MASTER

Supervisory control for flexible manufacturing systems

Priyanka

Award date:
2020

Technische Universiteit Eindhoven
University of Technology
Department of Electrical Engineering
Electronics Systems Research Group

Supervisory control for flexible manufacturing systems

Master Thesis

Priyanka
(1327321)

Company Supervisor
dr. Julien Schmaltz
julien.schmaltz@ict.nl

University Supervisors
Prof.dr.ir. J.P.M. (Jeroen) Voeten
J.P.M.Voeten@tue.nl

Alireza Mohammadkhani (MSc)
a.mohammadkhani@tue.nl

Eindhoven, October 2020

Abstract

With the advent of Industry 4.0, the complexity of cyber-physical systems (CPS) is ever increasing. These systems are developed according to their corresponding scientific disciplines, which are predominantly independent of each other. This separation of disciplines can no longer be sustained and needs to be bridged. Thus, a tight coordination between different disciplines is needed and an integrated approach must be taken for the design of these heterogeneous systems. The goal of this research is to bridge the gap between functional level specification and software level implementation for the design of a CPS, in particular, for flexible manufacturing systems (FMS). For an FMS, the functional level specifications are described using the activity framework. At the software level, the Dezyne toolset is used, which is a model-driven engineering toolset, to define its component behavior. A methodology is developed to automatically generate the Dezyne code from the activity models for the determinate scenarios. The generated Dezyne code is verified and validated to check correctness of the transformation made. Further, the solution is extended to handle exceptions in the activity framework. The Dezyne code is automatically generated for the various levels of criticality defined for exceptions and verified. The ”Factory Four” model is used as a case study which represents a small scale FMS. It can simulate the ordering process, production process and the delivery process.

Acknowledgement

I wish to thank a number of persons whose assistance were a milestone in the completion of this project.

First and foremost, I would like to express my deepest gratitude towards Prof.dr.ir. J.P.M. (Jeroen) Voeten for providing me with invaluable guidelines, great ideas and constructive supervision throughout the learning process of my master thesis. He convincingly guided and encouraged me to be professional and do the right thing even when the road got tough. Without his persistent help, the goal of this project would not have been realized.

I am indebted to dr. Julien Schmaltz for giving me the chance to be a part of ICT and providing me the space to apply new techniques. His useful remarks, infinite support and constant engagement proved monumental towards the success of this thesis.

I would like to recognize the invaluable assistance and constant supervision that Alireza Mohammadkhani provided during my thesis. He helped me a lot in streamlining the story line and guiding me through the difficult process of writing this thesis.

I would like to pay my special regards to Giray Ozel for his supportive attitude and continuous effort in developing the DSLs, and introducing me to the basic concepts needed for automating the code generation. I would also like to thank dr. Tanir Özçelebi for being in my defense committee.

Last but not the least, I owe more than thanks to my family and friends for constant encouragement throughout my life. Without their support, it is impossible for me to finish my education and graduate seamlessly.

Contents

Contents iv

List of Figures vi

List of Tables viii

1 Introduction 1
1.1 Activity Framework . . . 2
1.2 Tools Considered . . . 3
1.2.1 Logistics Specification and Analysis Tool . . . 3
1.2.2 Dezyne Toolset . . . 4
1.3 Problem context . . . 4
1.4 Problem statement . . . 5
1.5 Research questions . . . 5
1.6 Case study . . . 5
1.7 Research Approach . . . 6

2 Literature Study 8
2.1 Modeling an FMS . . . 8
2.2 Handling exceptions in an activity . . . 8
2.3 Code generation from activities . . . 9

3 Translation from Activity model to Dezyne code 10
3.1 Activity Framework . . . 10
3.2 Specification of an activity . . . 11
3.2.1 Logistics Specification and Analysis Tool (LSAT) . . . 11
3.2.2 Developing a Domain Specific Language (DSL) to define activity models: Activity DSL . . . 13
3.3 Semantics of an activity . . . 14
3.4 Dezyne Toolset . . . 16
3.4.1 Modeling in Dezyne . . . 16
3.5 Translating an activity model to Dezyne code . . . 17
3.5.1 Rules of translation . . . 18
3.5.2 Algorithm for translation . . . 19
3.6 Developing a Domain Specific Language (DSL) to define translation model: Transformation DSL . . . 21
3.7 Results . . . 21
3.8 Validation . . . 26
3.8.1 Correctness of translation made from activity model to the Dezyne code . . . 26
3.8.2 Scalability analysis . . . 27
3.9 Verification of the generated Dezyne code . . . 31

4 **Case Study: Translation from Activity model to Dezyne code** **35**
4.1 Modeling an FMS using the activity framework . . . 35
4.2 Defining the events . . . 37
4.3 Modeling the system in Activity DSL and Transformation DSL . . . 37
4.3.1 Activity model . . . 38
4.3.2 Translation model . . . 38
4.4 Generated Dezyne code . . . 39
4.5 Validating the translation method . . . 45
4.6 Verifying the generated code . . . 46

5 **Handling Exceptions in an activity** **48**
5.1 Exceptions in an activity . . . 48
5.2 Specification of exceptions in an activity . . . 51
5.3 Semantics of handling exceptions in an activity . . . 52
5.4 Modeling exceptions in Dezyne . . . 53
5.4.1 Behavior in Dezyne for Criticality level 1 . . . 53
5.4.2 Behavior in Dezyne for Criticality level 2 . . . 54
5.5 Rules for translation . . . 54
5.6 Algorithm for translation . . . 56
5.7 Adding Translation Model to the Transformation DSL . . . 60
5.8 Results . . . 61
5.9 Verification of the generated Dezyne code . . . 76

6 **Case Study: Handling Exceptions** **78**
6.1 Defining the events for handling exceptions . . . 78
6.2 Incorporating the exceptions in the Domain Specific Language . . . 79
6.2.1 Activity model . . . 79
6.2.2 Translation model . . . 81
6.3 Generated Dezyne code . . . 82
6.3.1 Criticality level 1 . . . 82
6.3.2 Criticality level 2 . . . 88
6.4 Verifying the generated code . . . 96

7 **Conclusions and future work** **98**
7.1 Conclusion . . . 98
7.2 Future Work . . . 98

Bibliography **100**

A **Dezyne Code for Resources** **102**
A.1 Resources of activities Act_1, Act_2 and Act_3 . . . 102
A.1.1 Critical level 0 . . . 102
A.1.2 Critical level 1 . . . 104
A.1.3 Critical level 2 . . . 107
A.2 Resources of Factory Four simulation model . . . 109
A.2.1 Critical level 0 . . . 109
A.2.2 Critical level 1 . . . 114
A.2.3 Critical level 2 . . . 120

List of Figures

1.1 Platform-based control layers of an FMS 1
1.2 Activity Framework 2
1.3 Workflow of LSAT 3
1.4 Dezyne software cycle 4
1.5 Factory Four model 6

3.1 Activities Act_1, Act_2 and Act_3 11
3.2 Activity DSL as an intermediary for code generation 13
3.3 Activities Act_1, Act_2 and Act_3 with timing information 15
3.4 Gantt chart for activities Act_1, Act_2 and Act_3 15
3.5 Dezyne abstraction levels 16
3.6 Activity model mapped to Dezyne elements 17
3.7 Activities Act_1, Act_2 and Act_3 with events 18
3.8 Dezyne code generation from Activity DSL and Transformation DSL 22
3.9 Sequence diagram for activity Act_1 26
3.10 Sequence diagram for activity Act_2 27
3.11 Sequence diagram for activity Act_3 27
3.12 Gantt chart for activity Act_1 obtained from the LSAT model 28
3.13 Gantt chart for activity Act_2 obtained from the LSAT model 28
3.14 Gantt chart for activity Act_3 obtained from the LSAT model 28
3.15 Gantt chart for activity Act_1, Act_2 and Act_3 obtained from Dezyne code 29
3.16 Activity diagram for activity Act_4 29
3.17 Verification results showing the number of states for activity Act_1 and Act_4 31
3.18 Verification result for activity Act_1 32
3.19 Verification result for activity Act_2 32
3.20 Verification result for activity Act_3 32
3.21 Verification result for activity *Act_combined* 34

4.1 Activity diagram for the *Warehouse Activity* 36
4.2 Activity diagram for the *Color Sorter Activity* 36
4.3 Events in Dezyne mapped to the *Warehouse Activity* 37
4.4 Events in Dezyne mapped to the *Color Sorter Activity* 37
4.5 Sequence diagram for the *Warehouse Activity* 44
4.6 Sequence diagram for the *Color Sorter Activity* 44
4.7 Gantt chart for the *Warehouse Activity* obtained from the LSAT model 45
4.8 Gantt chart for the *Color Sorter Activity* obtained from the LSAT model 45
4.9 Gantt chart for the *Warehouse Activity* obtained from Dezyne 46
4.10 Gantt chart for the *Color Sorter Activity* obtained from Dezyne 46
4.11 Verification result for the *Warehouse Activity* 46
4.12 Verification result for the *Color Sorter Activity* 47

5.1 Activities Act_1, Act_2 and Act_3 with exceptions 50

5.2 Gantt charts for activities Act_1, Act_2 and Act_3 for criticality level 1 52
5.3 Gantt chart for activities Act_1, Act_2 and Act_3 for criticality level 2 53
5.4 Activities Act_1, Act_2 and Act_3 with failure events 53
5.5 Sequence diagram for the activity Act_1 73
5.6 Sequence diagram for the activity Act_2 73
5.7 Sequence diagram for the activity Act_3 74
5.8 Sequence diagram for the activity Act_1 74
5.9 Sequence diagram for the activity Act_2 75
5.10 Sequence diagram for the activity Act_3 75
5.11 Verification result for the activity Act_1 76
5.12 Verification result for the activity Act_2 76
5.13 Verification result for the activity Act_3 76
5.14 Verification result for the activity Act_1 77
5.15 Verification result for the activity Act_2 77
5.16 Verification result for the activity Act_3 77

6.1 Events in Dezyne mapped to the *Warehouse Activity* 78
6.2 Events in Dezyne mapped to the *Color Sorter Activity* 79
6.3 Sequence diagram for the *Warehouse Activity* for criticality level 1 94
6.4 Sequence diagram for the *Color Sorter activity* for criticality level 1 94
6.5 Sequence diagram for the *Warehouse Activity* for criticality level 2 95
6.6 Sequence diagram for the *Color Sorter activity* for criticality level 2 95
6.7 Verification result for the *Warehouse Activity* 96
6.8 Verification result for the *Color Sorter Activity* 96
6.9 Verification result for the *Warehouse Activity* 96
6.10 Verification result for the *Color Sorter Activity* 97

List of Tables

5.1 Different levels of criticality for handling exceptions 50

Chapter 1

Introduction

There have been multiple shifts in the past with new technological advances in the industry, leading to various industrial revolutions. Industry 4.0, also known as the fourth industrial revolution, is a new vision for the future consisting of new technologies like advanced digitization, smart factories and Internet of Things, among others, aimed at achieving full automation of the manufacturing industry.

