an input-completion approach that may be applied to subsystems when the tioco conformance relation is inferred from subsystems to the composed systems.

Our work has some limitations. First, we do not use test purposes with time requirements in our test generation strategy because of some limitations the temporal properties may provide. Second, we do not offer a declarative notation for the compositional operator to be used with the tool because of lack of specialized people to give code support during the development of this work. In addition, our tool is restricted to deterministic real-time systems because we use an offline approach to generate test cases, limiting the algorithms used in the test case generation process [HLM+08]. Finally, our composition scope is restricted to two subsystems because the operators we propose are binary, allowing only two subsystems to be composed.

7.2 Future Works

After presenting our contributions, we point out some future works:

Extension of our compositional framework If we take into account compositional approaches from conventional language specifications, we can define other compositional operators to be used with the TIOSTS model. First, we suggest the addition of the renaming operator, which is responsible for replacing a set of actions by others. The second may be the choice operator, allowing two subsystems to be composed by more branches while corresponding each one to a single action. The third is the if-else operator that resembles to a boolean conditional command from classic programming languages. The fourth is the timeout oper-

ator that interrupts a subsystem depending on which time is passed, in contrast to the interruption operator that interrupts a subsystem if an action is executed. Finally, the hiding operator may be added to this framework in order to allow some actions to be internal to a subsystem. However, before adding this operator, we need to modify the TIOSTS definition to include the internal action τ.

Solutions to surpass non-conformity We indicate the adoption of approaches to overcome the non-conformance problem that tioco presents for some compositional operators [KT06; BGAL13], allowing compositions to be more widely applicable. Alternatively to using the input-completion process presented in Section 3.3, we suggest the adaptation of approaches presented for the ioco conformance relation to our work, like the definition of a weaker conformance relation (presented in [BRT04]) or the use of friendly environments (defined in [DHKN14]).

Perform further exploratory studies We suggest the application of the operators and the testing approach to more exploratory studies in order to identify limitations and outcomes.

Implementation of a testing architecture Since we provide the generation of models that represent test cases, we suggest the development of a testing architecture under the Android platform [Dev14] that supports compositional test case implementation and execution. Along with this work, we suggest the definition of transformation rules from TIOSTS to commands from the Android platform and the development of frameworks devoted to the application of these operators in other practical software development environments.

Detailed proofs We suggest to detail proofs by using an automated theorem prover to guarantee proofs validity.

Bibliography

[98914a] ISO/IEC 9899:1990. http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?csnumber=17782, 2014. Accessed: 12-15.

[98914b] ISO/IEC 9899:2011. http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=57853, 2014. Accessed: 12-15.

[AACM12] Wilkerson L. Andrade, Diego R. Almeida, Jeanderson B. Cândido, and Patrícia D. L. Machado. SYMBOLRT: A Tool for Symbolic Model-Based Test Case Generation for Real-Time Systems. In *19th Tools Session of the 3rd Brazilian Conference on Software: Theory and Practice (CBSoft 2012)*, pages 31–37, 2012. Best Tool Award.

[ABK12] Marc Aiguier, Frédéric Boulanger, and Bilal Kanso. A formal abstract framework for modelling and testing complex software systems. *Theoretical Computer Science*, 455:66–97, October 2012.

[AD94] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

[AM08] Wilkerson Lucena Andrade and Patricia Duarte Lima Machado. Modeling and testing interruptions in reactive systems using symbolic models. *SAST*, 8:34–43, 2008.

[AM09] Wilkerson Lucena Andrade and Patricia Duarte Lima Machado. Interruption Testing of Reactive Systems. *Formal Methods: Foundations and Applications*, pages 37–53, 2009.

[AM12] Wilkerson Lucena Andrade and Patricia Duarte Lima Machado. Testing interruptions in reactive systems. *Formal Aspects of Computing*, 24(3):331–353, 2012.

[AM13] W.L. Andrade and P.D.L. Machado. Generating test cases for real-time systems based on symbolic models. *Software Engineering, IEEE Transactions on*, 39(9):1216–1229, Sept 2013.

[AMAA09] Wilkerson Lucena Andrade, Patricia Duarte Lima Machado, Everton Leandro Galdino Alves, and Diego Rodrigues Almeida. Test case generation of embedded real-time systems with interruptions for FreeRTOS. In *Formal Methods: Foundations and Applications*, pages 54–69. Springer, 2009.

[AMJM11] Wilkerson L. Andrade, Patricia D. L. Machado, Thierry Jéron, and Herve Marchand. Abstracting time and data for conformance testing of real-time systems. In *Proceedings of the 2011 IEEE ICST Workshops*, pages 9–17, Washington, DC, USA, 2011. IEEE.

[BBJ02] George Boolos, John P Burgess, and Richard C Jeffrey. *Computability and logic*. Cambridge university press, 2002.

[BCDK12] G Blair, G Coulouris, J Dollimore, and T Kindberg. *Distributed Systems: Concepts and Design*. Boston: Addison-Wesley, 2012.

[bG14] Language Standards Supported by GCC. https://gcc.gnu.org/onlinedocs/gcc/Standards.html, 2014. Accessed: 12-15.

[BGAL13] B. Bannour, C. Gaston, M. Aiguier, and A. Lapitre. Results for compositional timed testing. In *Software Engineering Conference (APSEC, 2013 20th Asia-Pacific*, volume 1, pages 559–564, Dec 2013.

[Bin00] Robert Binder. *Testing Object-oriented Software Testing: Models, Patterns, and Tools*. Addison-Wesley Professional, 2000.

[BJSK11] Nathalie Bertrand, Thierry Jéron, Amélie Stainer, and Moez Krichen. Off-line test selection with test purposes for non-deterministic timed au-

tomata. In *Proceedings of TACAS'11/ETAPS'11*, pages 96–111. Springer-Verlag, 2011.

[BK+08] Christel Baier, Joost-Pieter Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.

[Bri10] L B Briones. Assume-guarantee reasoning with ioco testing relation. *on Testing Software and Systems: Short Papers*, pages 103–108, 2010.

[BRT04] Machiel Bijl, Arend Rensink, and Jan Tretmans. Compositional testing with ioco. In Alexandre Petrenko and Andreas Ulrich, editors, *Formal Approaches to Software Testing*, volume 2931 of *Lecture Notes in Computer Science*, pages 86–100. Springer Berlin Heidelberg, 2004.

[BY04] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. *Lectures on Concurrency and Petri Nets*, pages 87–124, 2004.