One of the enablers of Industry 4.0 are cyber-physical systems (CPS) where the digital and physical domains merge. These systems are characterised by a tight coupling between the cyber part, that is, computation, communication and control, with the physical processes, monitored and controlled by the cyber part. Usually, these two parts are designed separately, each using a specific set of engineering methods, tools, and technologies that are loosely coupled both on a syntactic and on a semantic level. These are also heterogeneous systems as they assimilate physical dynamics (continuous domain) with computational systems (discrete domain) [1]. These systems are ubiquitous and are commonly found in areas of automotive (autonomous automobile systems), aviation (automatic pilot avionics), infrastructure (smart grid), healthcare (medical monitoring) and manufacturing (flexible manufacturing systems).

Flexible manufacturing systems (FMS) are integrated, flexible and automated machines with a computer controlled complex of automated material handling system that can simultaneously process medium-sized volumes of a variety of part types [2]. These machines quickly adapt to changes in market demands with minimum cost by manufacturing a variety of products with high productivity, low cost and high accuracy. These machines have multiple production routes which can adapt to configuration changes resulting in the best product quality at the highest possible throughput. Therefore, there is a need for control software to control the product routing and execution order of products, while ensuring various system requirements. The design and control of an FMS can be viewed in terms of three layers: Plant, Process Control layer and Supervisory Control layer (Figure 1.1). *Plant* consists of the physical components of an FMS, for instance, a robotic arm, which is controlled by the Process Control layer. The *Process Control* layer usually consists

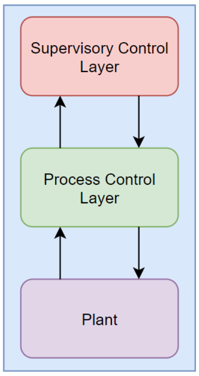


Figure 1.1: Platform-based control layers of an FMS

of embedded controllers which controls the operation of physical components, for instance, movement of the robotic arm. The design automation for this layer is rather mature and is handled by control engineers. The *Supervisory Control* layer is a layer on top of the Process Control layer. It controls the embedded controllers and ensures system requirements, in addition to preventing unsafe behavior of the system, such as collision of robotic arms. The development of supervisory controllers is a complex challenge due to ever increasing complexity of the FMS. Unlike the process control layer, the design automation for this layer is not mature and there is no universally accepted methodology for it.

1.1 Activity Framework

One of the design methodologies for supervisory control is the activity framework ([3], [4], [5]) which is shown in Figure 1.2. In the activity framework, the physical components of a manufacturing plant are decomposed into peripherals and resources, with resources being composed of peripherals. For instance, in an FMS, a robot is a resource, and its arm and gripper form its peripherals. The resources perform *actions* such as pick up a workpiece using robot. The dependencies between various actions is given by *activities*. For instance, picking up a workpiece at a workstation and moving to another workstation forms an activity. These activities are defined as a partial order over actions, and cover various functional requirements and constraints that need to be adhered to. The activities are combined to form *scenarios*. The activity

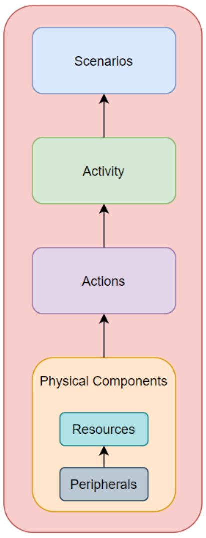


Figure 1.2: Activity Framework

framework enables modeling the requirements of an FMS at the functional level in terms of a scenario and obtaining the optimum sequence for that scenario, with respect to throughput or makespan by performing timing analysis on the specified scenario. Using this methodology, the functional requirements are modularly and concisely specified, which facilitates scalability. The toolset which provides a mechanism for activity modeling is Logistics Analysis and Specification Tool (LSAT) [6].

The activity framework does not form a complete design flow yet. No systematic approach exists to bridge the gap from the functional level requirements to the software code for an FMS. Thus, a gap exists between the functional level specification and the software level implementation. In addition, only determinate scenarios can be modeled at present. This methodology does not provide any mechanism to deal with exceptions, for instance, concerning a system's response in case of a collision of a robotic arm. Thus, there is a need to devise a method for handling exceptions in the activity framework at the functional level.

1.2 Tools Considered

Since a void exists between the functional level specification and the software level implementation of an FMS, it is imperative to work with tools which operate at these levels, in order to bridge the gap. The research will be carried within the scope of the two toolsets: LSAT which operates at the functional level, and Dezyne which operates at the software level. For the software level implementation, Dezyne toolset is considered as it is used in commonly used in the industry to model software components for systems.

1.2.1 Logistics Specification and Analysis Tool

LSAT enables modeling of an FMS using the activity framework. Its workflow is shown in Figure 1.3. An FMS is modeled by decomposing its parts into various peripherals and resources, and modeling the specifications through actions and activities. The interaction between physical components are defined in terms of actions and the dependencies between various actions is given by activities, which are partially ordered over actions. The controller is defined at the activity level and defines the order in which these activities must be deployed. In LSAT, the controller is specified using CIF 3.0 [7] in which activities are defined as an automata. Using these activities, a state space is generated on which logistics and constraints automata are imposed to reduce the size of the state space generated. The execution time of each action is captured in (max,+) matrices. The (max,+) automata is obtained as a product of (max,+) matrices and the automata representing activities. Once (max,+) automata is generated, it can be analysed to obtain optimal dispatching sequence (which is a sequence of activities).

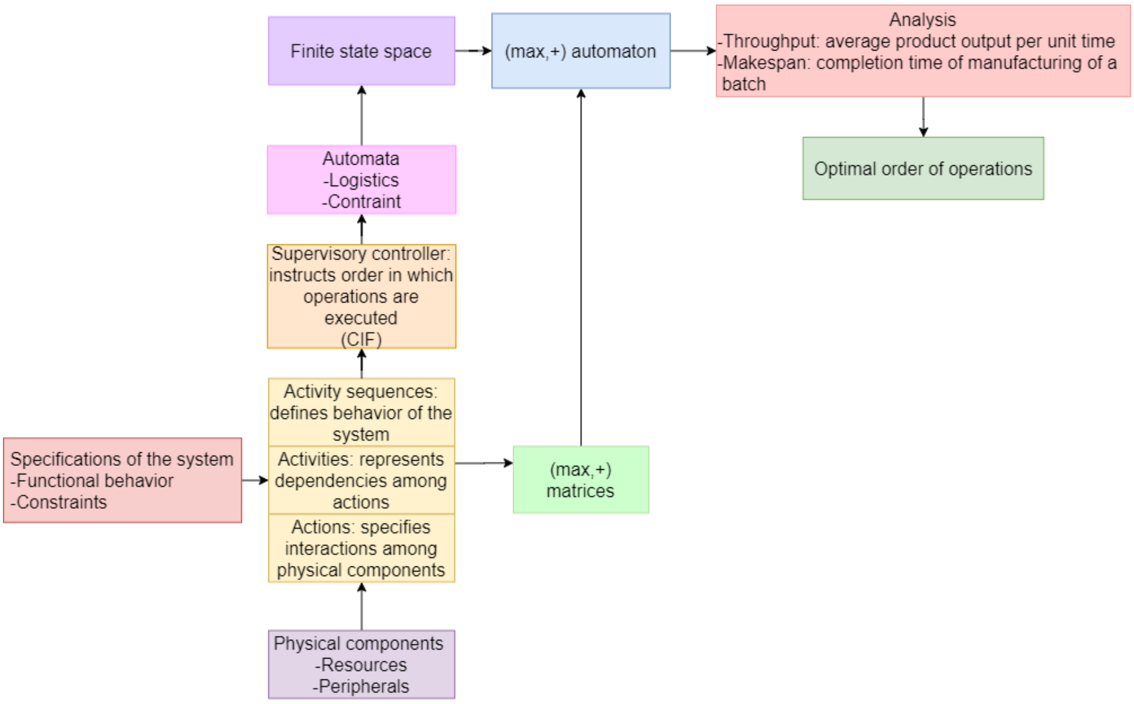


Figure 1.3: Workflow of LSAT

The analysis available in LSAT is in the form of throughput and makespan. Throughput is the steady-state product output per unit time, while makespan gives the completion time for manufacturing of a batch of products [8].

LSAT cannot be used to model non-deterministic activity sequences. It only models good-weather behavior. While verification of LSAT models using mCRL2 is under development, it does not have any mechanism for Dezyne code generation. But it does give an optimum dispatching sequence for throughput/makespan for a happy flow of activity sequences. This enables exploration of various design choices at earlier stages of the development process.

Although LSAT provides a mechanism to model an FMS using the activity framework methodology, it will not be used for this project since it has a lot of features for defining specifications which is not needed for the software code generation. Hence, a separate Domain Specific Language (DSL) will be developed to specify the requirements of an FMS.

1.2.2 Dezyne Toolset

Dezyne [9] is a model-driven engineering toolset that allows users to model interface behaviors and implementation behavior of the components and generate executable code from these (Figure 1.4). In addition, it provides a built-in formal verification engine, using mCRL2 language [10], to check the conformance of implementations to the modeled interfaces. The model can be scanned for errors and checks for unwanted properties like deadlocks, livelocks, incomplete mapping of events and responses, race conditions, illegal actions and compliance.

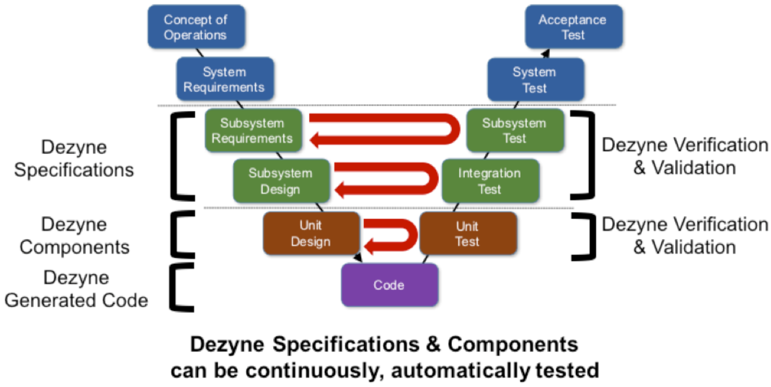


Figure 1.4: Dezyne software cycle

Dezyne support models that captures the behavior of the software system. The model serves as a means of communication between software designer(s) and other stakeholders. It ensures that the requirements formulated by the stakeholders are thorough, complete and effectively implemented. Dezyne also allows the software engineer to simulate software behavior at every step of the development process, which helps to verify whether the system meets the requirements. Once tested and verified, computer code is automatically generated from the model. [11]

In Dezyne, systems are modeled in terms of interfaces, components and behavior. Conceptually, an interface specification describes the sequence of allowed and expected events that can take place at a given interface. An *Interface* consists of functions which are used to trigger events. The implementation for an *interface* is provided by the *components*. *Components* define the *behavior* of systems in terms of states and the allowed events. Once the system is modeled, it can be verified and sequence diagram can be generated to check the behavior. This is complementary to the UML/SysML approach in which first sequence diagrams are defined followed by the behavior of the system.

1.3 Problem context

The design of the supervisory controller for an FMS is a complex problem, with complexity increasing with the size of these systems. In addition, the design automation for this layer is not mature yet, and there is no universal methodology to design and control an FMS. In this project, the activity framework is used as a methodology for the supervisory control of an FMS. This framework operates at functional level but is not complete. It enables one to specify the functional behavior of the system and to analyze various scenarios to obtain optimal throughput and/or makespan. For this purpose, an FMS is abstracted into peripherals, resources, actions, activity, activity sequences and scenarios. However, the scope of this project will be limited to abstraction levels of resources, actions and activities, since including peripherals and activity sequences have its own set of challenges. This framework also lacks a mechanism to generate executable Dezyne code from the models and no systematic approach yet exists to generate executable Dezyne code. Thus, a gap exists between the functional level specification and the software level implementation. In addition, only determinate behavior can be specified at present. There is no method to deal with exceptions that may interrupt the execution of activities at runtime.

At the software level, the Dezyne toolset provides a platform to specify, design, validate and formally verify the system. It can automatically generate code that meets safety and security requirements. In addition, a mechanism exists to handle exceptions that may occur in the system. However, specifications are at a lower abstraction level- the software level, in the form of actions, rather than defined as activity sequences.

Therefore, there is a need for systematic translation from functional level specifications to software level implementation. This can be achieved by translating activity models to Dezyne code. Once this is achieved, exception handling will be incorporated in models at the functional level.

1.4 Problem statement

There is no complete design flow yet for the design and control of an FMS, and a gap exists between the functional level specifications and the software level implementation for it. There is no systematic translation from the specifications described in terms of activities to the software code. Also, no mechanism for handling exceptions is present in the activities.

1.5 Research questions

The answers to the following research questions narrow the gap between functional specifications and software implementation for an FMS.

1. How to generate Dezyne code from activity models?
2. How to handle exceptions in the activity framework?

1.6 Case study

For reasons of cost and effort, it is not possible to get easy access to industrial machines for performing experiments. Assumptions made and approaches proposed cannot be validated and may be unrealistic for practical industrial platforms. Hence, a simulation model is needed to enable development of novel techniques that are practical in nature.

The Factory Four model [12] of FischerTechnik (Figure 1.5) is a highly flexible, modular, cost-effective and robust training model that can be used to carry out highly technical logistical processes. It is ideal for the demonstration of industrial automation. It is a small scale manufacturing system that depicts the ordering process, the production process and the delivery process.

The Factory Four model, made available for this project by the ICT Group, Netherlands, is a combination of models: a sorting line with color recognition, a multi-processing station with oven, an automated high-rack warehouse and a vacuum suction robot. It has a closed material cycle: widgets are outsourced from the high-bay warehouse, processed in the processing station, sorted by color in the sorting plant and then stored again in the high-bay warehouse. This is a never-ending, repetitive cycle. Still, the work flow can be divided into two main sequences:

1. Warehouse sequence: Moving the widget from the high-bay warehouse to the multi-processing station.
2. color sorter sequence: Moving the widget from the sorting line back to the high-bay warehouse.

The subsystems are described in detail below: [13]

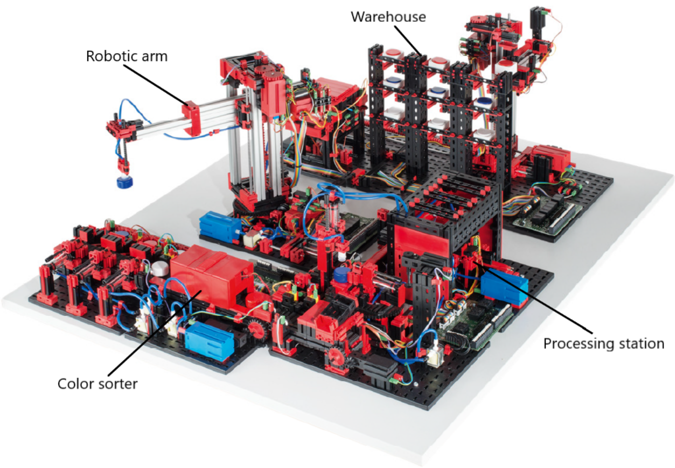


Figure 1.5: Factory Four model

1. Automated high-bay warehouse: It is a storage shelf that saves space and allows storage and retrieval of goods. It can hold 9 widgets at a time in a 3x3 arrangement. Each widget is stored in a box placed on the shelf. In addition, it has a separate robotic arm which moves these boxes from the storage space to the conveyor belt. This belt moves forward and pushes out the box, making the widget available to the vacuum gripper robot for picking. Once the widget is picked, the empty box is stored back into the storage space.

2. Multi-processing station with oven: The widgets automatically pass through several stations that simulate different processes in the multi-processing station with oven. Conveyor mechanisms such as a conveyor belt, a turntable and a vacuum suction gripper are used. These peripherals communicate with each other to prevent collisions. To initiate processing, a widget is placed on the oven pusher, the kiln door opens, retracts the kiln slider and the door is closed. Simultaneously, the vacuum suction gripper, which brings the widget to the turntable after the firing process, is requested. Following the firing process, the kiln gate is re-opened and the kiln slider is re-extended. The positioned vacuum suction gripper, already in place, picks the widget up, transports it to the turntable and places it there. The turntable positions the widget under the miller, waits there until the job is finished and then moves the widget to the ejector. The ejector pushes the widget onto the conveyor belt, which conveys the widget to the sorting system.

3. Sorting line with detection: The sorting path with color recognition automatically separates differently colored widgets. A conveyor belt feeds geometrically identical but differently colored components into a color sensor, where they are separated according to color using an optical color sensor. Once the color is recognised, the ejector pushes the widget to their respective bearing locations.

4. Vacuum gripper robot: Three-axis robot with vacuum suction gripper positions widgets quickly and precisely in the three-dimensional space. It picks up the widget and moves it within the working space. Positioning the suction gripper or transporting the widget can be defined as a point-to-point movement or as a continuous path. Controlling the individual axes can take place sequentially and / or in parallel and is significantly influenced by obstacles present in the work space.

1.7 Research Approach

The systematic translation from functional level specifications to the software level implementation for an FMS will be achieved through the following steps:

1. Define the functional requirements of an FMS: The functional requirements of an FMS is defined in terms of activities, actions and resources, and specifying the dependencies between actions.

2. Develop a Domain Specific Language (DSL) to define activity models: Since the LSAT provides specifications which are too descriptive for the purposes of this project, a new DSL is developed to define specifications of an FMS. The activity model uses concepts of the activity framework, and specifies the system in terms of resources, actions and activities.

3. Translating the activity model to Dezyne code: To obtain the translation, concepts of the activity framework is mapped to the Dezyne elements. This includes defining events for activities and actions, and handling dependencies between actions. This translation is then generalised and an algorithm is obtained.

4. Develop a Domain Specific Language (DSL) to define translation models: Another DSL is developed to define the translation model that specifies information needed for the translation of an activity model to the Dezyne code. This mainly contains defining events for the activity, and actions contained in that activity. The algorithm obtained for translation is added to this DSL for automated Dezyne code generation.

5. Handling exceptions: The solution is then extended for handling exceptions. The mechanism to handle exceptions is incorporated in the activity framework by defining specification and semantics for an exception. It is, then, added to the DSL defining the activity model. Next, the exception is mapped to the Dezyne elements by defining events and handling dependencies between actions in case an exception occurs. This is then generalised and an algorithm is obtained. This algorithm, along with the events, is then added to the DSL of the translation model for automated Dezyne code generation in case an exception occurs.

6. Validation: The translation is tested for accuracy using the Gantt charts obtained from modeling activities in LSAT, which is then compared with the Gantt charts obtained from sequence diagrams of the generated Dezyne code.

7. Verification of the generated code: The generated Dezyne code is verified for deadlock, livelock, completeness, compliance, illegal events, non-determinism, and type and range error in the Dezyne environment.

Chapter 2

Literature Study

2.1 Modeling an FMS

Development of FMS is a complex challenge since there is a gap that exists between functional level specifications and software level implementation. Systems are logical and physically divided, they need to run on different platforms, meet specific execution times and address communication issues. The complexity created by this gap between the problem domain and the deployment domain needs to be reduced through various modeling techniques.