[CT05] Richard H Carver and Kuo-Chung Tai. *Modern multithreading: implementing, testing, and debugging multithreaded Java and C++/Pthreads/Win32 programs.* John Wiley & Sons, 2005.

[Dam11] Adriana Carla Damasceno. A Systematic Review on Compositional Models. Technical Report SPLAB-2011-003, Available at http://splab.computacao.ufcg.edu.br/publications/technical-reports, 2011.

[Dev14] Android Developers. http://developer.android.com/, 2014. Accessed: 2014-06-04.

[DFM09] Adriana Damasceno, Adalberto Farias, and Alexandre Mota. A mechanized strategy for safe abstraction of csp specifications. In *Formal Methods: Foundations and Applications*, pages 118–133. Springer, 2009.

[DHKN14] Przemyslaw Daca, Thomas A Henzinger, Willibald Krenn, and Dejan Ničković. Compositional specifications for ioco testing. In *Proceedings of IEEE International Conference on Software Testing, Verification and Validation*, volume 7, pages 373–382. IEEE, 2014.

[DMJ07] Márcio Eduardo Delamaro, José Carlos Maldonado, and Mario Jino. *Introdução ao teste de software*. Elsevier, 2007.

[DNSVT07] A.C. Dias Neto, R. Subramanyan, M. Vieira, and G.H. Travassos. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*, pages 31–36. ACM, 2007.

[EPHJ07] Sigrid Eldh, Sasikumar Punnekkat, Hans Hansson, and Peter Jönsson. Component testing is not enough-a study of software faults in telecom middleware. In *Testing of Software and Communicating Systems*, pages 74–89. Springer, 2007.

[FGG07] Alain Faivre, Christophe Gaston, and Pascale Le Gall. Symbolic Model based Testing for Component Oriented Systems. *Testing of Software and Communicating Systems*, pages 90–106, 2007.

[Fra14a] JUnit Framework. http://junit.org/, 2014. Accessed: 2014-06-04.

[Fra14b] Robotium Framework. https://code.google.com/p/robotium/, 2014. Accessed: 2014-06-04.

[Gal04] Daniel Galin. *Software quality assurance: from theory to implementation*. Pearson education, 2004.

[Gar05] Angelo Gargantini. Conformance testing. In Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 87–111. Springer Berlin Heidelberg, 2005.

[Hal60] Paul Richard Halmos. *Naive set theory*. Springer Science & Business Media, 1960.

[HLM+08] Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using UPPAAL. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 77–117. Springer, 2008.

[IM04] WT Ingram and William Mahavier. Mathematics and computer science. 2004.

[Jec78] Thomas J Jech. *Set theory*, volume 79. Academic press, 1978.

[JP14] Standard Edition 8 API Specification Java Platform. `http://docs.oracle.com/javase/8/docs/api/`, 2014. Accessed: 2014-06-04.

[Kic14] Nico Kicillof. What is model-based testing?, October 2014. Available at `http://blogs.msdn.com/b/specexplorer/archive/2009/10/27/what-is-model-based-testing.aspx`.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[Kop11] H. Kopetz. *Real-time systems: design principles for distributed embedded applications*, volume 25. Springer, 2011.

[KT06] Moez Krichen and Stavros Tripakis. Interesting properties of the real-time conformance relation tioco. *Theoretical Aspects of Computing*, pages 317–331, 2006.

[KT09] M. Krichen and S. Tripakis. Conformance testing for real-time systems. *Formal Methods in System Design*, 34(3):238–304, 2009.

[Lap09] P.A. Laplante. *Real-Time Systems Design & Analysis*. Wiley-India, 2009.

[Li03] Jing Li. *Real time concepts for embedded systems*. Taylor & Francis US, 2003.

[LTX01] Louise Lorentsen, Antti-pekka Tuovinen, and Jianli Xu. *Modelling Feature Interactions in Mobile Phones*. FICS, 2001.

[LVLG90] C.D. Locke, D.R. Vogel, L. Lucas, and J.B. Goodenough. Generic avionics software specification. Technical report, DTIC Document, 1990.

[M+65] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[Mil99] Robin Milner. *Communicating and mobile systems: the pi calculus.* Cambridge university press, 1999.

[OS10] Ernst-Rüdiger Olderog and Mani Swaminathan. Layered composition for timed automata. In *Proceedings of the 8th international conference on Formal modeling and analysis of timed systems*, FORMATS'10, pages 228–242, Berlin, Heidelberg, 2010. Springer-Verlag.

[Pac11] Peter Pacheco. *An introduction to parallel programming.* Elsevier, 2011.

[PHL+11] Jan Peleska, Artur Honisch, Florian Lapschies, Helge Löding, Hermann Schmid, Peer Smuda, Elena Vorobev, and Cornelia Zahlten. A real-world benchmark model for testing concurrent real-time systems in the automotive domain. In Burkhart Wolff and Fatiha Zaïdi, editors, *Testing Software and Systems*, volume 7019 of *Lecture Notes in Computer Science*, pages 146–161. Springer Berlin Heidelberg, 2011.

[Sca14] SCARLETT Scalable and Reconfigurable Eletronics Platforms and Tools. http://www.scarlettproject.eu, 2014. Accessed: 2014-06-04.

[SNM09] A. Sampaio, S. Nogueira, and A. Mota. Compositional verification of input-output conformance via csp refinement checking. *Formal Methods and Software Engineering*, pages 20–48, 2009.

[SNMI14] Augusto Sampaio, Sidney Nogueira, Alexandre Mota, and Yoshinao Isobe. Sound and mechanised compositional verification of input-output conformance. *Software Testing, Verification and Reliability*, 24(4):289–319, 2014.

[Tan07] Andrew S Tanenbaum. *Modern operating systems.* Prentice Hall Press, 2007.

[TR11] Omer Nguena Timo and Antoine Rollet. Test selection for data-flow reactive systems based on observations. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–8. IEEE Computer Society, March 2011. 7th Workshop on Advances in Model Based Testing (A-MOST 2011).

[Tre99] J. Tretmans. Testing concurrent systems: A formal approach. *CONCUR'99 Concurrency Theory*, pages 779–779, 1999.

[UL07] Mark Utting and Bruno Legeard. *Practical Model Based Testing: A Tools Approach*. Elsevier, 2007.

[Val98] Antti Valmari. The state explosion problem. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer Berlin Heidelberg, 1998.

[Vea13] Margus Veanes. Applications of symbolic finite automata. In *Implementation and Application of Automata*, pages 16–23. Springer, 2013.

[vSBS10] Sabrina von Styp, Henrik Bohnenkamp, and Julien Schmaltz. A conformance testing relation for symbolic timed automata. In Krishnendu Chatterjee and Thomas Henzinger, editors, *Formal Modeling and Analysis of Timed Systems*, volume 6246 of *LNCS*, pages 243–255. Springer Berlin / Heidelberg, 2010.

[You08] Michal Young. *Software testing and analysis: process, principles, and techniques*. John Wiley & Sons, 2008.

[ZHHL11] Yongxin Zhao, Yanhong Huang, Jifeng He, and Si Liu. Formal Model of Interrupt Program from a Probabilistic Perspective. *2011 16th IEEE International Conference on Engineering of Complex Computer Systems*, 16:87–94, April 2011.

Appendix A

Proofs

We present proofs for theorems on compositionality properties of the tioco conformance relation and the sequential, interruption and parallel operators, introduced in Chapter 3. The style of our proofs follows a textual representation, which is based on Krichen and Tripakis [KT09], Bjil *et al.* [BRT04] and the general guidelines presented by [IM04].

Proofs are conducted on the numbered cases the operator may result when composing subsystems. These steps are strongly based on the normal forms and definitions of each operator (Chapter 3). Supporting operators and background definitions – for instance traces and TIOLTS definitions – are introduced in Section 2.3.2.2. Since toy examples were used to introduce normal forms and operators definition, we repeat them in each section and suggest the reader to follow proof steps based on them.

A.1 Theorem 2

In general terms, proof of Theorem 2 is divided in five steps: i) Assume that subsystem implementations are tioco conformant to their subsystem specifications, ii) Apply tioco definition Assumption i), iii) Correspond the TIOSTS model used in the sequential operator definition to the TIOLTS model used by the tioco definition, iv) Use previous steps to prove that every possible trace of the composed implementation is contained in the composed specification, using the normal form of the operator, which is exemplified in Figure A.1.

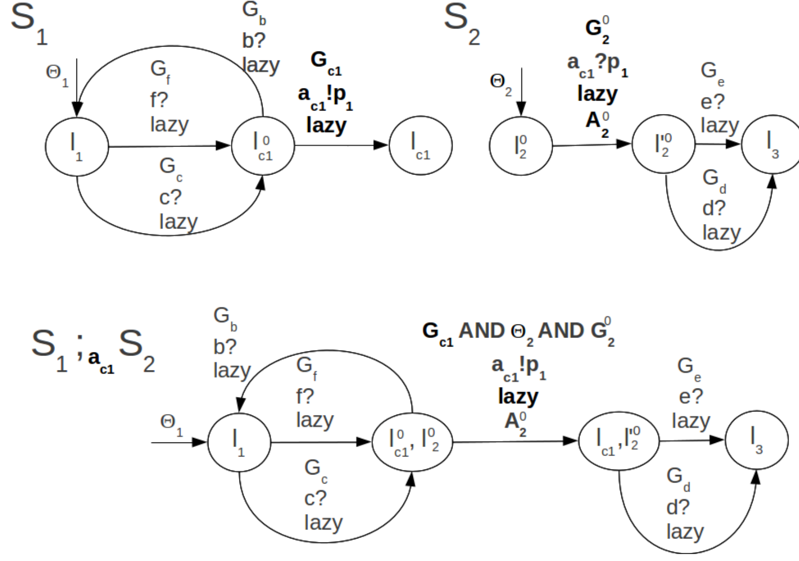


Figure A.1: Toy example for the sequential composition

Theorem 2 (tioco Sequential Composition)**.** *Let* $\mathcal{S}_1$ *and* $\mathcal{S}_2$ *be specifications and* $\mathcal{I}_1$*,* $\mathcal{I}_2$ *be implementations modeled by TIOSTSs that meet Definition 9. If* $\mathcal{I}_1$ tioco $\mathcal{S}_1 \wedge \mathcal{I}_2$ tioco $\mathcal{S}_2$ *then* $\mathcal{I}_1;_{a_{c1}} \mathcal{I}_2$ tioco $\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2$*.*

Proof. According to Definition 7, we need to prove that:

$\forall\ \sigma_1 \in \text{Traces}(\mathcal{S}_1)$: $\text{Out}(\mathcal{I}_1 \text{ after } \sigma_1) \subseteq \text{Out}(\mathcal{S}_1 \text{ after } \sigma_1) \wedge$

$\forall\ \sigma_2 \in \text{Traces}(\mathcal{S}_2)$: $\text{Out}(\mathcal{I}_2 \text{ after } \sigma_2) \subseteq \text{Out}(\mathcal{S}_2 \text{ after } \sigma_2) \Rightarrow$

$\forall\ \sigma \in \text{Traces}(\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2)$: $\text{Out}(\mathcal{I}_1;_{a_{c1}} \mathcal{I}_2 \text{ after } \sigma) \subseteq \text{Out}(\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2 \text{ after } \sigma)$

To correspond TIOLTS states (Definition 5) used by Traces to TIOSTS locations (Definition 4) used by the sequential operator and improve this proof readability, let TIOLTS $[\![\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2]\!] = \langle S, S^0, Act, T\rangle$ and $S_{(l^0_{c1}, l^0_2)} = \{\langle l, \nu, \psi\rangle | \langle l, \nu, \psi\rangle \in S \wedge \langle l, \nu, \psi\rangle \xrightarrow{\langle a,\gamma\rangle} \langle (l^0_{c1}, l^0_2), \nu', \psi'\rangle\}$. In addition, $\sigma, \rho \in \text{Traces}(\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2)$ and $\sigma = \rho \cdot a$. By Definition 6, $\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2 \xrightarrow{\rho \cdot a}$ and $\rho \cdot a \in \text{Traces}(\mathcal{S}_1\ ;_{a_{c1}}\ \mathcal{S}_2)$. From Definition 9, σ is fivefold:

i) $\sigma = \epsilon$

We replace σ by ϵ in Definition 7, resulting in $\text{Out}(\mathcal{I}_1 \text{ after } \sigma_1) \subseteq \text{Out}(\mathcal{S}_1 \text{ after } \sigma_1)$ $\wedge$ $\text{Out}(\mathcal{I}_2 \text{ after } \sigma_2) \subseteq \text{Out}(\mathcal{S}_2 \text{ after } \sigma_2) \Rightarrow \text{Out}(\mathcal{I}_1;_{a_{c1}} \mathcal{I}_2 \text{ after } \epsilon) \subseteq \text{Out}(\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2$ after $\epsilon)$. Since ϵ belongs to any set of traces, this trivially holds.