An automata based modeling language Compositional Interchange Format (CIF) [14] provides a method for modeling a CPS and supports synthesis of a supervisory controller, along with simulation-based validation and verification of models. The system is modeled as an uncontrolled plant on which control requirements are superimposed to generate a supervisory controller. CIF language is used in LSAT to specify the possible activity sequences in terms of automata, although it does not offer explicit support to model activities.

UML and SysML also provide a mechanism to specify the behavior of a system in terms of activities [15] which can be used to model a wide range of applications. UML supports structural modeling and behavioral modeling. It defines the semantics of actions which serve as fundamental units of behavior specifications. The sequence of action executions is defined by control flows or object flows which additionally provide input to actions from outputs of other actions using tokens. Tokens are not explicitly modeled in an activity, but are used for describing the execution order of actions in an activity. A graphical representation of an activity is given by an activity diagram [16]. It consists of action nodes interlinked by control flows. The control nodes such as forks, joins, decision and merge nodes are used to manage the control flow in case of parallel execution and decision-based execution of action nodes.

A formal modeling approach using model based method to design FMSs is proposed in [3], [4] and [5]. These form the basis for the activity framework. While the former two focus on design of throughput-optimal supervisory controllers for FMS, the latter extends the work to makespan-optimal supervisory controllers for FMS. A scenario based modeling method is introduced in which functioning of the system is described using determinate activities. These activities represent run-time situations which are abstracted into scenarios. Activities capture various functional behaviors such as complete manufacturing of a product. Using scenario based modeling approach a supervisory controller is synthesized which guarantees functional correctness in addition to optimizing performance criteria. It restricts the model behavior to ensure that only proper behavior is allowed. It builds upon activity models, and uses (max,+) algebra for performance analysis and design-space exploration.

2.2 Handling exceptions in an activity

The activity framework does not have a mechanism to handle exceptions. UML provides two constructs for handling exceptions in an activity: [17], [18]

1. Interruptible region: In case of interruptible region construct, a raised exception is passed from the point at which it is raised along the activity, until it reaches an action protected by an exception handler which is contained in the interruptible region. All tokens are destroyed in the part of activity from which raised exception passes through to reach the exception handler, after which the exception handler is executed. This usually results in abandonment of the activity or a portion of it.

2. Exception parameters: The other exception handling facility concerns the exception parameters. These provide output values to the outing control flow for actions. They destroy all the tokens in the activity or action they flow out of. If an exception output is provided, the other outputs and outgoing control are not, and the action must immediately terminate.

2.3 Code generation from activities

UML is not a fully formal language. It does not use a fully formalized semantics. In many places natural language is used for model specification. This leads to a scenario where the precise model presentation is difficult and leads to ambiguity during automatic implementation of UML models.

In order to use a UML diagram for code generation, it must be complemented with specification languages. One such language is the Object Constraint Language (OCL) [19] which is used in [20] for generating code from the activity diagrams. The activity diagram is considered a graph which supports the traversal through the activity diagram and generate the overall execution logic of the system. OCL expressions are included in the activity diagram to formally specify the constraints in a precise and concise way. The system design is prepared in activity diagram and additional details are added using OCL expressions. This system model is then converted to XML format, which after checking the OCL expressions, is passed to the code generation module.

A model-driven engineering approach focussed on the generation of C++ code from UML models is given in [21]. The behavior specifications of a system is modeled in terms of UML state-machine diagrams using the Action Language for Foundational UML (ALF) [22] and follows component-based pattern for code generation. The code generation process is composed of model-to-model transformation that transforms the input into intermediate representations. These intermediate representations consists of an instance model and an intermediate model. An instance model represents components and ports instances for enabling correct generation of the communication links between components at code level. The intermediate model contains all the required information, both structural and behavioral, derived from the design model to generate full implementation code. Finally, the C++ implementation code is generated through model to text transformation that entails both static and behavioral descriptions of the system.

In [4], a supervisory controller developed using the activity framework is translated into a controller that directs the execution of activities on the physical platform in real-time. It can dynamically schedule activities and respond to external triggers such as different types of products that are fed to the system. The transformation is applicable to platforms that provide an event-driven programming architecture to interact with the hardware. In this architecture, there is a main event loop that continuously checks for the occurrence of the events. Event handlers are linked to this loop to perform actions when an event occurs. The callbacks to all events that occur in the event loop are registered. The controller is linked to the application programming interface (API) of the system that abstracts from the hardware details. The precise implementation of the actions and the activities is dependent on the platform.

Chapter 3

Translation from Activity model to Dezyne code

An FMS is modeled using the activity framework. The functional behavior is expressed as activities and activity sequences. An activity consists of partially ordered actions performed by peripherals on various resources. The specifications of an FMS are defined in LSAT in terms of peripherals, resources, actions and activities. Since the scope of this project is restricted to resources, actions and activities, LSAT specifications become too expressive in nature due to various specification features available for the peripherals. To keep specifications concise, a new Domain Specific Language (DSL) is developed, which is a subset of LSAT. Further, the semantics of executing an activity is explained and represented in terms of Gantt charts.

The software level implementation for these activities is expressed as Dezyne models. A model in Dezyne has components which interact with interfaces through events. The activity is mapped to Dezyne code by defining events for activity, and actions contained within that activity, while handling dependencies between actions. Once this mapping is achieved, the rules for transformation is defined and an algorithm is developed for automated translation of an activity model to the Dezyne code. The information needed for translation and the algorithm is added to another DSL and automatic Dezyne code is generated.

3.1 Activity Framework

The activity framework is a scenario-based formal modeling approach for the design of an FMS. In this framework, system functionality is described using determinate activities. These activities capture various functional behaviors such as the complete manufacturing of a product. Using scenario based modeling approach a supervisory controller is synthesized which guarantees functional correctness in addition to optimizing the performance criteria.

The physical components of an FMS is abstracted in terms of peripherals, resources and actions. The system consists of a set of **peripherals** which execute **actions**. The set of peripherals is aggregated into **resources**. These resources can be claimed before performing an action and released after an action completes execution. On top of these actions, an **activity** is constructed to define functional behavior of the system, for instance, a manufacturing operation. An activity is specified as a directed acyclic graph, which consists of a set of actions executed on different resources, and a set of dependencies between those actions. A **scenario** captures more elaborate functional behaviors by considering multiple activities, such as the complete manufacturing of a product in an FMS. It defines the order in which various activities are executed.

The structure of an activity as a directed acyclic graph is shown using three example cases in Figure 3.1. The nodes in the graph represents either an action executed by a peripheral (shown in green), or the claim or release of a resource (shown in orange).

Activity Act_1 has one resource *r1* executing two actions *a* and *b*. Action *a* is executed by peripheral *p1* and action *b* is executed by peripheral *p2*. At the start of the activity, resource *r1* is claimed and once the actions complete execution, the resource is released.

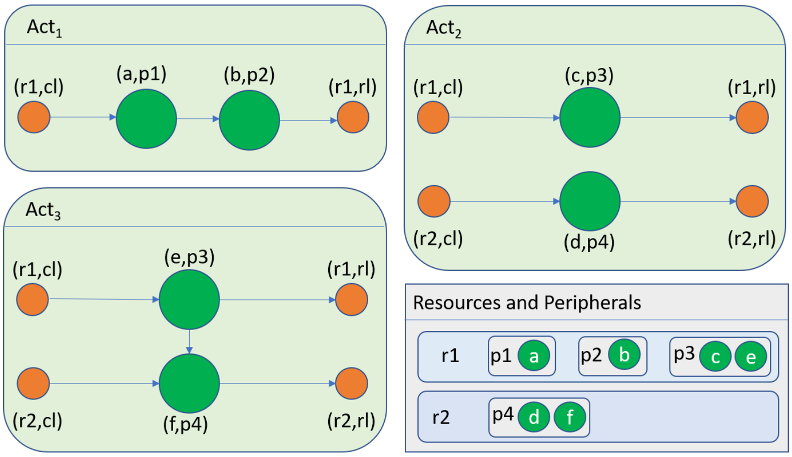


Figure 3.1: Activities Act_1, Act_2 and Act_3

Similarly, activity Act_2 has two resources *r1* and *r2* executing actions *c* and *d* respectively. Action *c* is executed by peripheral *p3* on resource *r1* and action *d* is executed by peripheral *p4* on resource *r2*. In this activity, no dependency is defined between actions *c* and *d*. Hence, these actions can execute concurrently.

The dependency is defined for actions *e* and *f* in activity Act_3. It has two actions *e* and *f* executed on resources *r1* and *r2* respectively. However, action *f* must execute after action *e*. This dependency is shown as an edge emanating from action *e* and ending at action *f*.

3.2 Specification of an activity

3.2.1 Logistics Specification and Analysis Tool (LSAT)

The model-based engineering tool which enables the design and analysis of an FMS using the activity framework is the Logistics Specification and Analysis Tool (LSAT). In LSAT, an FMS is defined in terms of resources, peripherals, actions, motion profiles, and activities which are a set of partially ordered actions executed by the peripherals. The following types of specifications can be created in LSAT for a system: [8]

1. **Machine specification:** In this specification, physical components of a system is defined in terms of resources, peripherals and actions. The specification starts with defining a *peripheral* which can be either movable or unmovable in nature. Next, for each peripheral, the *actions* it can execute is specified. For movable peripherals, for each action, set points and axes are defined. *Set points* specify the physical coordinate system of the peripheral on which the motion profile settings are applied, and *axes* relate to the symbolic coordinate system on which the physical locations are applied. An axis specifies which set points change when its value changes.

   After defining a list of peripherals with their corresponding actions, *resources* are specified for a system. For each resource, the peripherals it contains are defined. For every movable peripheral in a resource its *symbolic positions* are also specified. In case, a peripheral has multiple axes, a symbolic position may be expressed in terms of its *axes position.*

   Next, paths, and profiles for the defined paths are described. *Paths* refer to the moves that are allowed in the system. If a path is declared between two locations the move is allowed using the speed *profile* as specified by the path. A path can be a unidirectional path, a bidirectional path or a full mesh.