ii) $\sigma = \rho \cdot a$ with $a \in \Sigma_1\ \wedge s \notin S_{(l^0_{c1}, l^0_2)}$

We use (3.1) from Definition 9, resulting in $\rho = \sigma_1$ and $\sigma_1 \cdot a \in \text{Traces}(\mathcal{S}_1 \;;_{a_{c1}}\; \mathcal{S}_2)$. Because we assume that $\forall\ \sigma_1 \in \text{Traces}(\mathcal{S}_1)$: $\text{Out}(\mathcal{I}_1 \text{ after } \sigma_1) \subseteq \text{Out}(\mathcal{S}_1 \text{ after } \sigma_1)$, we have $\forall\ \sigma_1 \cdot a \in \text{Traces}(\mathcal{S}_1 \;;_{a_{c1}}\; \mathcal{S}_2)$: $\text{Out}(\mathcal{I}_1;_{a_{c1}} \mathcal{I}_2 \text{ after } \sigma_1 \cdot a) \subseteq \text{Out}(\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2$ after $\sigma_1 \cdot a)$ and, by Definition 7, $\mathcal{I}_1;_{a_{c1}} \mathcal{I}_2$ tioco $\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2$.

iii) $\sigma = \rho \cdot a$ with $a \in \Sigma_1 \ \wedge s \in S_{(l^0_{c1}, l^0_2)}$

We use (3.3) from Definition 9, resulting in $\rho = \sigma_1$, $a \neq a_{c1}$ and $\sigma_1 \cdot a \in \text{Traces}(\mathcal{S}_1 \;;_{a_{c1}}\; \mathcal{S}_2)$. Since we assume $\forall\ \sigma_1 \in \text{Traces}(\mathcal{S}_1)$: $\text{Out}(\mathcal{I}_1 \text{ after } \sigma_1) \subseteq \text{Out}(\mathcal{S}_1 \text{ after } \sigma_1)$ and $\mathcal{S}_1$ and $\mathcal{S}_2$ follow SC normal form from Definition 8, we have $\forall\ \sigma_1 \cdot a \in \text{Traces}(\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2)$: $(\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2 \text{ after } \sigma_1 \cdot a) = \{\langle l, \nu, \psi\rangle | l = (l^0_{c1}, l^0_2)\}$. Thus, $\forall\ \sigma_1 \cdot a \in \text{Traces}(\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2)$: $\text{Out}(\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2 \text{ after } \sigma_1 \cdot a) = \{\langle a, \gamma\rangle | a = a_{c1})\}$.

In addition, assuming that $\mathcal{I}_1$ and $\mathcal{I}_2$ follow SC normal form from Definition 8 and $\forall\ \sigma_1 \in \text{Traces}(\mathcal{S}_1)$: $\text{Out}(\mathcal{I}_1 \text{ after } \sigma_1) \subseteq \text{Out}(\mathcal{S}_1 \text{ after } \sigma_1)$, we have $\forall\ \sigma_1 \cdot a \in \text{Traces}(\mathcal{I}_1;_{a_{c1}} \mathcal{I}_2)$: $\text{Out}(\mathcal{I}_1;_{a_{c1}} \mathcal{I}_2 \text{ after } \sigma_1 \cdot a) = \{\langle a, \gamma\rangle | a = a_{c1})\}$. Finally, $\forall\ \sigma_1 \cdot a \in \text{Traces}(\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2)$: $\text{Out}(\mathcal{I}_1;_{a_{c1}} \mathcal{I}_2 \text{ after } \sigma_1 \cdot a) = \text{Out}(\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2 \text{ after } \sigma_1 \cdot a)$. Hence, $\forall\ \sigma_1 \cdot a \in \text{Traces}(\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2)$: $\text{Out}(\mathcal{I}_1;_{a_{c1}} \mathcal{I}_2 \text{ after } \sigma_1 \cdot a) \subseteq \text{Out}(\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2 \text{ after } \sigma_1 \cdot a)$ and, by Definition 7, $\mathcal{I}_1;_{a_{c1}} \mathcal{I}_2$ tioco $\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2$.

iv) $\sigma = \rho \cdot a$ with $a = a_{c1}$

We use (3.5) from Definition 9, resulting in $\rho = \sigma_1$ and $\sigma_1 \cdot a_{c1} \in \text{Traces}(\mathcal{S}_1 \;;_{a_{c1}}\; \mathcal{S}_2)$. Since we assume $\forall\ \sigma_1 \in \text{Traces}(\mathcal{S}_2)$: $\text{Out}(\mathcal{I}_2 \text{ after } \sigma_1) \subseteq \text{Out}(\mathcal{S}_2 \text{ after } \sigma_1)$ and $\mathcal{S}_1$ and $\mathcal{S}_2$ follow Definition 8, $\forall\ \sigma_1 \cdot a_{c1} \in \text{Traces}(\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2)$: $(\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2 \text{ after } \sigma_1 \cdot a_{c1}) = \{\langle l, \nu, \psi\rangle | l = (l_{c1}, l^{0'}_2)\}$. Because we use (3.4) and (3.5) from Definition 9, $\forall\ \sigma_1 \cdot a_{c1} \in \text{Traces}(\mathcal{S}_1 \;;_{a_{c1}}\; \mathcal{S}_2)$: $\text{Out}(\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2 \text{ after } \sigma_1 \cdot a_{c1}) = \text{Out}\ (\mathcal{S}_2 \text{ after } \overline{a_{c1}})$.

Moreover, assuming that $\text{Out}(\mathcal{I}_2 \text{ after } \sigma_1) \subseteq \text{Out}(\mathcal{S}_2 \text{ after } \sigma_1)$ and $\mathcal{I}_1$ and $\mathcal{I}_2$ follow Definition 8, we have $\text{Out}(\mathcal{I}_1;_{a_{c1}} \mathcal{I}_2 \text{ after } \sigma_1 \cdot a_{c1}) = \text{Out}\ (\mathcal{S}_2 \text{ after } \overline{a_{c1}})$. Thus, $\forall\ \sigma_1 \cdot a_{c1} \in \text{Traces}(\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2)$: $\text{Out}(\mathcal{I}_1;_{a_{c1}} \mathcal{I}_2 \text{ after } \sigma_1 \cdot a_{c1}) = \text{Out}(\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2 \text{ after } \sigma_1 \cdot a_{c1})$ and $\forall\ \sigma_1 \cdot a_{c1} \in \text{Traces}(\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2)$: $\text{Out}(\mathcal{I}_1;_{a_{c1}} \mathcal{I}_2 \text{ after } \sigma_1 \cdot a_{c1}) \subseteq \text{Out}(\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2 \text{ after } \sigma_1 \cdot a_{c1})$. By Definition 7, $\mathcal{I}_1;_{a_{c1}} \mathcal{I}_2$ tioco $\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2$.

v) $\sigma = \rho \cdot a$ with $a \in \Sigma_2 \backslash \{\overline{a_{c1}}\}$

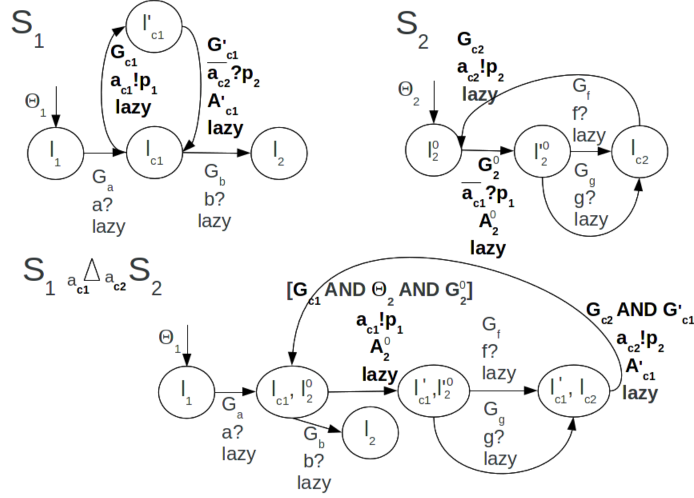


Figure A.2: Example of weak interruption composition

We use (3.2) from Definition 9, resulting in $\rho = \sigma_1 \cdot a_{c1} \cdot \sigma_2$ and $\sigma_1 \cdot a_{c1} \cdot \sigma_2 \cdot a \in$ Traces($\mathcal{S}_1 \;;_{a_{c1}} \mathcal{S}_2$). Because we assume that $\forall\ \sigma_1 \in$ Traces($\mathcal{S}_1$): Out($\mathcal{I}_1$ after σ_1) $\subseteq$ Out($\mathcal{S}_1$ after σ_1), $\forall\ \sigma_2 \in$ Traces($\mathcal{S}_2$): Out($\mathcal{I}_2$ after σ_2) $\subseteq$ Out($\mathcal{S}_2$ after σ_2) and Definition 8 constrains clocks from C_2 to restart from the transition that contains the a_{c1} action, we have $\forall\ \sigma_1 \cdot a_{c1} \cdot \sigma_2 \cdot a \ \in$ Traces($\mathcal{S}_1 \;;_{a_{c1}} \mathcal{S}_2$): Out($\mathcal{I}_1;_{a_{c1}} \mathcal{I}_2$ after $\sigma_1 \cdot a_{c1} \cdot \sigma_2 \cdot a$) $\subseteq$ Out($\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2$ after $\sigma_1 \cdot a_{c1} \cdot \sigma_2 \cdot a$) and, by Definition 7, $\mathcal{I}_1;_{a_{c1}} \mathcal{I}_2$ tioco $\mathcal{S}_1;_{a_{c1}} \mathcal{S}_2$.

□

A.2 Theorem 3

Similar to the sequential operator, proof of Theorem 3 is divided into three steps: i) Assume that subsystem implementations are tioco conformant to their subsystem specifications, ii) Apply tioco definition to Assumption i, iii) Use previous steps and the normal form of the interruption operator (Figure A.2) to prove that traces of the composed specification contain traces of the composed implementation.

Theorem 3 (tioco weak interruption Composition)**.** *Let $\mathcal{I}_1$, $\mathcal{I}_2$, $\mathcal{S}_1$ and $\mathcal{S}_2$ be four subsystems. If $\mathcal{I}_1$ tioco $\mathcal{S}_1$ and $\mathcal{I}_2$ tioco $\mathcal{S}_2$ then $\mathcal{I}_1\,{}_{a_{c1}}\!\triangle_{a_{c2}}\mathcal{I}_2$ tioco $\mathcal{S}_1\,{}_{a_{c1}}\!\triangle_{a_{c2}}\mathcal{S}_2$.*

Proof. According to Definition 7, we need to prove that:

$\forall\ \sigma_1 \in \text{Traces}(\mathcal{S}_1)$: $\text{Out}(\mathcal{I}_1 \text{ after } \sigma_1) \subseteq \text{Out}(\mathcal{S}_1 \text{ after } \sigma_1) \wedge$

$\forall\ \sigma_2 \in \text{Traces}(\mathcal{S}_2)$: $\text{Out}(\mathcal{I}_2 \text{ after } \sigma_2) \subseteq \text{Out}(\mathcal{S}_2 \text{ after } \sigma_2) \Rightarrow$

$\forall\ \sigma \in \text{Traces}(\mathcal{S}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{S}_2)$: $\text{Out}(\mathcal{I}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{I}_2 \text{ after } \sigma) \subseteq \text{Out}(\mathcal{S}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{S}_2 \text{ after } \sigma)$

Let $\sigma, \rho \in \text{Traces}(\mathcal{S}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{S}_2)$. By Definition 6, $\mathcal{S}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{S}_2 \xrightarrow{\sigma}$. From this point, σ is fourfold:

i) $\sigma = \epsilon$

We replace σ by ϵ in Definition 7, resulting in $\text{Out}(\mathcal{I}_1 \text{ after } \sigma_1) \subseteq \text{Out}(\mathcal{S}_1 \text{ after } \sigma_1) \wedge \text{Out}(\mathcal{I}_2 \text{ after } \sigma_2) \subseteq \text{Out}(\mathcal{S}_2 \text{ after } \sigma_2) \Rightarrow \text{Out}(\mathcal{I}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{I}_2 \text{ after } \epsilon) \subseteq \text{Out}(\mathcal{S}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{S}_2 \text{ after } \epsilon)$. Since ϵ belongs to any set of traces, this trivially holds.

ii) $\sigma = \rho \cdot a$ with $a \in (\Sigma_1 \backslash \{a_{c1}, a'_{c1}\} \cup \Sigma_2 \backslash \{a_{c2}, a_2^0\})$