2. **Setting specification:** The setting specification establishes the profiles and positions for the defined peripherals. For a movable peripheral, it is required to set the motion *profiles* which can be either a third-order profiles or a second-order profiles. There is also an option to specify settling time per profile. For symbolic *positions*, a physical relative location is indicated which can be a set of maximum, minimal and default values, or a fixed value. Next, for each action executed by the

peripheral, *timing* must be specified. This timing can be either a fixed value or may vary in nature. The varying timing value is usually indicated using distributions such as a normal distribution, a triangular distribution or a PERT distribution.

3. **Activity specification:** In this specification, an *activity* is described by defining individual actions and specifying dependencies between these actions. As a *prerequisite*, the initial positions of all movable peripherals used in the activity is specified. Next, a list of *actions* contained in this activity is defined. An action can be either a resource *claiming* action, a resource *releasing* action or an action executed by a peripheral. A variable name is defined for each action. Subsequently, an *action flow* is defined to specify dependencies between actions. These dependencies are indicated using arrows $->$ and sync bars $|$. The arrows represent sequential execution of actions. For instance, $a1->a2$ specifies that action *a2* starts execution when action *a1* completes. A sync bar is used to specify synchronization points in an activity. For instance, $a1->|S1->a2$ and $|S1->a3$ specifies that actions *a2* and *a3* start execution concurrently after action *a1* completes execution.

4. **Activity dispatching specification:** This specification enables the scheduling of a sequence of activities. Initially, the number of *iterations* specify the default number of iterations for all resources or for a specific resource, in order to calculate the *throughput*. Next, an activity sequence is specified within *activities*, which contains one or more activities. These activities are scheduled as parallel as possible adhering to the claiming and releasing of resources. If activities cannot be scheduled in parallel the order specified is applied. Optionally, an *offset* can be indicated which ensures that the activities do not start before the offset time.

The analysis available in LSAT is in the form of throughput and makespan. Throughput is the steady-state product output per unit time, while makespan gives the completion time for manufacturing a batch of products. These parameters also enable improving performance by identifying where the specification can be optimized. However, performance optimizations and analysis is not required for Dezyne code generation, hence, these features become unnecessary.

In addition, for the purpose of this project, the specifications for activities provided by LSAT is rather rich and even superfluous in nature. Since the scope of this project is limited to the activity level, the specifications for activity sequences becomes expendable. This completely eliminates the need for activity dispatching specification in the DSL. In addition, as only one activity is dealt with at a time, the need for claiming and releasing of resources specified in activity specification is also eliminated. Furthermore, LSAT is user-friendly in nature, hence, it has syntactic sugar such as sync bars to specify the dependencies between various actions within an activity. However, these dependencies can be specified without the use of sync bars, making them redundant for the purposes of this project.

In this project, resources form the lowest aggregation level of the physical components of an FMS, rather than the peripherals. Since the concept of peripherals is excluded from the scope of this project, the need for setting specification in LSAT is completely eliminated. Furthermore, in machine specification, the various features related to peripherals which are available in LSAT, for instance, set points, symbolic and axes positions, motion profiles, etc. become unnecessary for the purposes of this project.

LSAT only models determinate scenarios. It has no mechanism to deal with handling exceptions, that is, specifying that an exception may occur in an activity and defining the response of the system when an action fails. Since one of the goals of this project is to include exceptions in the activity framework, the specification that an exception may occur must be included in an activity model. However, LSAT does not provide a mechanism to do so.

For the reasons listed above, a Domain Specific Language (DSL) is developed as an intermediate step towards Dezyne code generation (Figure 3.2). It contains concepts which form a subset of LSAT, in specific, resources, actions and activities, to describe the specifications of an FMS. It is further extended to include exceptions in an activity model.

Figure 3.2: Activity DSL as an intermediary for code generation

3.2.2 Developing a Domain Specific Language (DSL) to define activity models: Activity DSL

Domain Specific Languages are specific languages created to capture or document the requirements and behavior of a specific domain. It solves a limited set of problems and supports a definitive set of tasks related to that domain. It is created for a limited sphere of applicability and use for a specific context, while being powerful enough to represent and address the problems and solutions in that sphere. [23]

A textual DSL, called the **Activity DSL**, is developed to provide specification of an FMS using the activity framework in terms of resources, actions and activity. This DSL acts as an intermediary to generate Dezyne code from the specifications of an FMS.

Activity Model

To model an FMS, a list of *Resources* with their corresponding *actions* is defined. A resource is specified with the keyword **Resource** followed by a name of that resource. Next, the type of actions executed on that resource is defined. An action type is specified using the **Action type** keyword followed by a name for that action. Once all the resources and their corresponding action types are specified for a system, the *activities* can be defined. An activity is specified using the **Activity** keyword followed by its name. Next, instances of action types running on different resources contained within that activity are specified with a variable to denote each action. Each action type in the list refers to the resource it is executed on. This is specified using a format *variable: resource name.action type name*. The **Dependencies** specify the relationship between the various actions, that is, which actions must complete in order for the next actions to start. This is specified using a **->** syntax. In case multiple actions need to complete for an action to start, each dependency must be specified individually. In case there is an action for which no dependency is defined, it is executed in parallel since an activity is partially ordered over actions.

For an activity model, the following rules are defined in the Activity DSL:

1. Unique resource name: No model can have two or more resources with the same name.
2. Unique action names within a resource: Two or more actions within the same resource are not allowed to have identical names. However, identical action names executed on different resources are allowed.
3. Unique activity name: A user can specify multiple activities in an activity model but identical names for two or more activities are not allowed.
4. Unique variables for actions within an activity: While defining the instances of actions in an activity, no two or more actions are allowed to have same variables. In case there are two instances of the same action in an activity, they can be specified using two different variables.
5. Self-dependency: A dependency from an action to itself is not allowed.
6. Duplicate dependencies: The case where a dependency between two actions is specified more than once is not allowed.

The activity model for our example cases is shown below. There are two resources *r1* and *r2*. As can be seen from Figure 3.1, actions *a*, *b*, *c* and *e* are executed on resource *r1*. Actions *d* and *f* are executed on resource *r2*. Next, activity Act_1 is defined. It has two actions *a* and *b* running on the same resource *r1*. The list of actions within the activity are indicated as *A1: r1.a* and *A2: r1.b*. Next, dependencies between actions are specified. The only dependency in this activity is that action *b* must execute after action *a*. This is indicated in the DSL as *A1 -> A2*.

Similarly, activity Act_2 is defined. It has two actions *c* and *d* running on different resources, resource *r1* and *r2* respectively. The instances of action types within the activity is defined as *A3: r1.c* and *A4: r2.d*. Next, dependencies between actions are specified. Since, there is no dependency between actions *c* and *d*, they can execute concurrently.

Next, activity Act_3 is defined. It has two actions *e* and *f* running on different resources, resource *r1* and *r2* respectively. The instances of action types within the activity are defined as *A5: r1.e* and *A6: r2.f*. Next, dependencies between actions is specified. The only dependency in this activity is that action *f* must execute after action *e*. This is indicated in the DSL as *A5 -> A6*.

The **activity model** for activities Act_1, Act_2 and Act_3

```
Resource r1
   Action type a
   Action type b
   Action type c
   Action type e
Resource r2
   Action type d
   Action type f

Activity Act1
   Criticality level 0
   A1 : r1.a
   A2 : r1.b
   Dependencies
   A1 -> A2

Activity Act2
   Criticality level 0
   A3 : r1.c
   A4 : r2.d
Dependencies

Activity Act3
   Criticality level 0
   A5 : r1.e
   A6 : r2.f
   Dependencies
   A5 -> A6
```

3.3 Semantics of an activity

The execution of an activity has two main aspects: synchronisation and delay. Synchronisation means when an action waits for all incoming dependencies to complete, and delay means that the action takes a fixed amount of time to execute. The activity framework has timing information on action level, activity level, and activity sequence level. In LSAT, the timing for each action executed by a peripheral is specified in setting specification. This is extracted in (max,+) matrices. Using this, the timing information at activity level and activity sequence level is obtained. [5]

The execution time for each action in the examples is introduced in Figure 3.3. The actions running on resource *r1* take 1 time unit to execute whereas actions running on resource *r2* take 2 time units. The execution of the activities is shown as a Gantt chart in Figure 3.4. The time and resources are represented on the X and Y axis respectively. Thick edges indicate the time at which resources are claimed and released by the activity.

Activity Act_1 uses only resource *r1* which executes two actions *a* and *b* with execution time of 1 time unit. Resource *r1* is claimed at time stamp 0 and action *a* starts execution. It executes till time stamp 1. Action *b* starts execution only after action *a* completes execution. Action *a* completes at time stamp 1 and

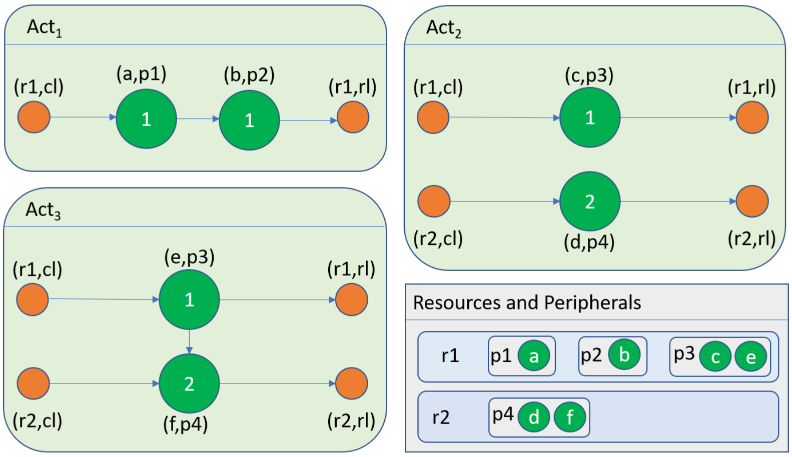
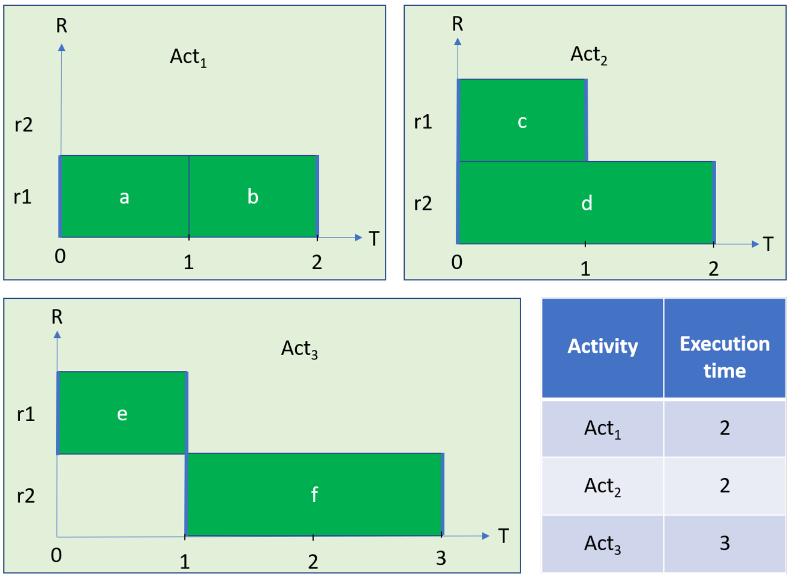


Figure 3.3: Activities Act_1, Act_2 and Act_3 with timing information

action b starts execution. Action b completes execution at time stamp 2, the resource *r1* is released, and activity Act_1 completes execution. Thus, the execution time of activity Act_1 is 2 time units.