We use (3.10) or (3.13) from Definition 12, resulting in $\rho \cdot a \in \text{Traces}(\mathcal{S}_{1\ \mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{S}_2)$. Because we assume that $\forall\ \sigma_1 \in \text{Traces}(\mathcal{S}_1)$: $\text{Out}(\mathcal{I}_1 \text{ after } \sigma_1) \subseteq \text{Out}(\mathcal{S}_1 \text{ after } \sigma_1) \wedge \forall\ \sigma_2 \in \text{Traces}(\mathcal{S}_2)$: $\text{Out}(\mathcal{I}_2 \text{ after } \sigma_2) \subseteq \text{Out}(\mathcal{S}_2 \text{ after } \sigma_2)$ and we require that $\mathcal{S}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{S}_2$ and $\mathcal{I}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{I}_2$ fill WIC normal form from Definition 10, we have $\forall\ \rho \cdot a \in \text{Traces}(\mathcal{S}_{1\ \mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\ \mathcal{S}_2)$: $\text{Out}(\mathcal{I}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{I}_2 \text{ after } \rho \cdot a) \subseteq \text{Out}(\mathcal{S}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{S}_2 \text{ after } \rho \cdot a)$ and, by Definition 7, $\mathcal{I}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{I}_2$ tioco $\mathcal{S}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{S}_2$.

iii) $\sigma = \rho \cdot a$ with $a = a_{c1}$

We use (3.11) from Definition 12, resulting in $\rho \cdot a_{c1} \in \text{Traces}(\mathcal{S}_{1\ \mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\ \mathcal{S}_2)$. Since we assume $\forall\ \sigma_2 \in \text{Traces}(\mathcal{S}_2)$: $\text{Out}(\mathcal{I}_2 \text{ after } \sigma_2) \subseteq \text{Out}(\mathcal{S}_2 \text{ after } \sigma_2)$ and $\mathcal{S}_1$ and $\mathcal{S}_2$ follow WIC normal form from Definition 10, where clocks from C_2 are constrained to restart from the transition that contains the a_{c1} action, $\forall\ \rho \cdot a_{c1} \in \text{Traces}(\mathcal{S}_{1\ \mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{S}_2)$: $(\mathcal{S}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{S}_2 \text{ after } \rho \cdot a_{c1}) = \{\langle l, \nu, \psi\rangle | l = (l'_{c1}, l_2'^0)\}$. Because we use (3.11) from Definition 12, $\forall\ \rho \cdot a_{c1} \in \text{Traces}(\mathcal{S}_{1\ \mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\ \mathcal{S}_2)$: $\text{Out}(\mathcal{S}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{S}_2 \text{ after } \rho \cdot a_{c1}) = \text{Out}(P_{\mathcal{S}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{S}_2})$, with $P_{\mathcal{S}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{S}_2} = \{\langle l, \nu, \psi\rangle | l = l_2'^0 \wedge l \in L_{\mathcal{S}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{S}_2}\}$. In the sequence, assuming that $\text{Out}(\mathcal{I}_2 \text{ after } \sigma_2) \subseteq \text{Out}(\mathcal{S}_2 \text{ after } \sigma_2)$ and $\mathcal{I}_1$ and $\mathcal{I}_2$ follow Definition 10, where clocks from C_2 are constrained to restart from the transition that contains the a_{c1} action, we have $\text{Out}(\mathcal{I}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{I}_2 \text{ after } \rho \cdot a_{c1}) = \text{Out}(P_{\mathcal{I}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{I}_2})$, with $P_{\mathcal{I}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{I}_2} = \{\langle l, \nu, \psi\rangle | l = l_2'^0 \wedge l \in L_{\mathcal{I}_{1\,\mathrm{a_{c1}}}\triangle_{\mathrm{a_{c2}}}\mathcal{I}_2}\}$.

Thus, $\forall\ \rho{\cdot}a_{c1} \in \text{Traces}(\mathcal{S}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{S}_2)$: $\text{Out}(\mathcal{I}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{I}_2 \text{ after } \rho{\cdot}a_{c1}) \subseteq \text{Out}(\mathcal{S}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{S}_2$ after $\rho \cdot a_{c1})$. Consequently, $\forall\ \rho \cdot a \in \text{Traces}(\mathcal{S}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{S}_2)$: $\text{Out}(\mathcal{I}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{I}_2$ after $\rho{\cdot}a) \subseteq \text{Out}(\mathcal{S}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{S}_2$ after $\rho{\cdot}a)$. By Definition 7, $\mathcal{I}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{I}_2$ tioco $\mathcal{S}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{S}_2$.

iv) $\sigma = \rho \cdot a$ with $a = a_{c2}$

We use (3.12) from Definition 12, resulting in $\rho \cdot a_{c2} \in \text{Traces}(\mathcal{S}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{S}_2)$. Since we assume $\forall\ \sigma_1 \in \text{Traces}(\mathcal{S}_1)$: $\text{Out}(\mathcal{I}_1$ after $\sigma_1) \subseteq \text{Out}(\mathcal{S}_1$ after $\sigma_1)$ and $\mathcal{S}_1$ and $\mathcal{S}_2$ follow WIC normal form from Definition 10, $\forall\ \rho \cdot a_{c2} \in \text{Traces}(\mathcal{S}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{S}_2)$: $(\mathcal{S}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{S}_2$ after $\rho \cdot a_{c2}) = \{\langle l, \nu, \psi\rangle | l = (l_{c1}, l_2^0)\}$. Because we use (3.10) and (3.11) from Definition 12, $\forall\ \rho \cdot a_{c2} \in \text{Traces}(\mathcal{S}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{S}_2)$: $\text{Out}(\mathcal{S}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{S}_2$ after $\rho \cdot a_{c2}) = \text{Out}(P_{\mathcal{S}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{S}_2})$, with $P_{\mathcal{S}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{S}_2} = \{\langle l, \nu, \psi\rangle | l = l_{c1} \wedge l \in L_{\mathcal{S}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{S}_2}\}$.

In the sequence, assuming that $\text{Out}(\mathcal{I}_1$ after $\sigma_1) \subseteq \text{Out}(\mathcal{S}_1$ after $\sigma_1)$ and $\mathcal{I}_1$ and $\mathcal{I}_2$ follow Definition 10, we have $\text{Out}(\mathcal{I}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{I}_2$ after $\rho \cdot a_{c2}) = \text{Out}(P_{\mathcal{I}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{I}_2})$, with $P_{\mathcal{I}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{I}_2} = \{\langle l, \nu, \psi\rangle | l = l_{c1} \wedge l \in L_{\mathcal{I}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{I}_2}\}$.

Thus, $\forall\ \rho{\cdot}a_{c2} \in \text{Traces}(\mathcal{S}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{S}_2)$: $\text{Out}(\mathcal{I}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{I}_2 \text{ after } \rho{\cdot}a_{c2}) \subseteq \text{Out}(\mathcal{S}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{S}_2$ after $\rho \cdot a_{c2})$. Consequently, $\forall\ \rho \cdot a \in \text{Traces}(\mathcal{S}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{S}_2)$: $\text{Out}(\mathcal{I}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{I}_2$ after $\rho{\cdot}a) \subseteq \text{Out}(\mathcal{S}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{S}_2$ after $\rho{\cdot}a)$. By Definition 7, $\mathcal{I}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{I}_2$ tioco $\mathcal{S}_1{}_{\text{ac1}}\triangle_{\text{ac2}}\mathcal{S}_2$.

□

A.3 Theorem 4

We use two lemmas to base the proof of Theorem 4. Lemma 1 proves that, if two subsystems are input-complete, the system which is composed by the parallel operator is also input-complete.

Lemma 1. *If $\mathcal{S}_1$ and $\mathcal{S}_2$ are two input-complete TIOSTS, $\mathcal{S}_1 \parallel \mathcal{S}_2$ is also input-complete.*

Proof. A location l of $\mathcal{S}_1 \parallel \mathcal{S}_2$ is a pair (l_1, l_2) where $l_1 \in L_1$ and $l_2 \in L_2$. By assumption, $\mathcal{S}_1$ and $\mathcal{S}_2$ are input-complete with relation to $\Sigma_1^?$ and $\Sigma_2^?$ in this sequence. Thus, $\forall\ a_i \in \Sigma_i^?$, $l_i \xrightarrow{a}$ with $i = 1, 2$. From this point, we identify two cases:

i) $(a \notin \Sigma_2) \vee (a \notin \Sigma_1)$

For each $((l_1, l_2), a, G, A, d, (l'_1, l'_2)) \in \mathcal{T}$, $a = a_1$ or $a = a_2$. Hence, $a \in \mathcal{S}_1 \parallel \mathcal{S}_2$ and $l \xrightarrow{a}$ holds.

ii) $(a \in \Sigma^?_1 \cap \Sigma^!_2) \vee (a \in \Sigma^!_1 \cap \Sigma^?_2)$

For each $((l_1, l_2), a, G, A, d, (l'_1, l'_2)) \in \mathcal{T}$, $a \notin \Sigma^?_1 \cap \Sigma^?_2$. Hence, this trivially holds.

□

Lemma 2 shows that traces of an input-complete specification contains the traces of an in an input-complete implementation if and only if the implementation is tioco conformant to the specification.

Lemma 2. *Let $\mathcal{I}$ and $\mathcal{S}$ be two input-complete TIOSTS such that $\Sigma_{\mathcal{I}} = \Sigma_{\mathcal{S}}$. Thus: $\mathcal{I}$ tioco $\mathcal{S} \Leftrightarrow$ Traces($\mathcal{I}$) $\subseteq$ Traces($\mathcal{S}$).*

Proof. We follow a similar line of reasoning as the one presented by Bijl *et al* [BRT04] in the scope of IOTS models. The proof is twofold:

i) $\mathcal{I}$ tioco $\mathcal{S} \Rightarrow \text{Traces}(\mathcal{I}) \subseteq \text{Traces}(\mathcal{S})$

Let $x \in Out(\mathcal{I}\ after\ \sigma)$, then $\mathcal{I} \xrightarrow{\sigma \cdot x}$, which implies $\sigma \cdot x \in \text{Traces}(\mathcal{I})$. By Definition 7, $\forall\ \sigma \in \text{Traces}(\mathcal{S})$: $Out(\mathcal{I}\ after\ \sigma) \subseteq Out(\mathcal{S}\ after\ \sigma)$. Hence, $x \in Out(\mathcal{S}\ after\ \sigma)$ and $\mathcal{S} \xrightarrow{\sigma \cdot x}$, from which it follows that $\sigma \cdot x \in \text{Traces}(\mathcal{S})$

ii) $\text{Traces}(\mathcal{I}) \subseteq \text{Traces}(\mathcal{S}) \Rightarrow \mathcal{I}$ tioco $\mathcal{S}$

Let $\sigma \in \text{Traces}(\mathcal{I})$. By induction on the structure of σ, we have:

- Basic step: $\sigma = \epsilon$

$\epsilon \in \text{Traces}(\mathcal{S})$ trivially holds. Hence $\epsilon \in \text{Traces}(\mathcal{S})$: $Out(\mathcal{I}\ after\ \sigma) \subseteq Out(\mathcal{S}\ after\ \sigma)$, which implies $\mathcal{I}$ tioco $\mathcal{S}$ by Definition 7.

- Induction step: We identify two cases:

(a) $\sigma = \rho \cdot a$ with $a \in \Sigma^?_{\mathcal{I}}$

Since $\sigma \in \text{Traces}(\mathcal{I})$, $\sigma \in \text{Traces}(\mathcal{S})$. So, $\exists\ \mathcal{S}' : \mathcal{S} \xrightarrow{\rho} \mathcal{S}'$ and because of the assumption $\Sigma_{\mathcal{I}} = \Sigma_{\mathcal{S}}$, $\mathcal{S}' \xrightarrow{a}$ always holds. Hence, $\rho \cdot a \in \text{Traces}(\mathcal{S})$: $Out(\mathcal{I}\ after\ \sigma) \subseteq Out(\mathcal{S}\ after\ \sigma)$, which implies by Definition 7 that $\mathcal{I}$ tioco $\mathcal{S}$.