Activity	Execution time
Act_1	2
Act_2	2
Act_3	3

Figure 3.4: Gantt chart for activities Act_1, Act_2 and Act_3

Activity Act_2 uses resource *r1* and *r2*, which execute two actions c and d with execution time of 1 and 2 time units respectively. Since there is no dependency defined between actions c and d, these actions can execute concurrently. Both resources are claimed at time stamp 0. Action c starts executing on resource *r1* and simultaneously, action d starts executing on resource *r2*. Action c completes execution at time stamp 1 and resource *r1* is released. Action d completes execution at time stamp 2, resource *r2* is released and activity Act_2 completes execution. Thus, the execution time for activity Act_2 is 2 time units.

Activity Act_3 uses resource *r1* and *r2*, which execute two actions e and f with execution time of 1 and 2 time units respectively. Since there is a requirement that action f must execute after action e, only resource *r1* is claimed at time stamp 0 and action e starts execution. It executes till time stamp 1 and resource *r1* is released. Once action e completes execution, resource *r2* is claimed and action f starts execution. Action f completes execution at time stamp 3, the resource *r2* is released, and activity Act_3 completes execution. Thus, the execution time for activity Act_3 is 3 time units.

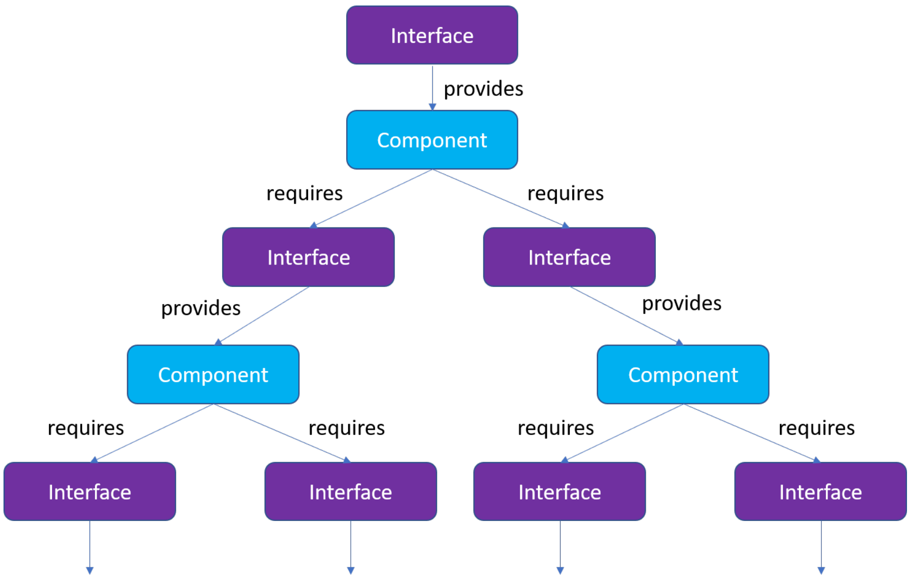


Figure 3.5: Dezyne abstraction levels

3.4 Dezyne Toolset

Dezyne is a model-driven software engineering tool that enables the design of the structure and behavior of a software system. It provides software engineers with a methodology to create, explore and formally verify component based designs for embedded and technical software systems. It provides a complete environment for specifying, designing, testing, generating and building model based software components. Graphical representations of component models in the form of state charts, event tables, system models and sequence charts enable engineers to easily understand, navigate, communicate and document their architectures and designs. Software components built with Dezyne are simple to reuse and easy to extend.

Dezyne has a formally specified semantic which enables the direct translation of a model into mathematical representation which is then subjected to automated analysis using formal methods. If a design error is found, the sequence of events that led to that error is displayed in the Dezyne simulator. [24]

3.4.1 Modeling in Dezyne

A Dezyne component is described using two model types, an *interface* specification model and a *component* implementation model [25]. Interface specifications define the externally visible behavior of a component. Implementation models contain the operational logic of a component which are verified against any interface specification they provide or require, and can therefore, be easily composed into complete hierarchical systems (Figure 3.5).

An *interface* contains the definition of the *events*, their direction (*in* or *out*) and, a specification of the externally visible behavior of the component that will provide an implementation of the interface. It has a *behavior* describing the protocol of its events. The direction of events is determined as seen from the component providing the service. The stimulus that comes into the components is defined by *in* events and the component's response to it is given by *out* events. Components use or provide functionality via interfaces and cannot directly interact with other components. Therefore, interface specifications does not contain component implementation details.

A *component* implements a certain functionality or a component implementation may use the functionality offered by other components. The functionality that is offered by a component to others is defined in that component's interface. A component behaves according to its provided interface and makes use of other components through their required interfaces. Communication between components is performed through their *ports*, which are instances of interfaces. Each port has a direction according to its intention (provides or requires).

The functionality of a component can be used by sending and receiving events to and from the component through ports over the interface. The *behavior* contains a collection of statements that define the *actions* that will be performed based on the events received. The behavior specifies which events can be received and sent at which stages in the execution process, or in other words, the protocol a user of the interface must obey. Actions can change the value of a variable, invoke a function, generate another event or change to another state. In response to one event trigger, there can be different actions.

3.5 Translating an activity model to Dezyne code

An activity in the Dezyne environment is defined using an *interface* and a corresponding *component* for it. The `activity interface` defines the start and end behavior of the activity. It defines the sequence of allowed actions from various resources. These resources have a separate *interface* and a corresponding *component*. The `resource interface` defines the actions which that resource can execute.

Transition into the next or the post action in the *component* is triggered by the completion of previous actions. The action itself is defined in terms of events, in particular, the *in* and *out* events. Events often originate from inside the system, such as finishing of a task. As soon as the incoming transition of an action is triggered, its entry (*in*) event starts executing. Once the entry event has finished execution, the action is considered to be complete. When the action is complete, the outgoing transition is enabled along with an end (*out*) event. This *out* event triggers the execution of subsequent actions.

In case, multiple actions need to finish in order to execute the next action, Boolean variables can be used to keep track of completion status of each action. The completion Boolean is initially set to *false*. When an action completes execution, it is set to *true*. Before execution of the next action, the completion Booleans of all incoming dependent actions are checked and if they are all *true*, then the action starts execution with its *in* event triggered.

The behavior for the software component of the activities in the example cases can be represented in terms of an activity diagram with events mapped to it as shown in Figure 3.7. Activity Act_1 starts with the StartAct1() event. This triggers action *a* to start execution which is the initial action of this activity, and Start_a() event is called. When action *a* completes execution Complete_a() event is returned, which triggers action *b* to start execution. Action *b* starts by calling Start_b() event and forms a post action of action *a* as it executes after *a*. When action *b* finishes Complete_b() event is returned, and this triggers CompleteAct1() event indicating the end of the activity.

Similarly, activity Act_3 starts with the StartAct3() event. This triggers action *e* to start execution which is the initial action of this activity, and Start_e() event is called. When action *e* completes execution Complete_e() event is returned, which triggers action *f* to start execution by triggering Start_f() event. Action *f* is a post action of action *e* as it executes after *e*. When action *f* finishes Complete_f() event is returned, and

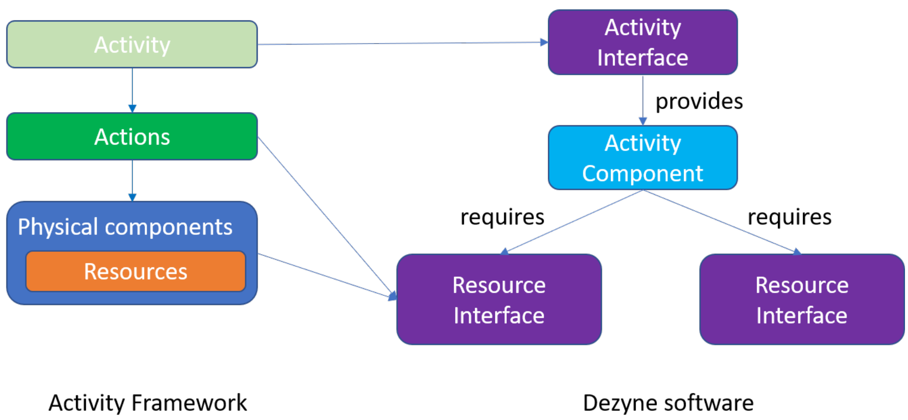


Figure 3.6: Activity model mapped to Dezyne elements

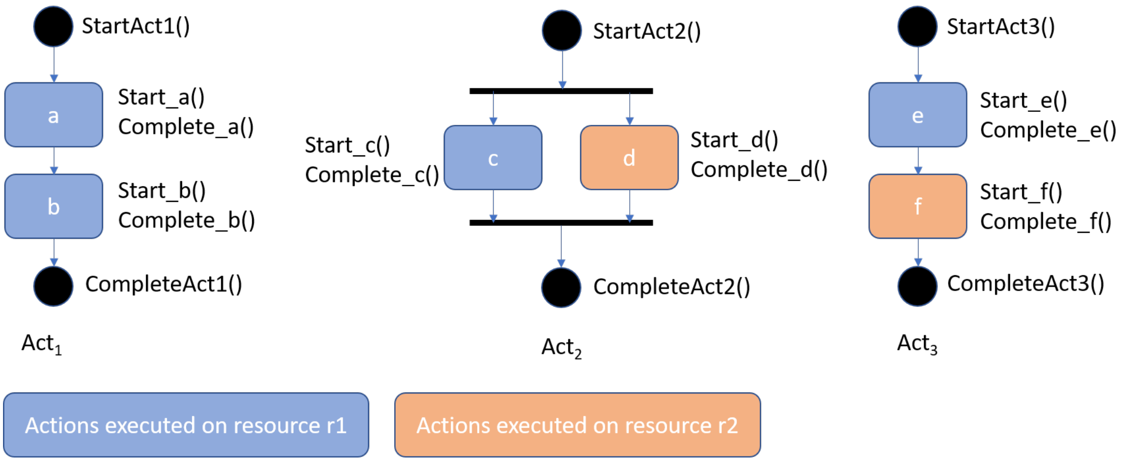


Figure 3.7: Activities Act_1, Act_2 and Act_3 with events

this triggers CompleteAct3() event indicating the end of the activity.

Execution of activity Act_2 differs from the other two activities. Act_2 starts with the StartAct2() event. There are two initial actions *c* and *d* which execute in parallel. The start of the activity triggers execution of Start_c() event and Start_d() event. Since the activity should end only when both actions complete execution, actions *c* and *d* form a set of dependent actions for the completion of the activity. Boolean completion variables are used to keep track of the status of these actions, for instance, c_complete and d_complete. Initially, both variables are set to *false*. When action *c* completes, Complete_c() event is returned and variable c_complete is set to *true*. Similarly, when action *d* completes, Complete_d() event is returned and variable d_complete is set to *true*. When both actions complete and their corresponding Boolean completion variables are set to *true*, then activity completes execution with the event CompleteAct2().

3.5.1 Rules of translation

The mapping of an activity to Dezyne code follows the following set of rules:

1. Activity events ∈ {`Start_A, End_A`}.
   For each activity, to define its start and end behavior in the `activity interface`, a tuple of two events is defined: a start event `Start_A` defined as an *in* event, and an end event `End_A` defined as an *out* event.

2. Activity interface states ∈ {`idle, execute`}.
   Each `activity interface` has two states: `idle` and `execute`. The initial state is the `idle` state. When the activity starts execution with the invocation of start event `Start_A` the state changes to `execute`. Once the execution of the activity completes, the end event `End_A` is called and the state reverts to the `idle` state.

3. Action events ∈ {`start_action_event, end_action_event`}.
   For each action, to define its start and end behavior in the `resource interface`, a tuple of two events is defined: a start event `start_action_event` defined as an *in* event, and an end event `end_action_event` defined as an *out* event.

4. Action status variable ∈ {`true, false`}.
   For each action, a Boolean variable `action_complete` is defined in the `activity component` to represent its execution status; if an action completes execution its value corresponds to *true* otherwise its *false*.

5. Dependencies between actions ∈ {`Initial actions, dep, post`}.
   `Initial actions` are the actions triggered by the start of the activity.

Dep(a) is a set of actions that need to complete before action a can start.
Post(a) is a set of actions triggered by the completion of action a.

6. Activity component states ∈ {idle, execute}.
   Each activity component has two states: idle and execute.

   (a) The initial state is the idle state. In the idle state, when the start event of the activity Start_A is called, it triggers the execution of start_action_event of the corresponding initial actions and the state changes to execute.

   (b) In the execute state, the sequence of actions corresponding to different resources, following the completion of initial actions, is executed. This sequence of actions is defined using a set of dependencies. If an action completes execution, the end_action_event is triggered and its corresponding Boolean completion variable is set to *true*. Then, its dependent actions (dep) are checked for completion, and once completed, all post actions (post) are triggered for execution using their corresponding start_action_event. Once all actions complete execution, EndActivity function is invoked. This function contains the end event of the activity End_A and a Reset function. The Reset function sets the action completion variables, action_complete back to *false*, and reverts the activity component state to idle state.

3.5.2 Algorithm for translation

Using the above rules, an algorithm that translates an activity to the Dezyne code is given below.

1. ***Interface***
   Inputs: name of the activity <Activity_Name>, and two events defining the start and end of the activity: <Start_A>,<End_A>

```
interface I<Activity_Name> {

   //Define start and end events
   in void <Start_A>();
   out void <End_A>();

   behaviour {
      //Define two states
      enum Activity_states_t {IDLE, EXECUTE};
      //Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      [state.IDLE] {
         //Define behaviour for start event of activity
         on <Start_A>: {
            state = Activity_states_t.EXECUTE;
         }
      }

      [state.EXECUTE] {
         on <Start_A>: illegal;
         on inevitable: {
            //Return end event of activity
            <End_A>;
            state = Activity_states_t.IDLE;
         }
      }
   }
}
```

2. ***Component***

   Resource `<resource_name>` provides interface `<interface_name>` with the action `<action_name>` which starts with `<start_action_event>` and ends with `<end_action_event>`.

```
import <resource_interface_name>.dzn;

component <Activity_Name>_Comp {

   provides I<Activity_Name> p_<Activity_Name>;
   requires <resource_interface_name> r_<resource_name>;

   behaviour {

      // Define two states
      enum Activity_states_t {IDLE, EXECUTE};
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;
      // For every action <action_name> define a boolean variable
      bool <resource_name>_<action_name>_complete = false;

      void Reset()
      {
         state = Activity_states_t.IDLE;
         <resource_name>_<action_name>_complete = false;
      }

      void EndActivity() {
         p_<Activity_Name>.<End_A>();
         Reset();
      }

      [state.IDLE] {

         // Define behaviour for start event of activity
         on p_<Activity_Name>.<Start_A>(): {
            state = Activity_states_t.EXECUTE;

            // Insert code to start the first actions
            // For every action in initials
            r_<resource_name>.<start_action_event>();

         }
      }

      [state.EXECUTE] {

         // Generate code following the activity dependencies
         // Given a tuple <a, dep, post>
         // a = an action of resource named <resource_name>
         // dep = all actions that need to be completed before a can start
         // post = all actions that need to be started when a completes

         on r_<resource_name>.<end_action_event>(): {

            // For action a, change boolean to indicate a finishes execution
            <resource_name>_<action_name>_complete = true;

            if (<all actions in dep have their boolean to true>) {

                  // If post is not empty, start all post actions
                  // For all actions in post
                  r_<resource_name>.<start_action_event>();

                  // If post is empty, then complete the activity
                  EndActivity();
            }
         }
```

```
        }
    }
}
```

3.6 Developing a Domain Specific Language (DSL) to define translation model: Transformation DSL

After specifying activities in the Activity DSL, in order to generate Dezyne code, additional information is needed which is specified in another DSL, called the **Transformation DSL**. This DSL defines translation models. In a translation model, for each **activity**, its **start event** and **end event** are specified. For each **resource** used within the activity, a resource **interface** is mentioned, and for each **action** executed in an activity on that resource, a **start event** and an **end event** is specified. The translation model for the example cases is shown below.

```
Activity Act1
StartEvent: StartAct1()
EndEvent: CompleteAct1()

Activity Act2
StartEvent:StartAct2()
EndEvent: CompleteAct2()

Activity Act3
StartEvent: StartAct3()
EndEvent: CompleteAct3()

Resource r1
Interface: Ir1

Action a
StartEvent: Start_a()
EndEvent: Complete_a()

Action b
StartEvent: Start_b()
EndEvent: Complete_b()

Action c
StartEvent: Start_c()
EndEvent: Complete_c()

Action e
StartEvent: Start_e()
EndEvent: Complete_e()

Resource r2
Interface: Ir2

Action d
StartEvent: Start_d()
EndEvent: Complete_d()

Action f
StartEvent: Start_f()
EndEvent: Complete_f()
```

3.7 Results

After specifying an activity model and a translation model in the Activity DSL and the Transformation DSL respectively for the example cases, Dezyne code is generated as shown in figure 3.8. The Dezyne code for resources *r1* and *r2* is given in section A.1.1 for reference.

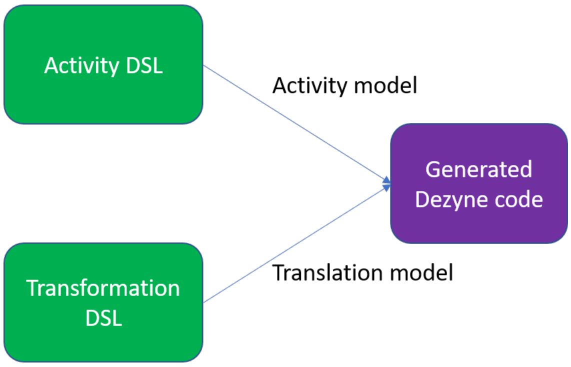


Figure 3.8: Dezyne code generation from Activity DSL and Transformation DSL

1. For Act_1 the generated code is shown below.

   (a) ***Interface***

```
interface IAct1 {
   // Define start and end events
   in void StartAct1();
   out void CompleteAct1();

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      [state.IDLE] {
         // Define behaviour for start event of activity
         on StartAct1: {
            state = Activity_states_t.EXECUTE;
         }
      }

      [state.EXECUTE] {
         on StartAct1: illegal;
         on inevitable: {
            // Return end event of activity
            CompleteAct1;
            state = Activity_states_t.IDLE;
         }
      }
   }
}
```