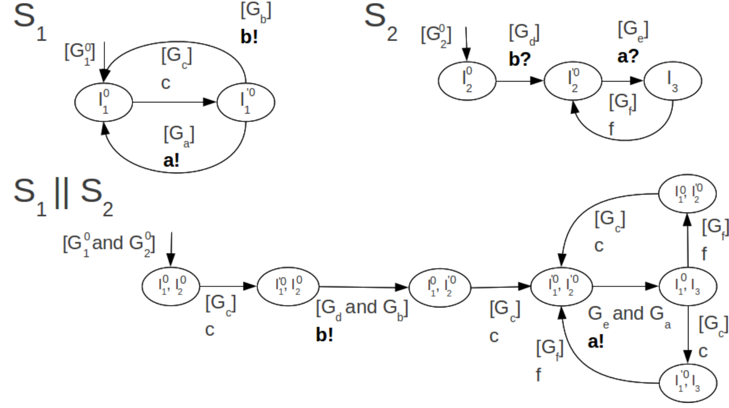


Figure A.3: Toy example for the parallel composition

(b) $\sigma = \rho \cdot x$ with $x \in \Sigma^!_{\mathcal{I}}$

If $\mathcal{I} \xrightarrow{\rho \cdot x}$, this implies that $\rho \cdot x \in \text{Traces}(\mathcal{I})$ and $x \in Out(\mathcal{I}\ after\ \rho \cdot x)$. Since $\text{Traces}(\mathcal{I}) \subseteq \text{Traces}(\mathcal{S})$, we have $\mathcal{S} \xrightarrow{\rho \cdot x}$ and $x \in Out(\mathcal{S}\ after\ \rho \cdot x)$. Consequently, $\forall \rho \cdot x \in \text{Traces}(\mathcal{S})$: $Out(\ \mathcal{I}\ after\ \rho \cdot x) \ \subseteq\ Out(\mathcal{S}\ after\ \rho \cdot x)$, which implies $\mathcal{I}$ tioco $\mathcal{S}$ by Definition 7.

□

Proof of the compositionality property for the parallel operator is composed by the steps: i) Assume that subsystem specifications are tioco conformant to their subsystem implementations, ii) Assume that subsystems are input-complete, iii) From Lemma 1, infer that the composed implementation and the composed specification are input-complete, iv) Use previous steps, Lemma 2 and the parallel operator definition to prove that traces of the composed specification contain traces of the composed implementation. Since the parallel operator has three different rules to build the set of transitions from the composed system (Figure A.3), the cases from the proof correspond to them.

Theorem 4 (tioco Parallel Composition)**.** *Let specifications $\mathcal{S}_1$, $\mathcal{S}_2$ and implementations $\mathcal{I}_1$, $\mathcal{I}_2$ be input-complete TIOSTS models. Also $\Sigma_{S_1} = \Sigma_{I_1}$ and $\Sigma_{S_2} = \Sigma_{I_2}$. If $\mathcal{I}_1$* tioco *$\mathcal{S}_1 \wedge \mathcal{I}_2$* tioco *$\mathcal{S}_2$ then $\mathcal{I}_1 \parallel \mathcal{I}_2$* tioco *$\mathcal{S}_1 \parallel \mathcal{S}_2$.*

Proof. The proof of Theorem 4 follows a similar line of reasoning as the one presented by Krichen and Tripakis [KT06] for the parallel synchronization operator in the scope of TAIO models. Accordingly, considering Lemma 2, we need to prove that:

$\text{Traces}(\mathcal{I}_1) \subseteq \text{Traces}(\mathcal{S}_1) \wedge \text{Traces}(\mathcal{I}_2) \subseteq \text{Traces}(\mathcal{S}_2) \Rightarrow$

$\text{Traces}(\mathcal{I}_1 \parallel \mathcal{I}_2) \subseteq \text{Traces}(\mathcal{S}_1 \parallel \mathcal{S}_2)$

Let $\sigma, \rho \in \text{Traces}(\mathcal{I}_1 \parallel \mathcal{I}_2)$. From Definition 13, σ is fourfold:

i) $\sigma = \epsilon$

$\epsilon \in \text{Traces}(\mathcal{I}_1 \parallel \mathcal{I}_2)$ and hence $\mathcal{I}_1 \parallel \mathcal{I}_2 \xrightarrow{\epsilon} \mathcal{I}_1' \parallel \mathcal{I}_2'$ by (3.16). Since $\epsilon \in \text{Traces}(\mathcal{S}_1 \parallel \mathcal{S}_2)$ by the definition of a general set, this trivially holds.

ii) $\sigma = \rho \cdot a$ with $a \notin \Sigma_2$. Because $a \notin \Sigma_2$, we use (3.14), which results $\rho \cdot a \in \text{Traces}(\mathcal{I}_1 \parallel \mathcal{I}_2)$. Since we assume that $\text{Traces}(\mathcal{I}_1) \subseteq \text{Traces}(\mathcal{S}_1)$ and $\Sigma_{\mathcal{I}_1} = \Sigma_{\mathcal{S}_1}$, $\mathcal{S}_1 \parallel \mathcal{S}_2 \xrightarrow{\rho \cdot a}$, which implies that $\rho \cdot a \in \text{Traces}(\mathcal{S}_1 \parallel \mathcal{S}_2)$. Consequently, $\text{Traces}(\mathcal{I}_1 \parallel \mathcal{I}_2) \subseteq \text{Traces}(\mathcal{S}_1 \parallel \mathcal{S}_2)$.

iii) $\sigma = \rho \cdot a$ with $a \notin \Sigma_1$. We use (3.15) from Definition 13. The next steps are analogous to ii).

iv) $\sigma = \rho \cdot a$ with $(a \in \Sigma_1^? \cap \Sigma_2^!) \vee (a \in \Sigma_1^! \cap \Sigma_2^?)$. Because $(a \in \Sigma_1^? \cup \Sigma_2^!) \vee (a \in \Sigma_1^! \cup \Sigma_2^?)$, we use (3.16), which results $\rho \cdot a \in \text{Traces}(\mathcal{I}_1 \parallel \mathcal{I}_2)$. Since $\text{Traces}(\mathcal{I}_1) \subseteq \text{Traces}(\mathcal{S}_1) \wedge \text{Traces}(\mathcal{I}_2) \subseteq \text{Traces}(\mathcal{S}_2)$, a is synchronizable in $\mathcal{I}_1 \parallel \mathcal{I}_2$ and $a \in \Sigma_1^! \cup \Sigma_2^!$. In addition, $\mathcal{I}_1$ and $\mathcal{I}_2$ are input-complete, so $\mathcal{I}_1 \parallel \mathcal{I}_2$ is input-complete by Lemma 1. Similarly, $\mathcal{S}_1 \parallel \mathcal{S}_2$ is input-complete. By the input-completeness and synchronization of $\mathcal{S}_1 \parallel \mathcal{S}_2$, $\mathcal{S}_1 \parallel \mathcal{S}_2 \xrightarrow{\rho \cdot a}$ holds and $\text{Traces}(\mathcal{I}_1 \parallel \mathcal{I}_2) \subseteq \text{Traces}(\mathcal{S}_1 \parallel \mathcal{S}_2)$.

□