   (b) ***Component***

```
import IAct1.dzn;
import Ir1.dzn;

component Act1_Comp {
   provides IAct1 p_Act1;
   requires Ir1 r_r1;

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      // For every action define a boolean variable
      bool r1_a_complete = false;
```

```
      bool r1_b_complete = false;

      void Reset() {
         state = Activity_states_t.IDLE;
         r1_a_complete = false;
         r1_b_complete = false;
      }

      void EndActivity() {
         p_Act1.CompleteAct1();
         Reset();
      }

      [state.IDLE] {
         // Define behaviour for start event of activity
         on p_Act1.StartAct1(): {
            state = Activity_states_t.EXECUTE;

            // Insert code to start the first actions
            // For every action in initials
            r_r1.Start_a();
         }
      }

      [state.EXECUTE] {
         on r_r1.Complete_a(): {
            r1_a_complete = true;
            r_r1.Start_b();
         }

         on r_r1.Complete_b(): {
            r1_b_complete = true;
            EndActivity();
         }
      }
   }
}
```

2. For Act_2 the generated Dezyne code is shown below.

   (a) ***Interface***

```
interface IAct2 {
   // Define start and end events
   in void StartAct2();
   out void CompleteAct2();

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      [state.IDLE] {
         // Define behaviour for start event of activity
         on StartAct2: {
            state = Activity_states_t.EXECUTE;
         }
      }

      [state.EXECUTE] {
         on StartAct2: illegal;
         on inevitable: {
            // Return end event of activity
            CompleteAct2;
            state = Activity_states_t.IDLE;
         }
```

```
        }
    }
}
```

(b) ***Component***

```
import IAct2.dzn;
import Ir1.dzn;
import Ir2.dzn;

component Act2_Comp {
   provides IAct2 p_Act2;
   requires Ir1 r_r1;
   requires Ir2 r_r2;

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      // For every action define a boolean variable
      bool r1_c_complete = false;
      bool r2_d_complete = false;

      void Reset() {
         state = Activity_states_t.IDLE;
         r1_c_complete = false;
         r2_d_complete = false;
      }

      void EndActivity() {
         p_Act2.CompleteAct2();
         Reset();
      }

      [state.IDLE] {
         // Define behaviour for start event of activity
         on p_Act2.StartAct2(): {
            state = Activity_states_t.EXECUTE;

            // Insert code to start the first actions
            // For every action in initials
            r_r1.Start_c();
            r_r2.Start_d();
         }
      }

      [state.EXECUTE] {
         on r_r1.Complete_c(): {
            r1_c_complete = true;
            if (r2_d_complete) {
               EndActivity();
            }
         }

         on r_r2.Complete_d(): {
            r2_d_complete = true;
            if (r1_c_complete) {
               EndActivity();
            }
         }
      }
   }
}
```

3. For Act_3 the generated Dezyne code is shown below.

   (a) ***Interface***

```
interface IAct3 {
   // Define start and end events
   in void StartAct3();
   out void CompleteAct3();

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      [state.IDLE] {
         // Define behaviour for start event of activity
         on StartAct3: {
            state = Activity_states_t.EXECUTE;
         }
      }

      [state.EXECUTE] {
         on StartAct3: illegal;
         on inevitable: {
            // Return end event of activity
            CompleteAct3;
            state = Activity_states_t.IDLE;
         }
      }
   }
}
```

   (b) ***Component***

```
import IAct3.dzn;
import Ir1.dzn;
import Ir2.dzn;

component Act3_Comp {
   provides IAct3 p_Act3;
   requires Ir1 r_r1;
   requires Ir2 r_r2;

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      // For every action define a boolean variable
      bool r1_e_complete = false;
      bool r2_f_complete = false;

      void Reset() {
         state = Activity_states_t.IDLE;
         r1_e_complete = false;
         r2_f_complete = false;
      }

      void EndActivity() {
         p_Act3.CompleteAct3();
         Reset();
      }

      [state.IDLE] {
         // Define behaviour for start event of activity
         on p_Act3.StartAct3(): {
```

```
                state = Activity_states_t.EXECUTE;

                // Insert code to start the first actions
                // For every action in initials
                r_r1.Start_e();
            }
        }

        [state.EXECUTE] {
            on r_r1.Complete_e(): {
                r1_e_complete = true;
                r_r2.Start_f();
            }

            on r_r2.Complete_f(): {
                r2_f_complete = true;
                EndActivity();
            }
        }
    }
}
```

Sequence diagrams of the generated Dezyne code

The behavior of the generated code is shown as a sequence diagram for activities Act_1, Act_2 and Act_3 in Figure 3.9, 3.10 and 3.11 respectively. It can be seen that the behavior obtained is same as in Figure 3.4.

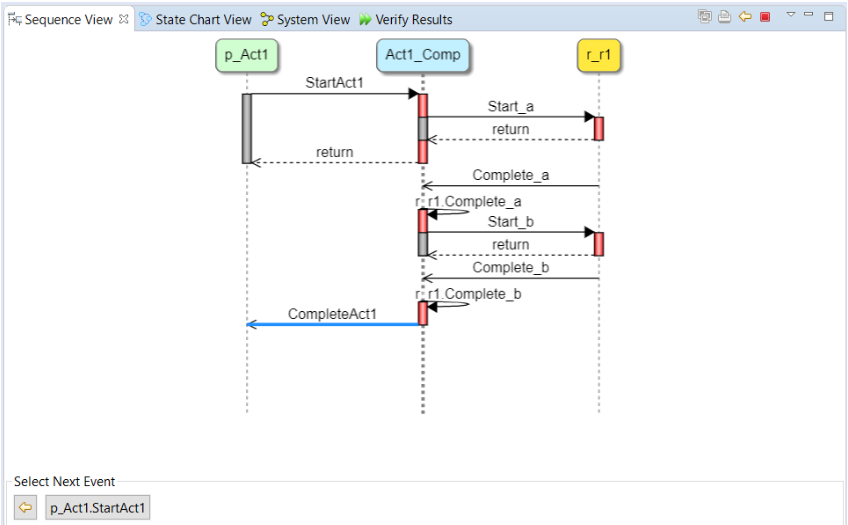


Figure 3.9: Sequence diagram for activity Act_1

3.8 Validation

Validation ensures that the output behavior obtained from the implementation is same as the specifications defined for the input.

3.8.1 Correctness of translation made from activity model to the Dezyne code

The systematic translation from the activity model to the generated Dezyne code is considered accurate if the behavior obtained from generated Dezyne code is identical to the specification of an activity. This is validated using Gantt charts. The Gantt chart for an activity is obtained from the LSAT model, and is

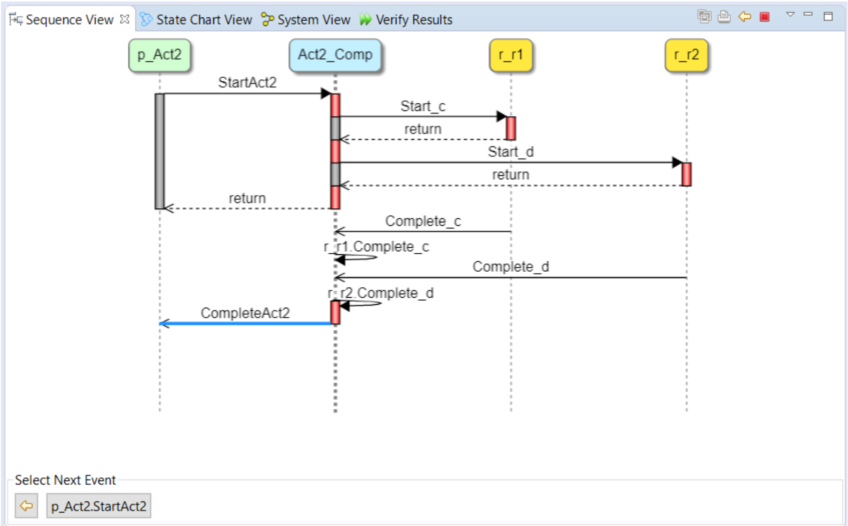


Figure 3.10: Sequence diagram for activity Act_2

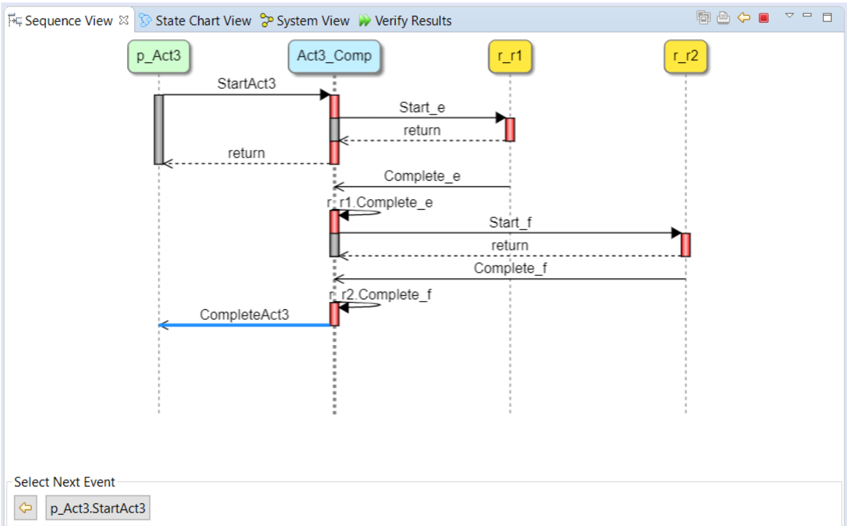


Figure 3.11: Sequence diagram for activity Act_3

compared to the Gantt chart obtained from the sequence diagram of the generated code. In both cases, the behavior exhibited must be same.

For activity Act_1, Act_2 and Act_3, the Gantt chart obtained from LSAT are shown in Figures 3.12, 3.13 and 3.14.

The Gantt charts obtained from the Dezyne code generated are shown in Figure 3.15. It can be seen that the Gantt charts obtained from LSAT and the Gantt charts obtained from Dezyne have similar behavior in terms of ordering of actions. Hence, the translation made from the activity model to the Dezyne code is successfully validated.

3.8.2 Scalability analysis

With each addition of a new action in an activity, it is expected that the state-space increases and so will the length of the generated Dezyne code. However, this is not the case. For each action added in an activity, in the Activity DSL an *Action Type*, action type's *instance* and the action *Dependencies* are added. In the Transformation DSL, the *events* for the action is defined. This translates to a generated code with code

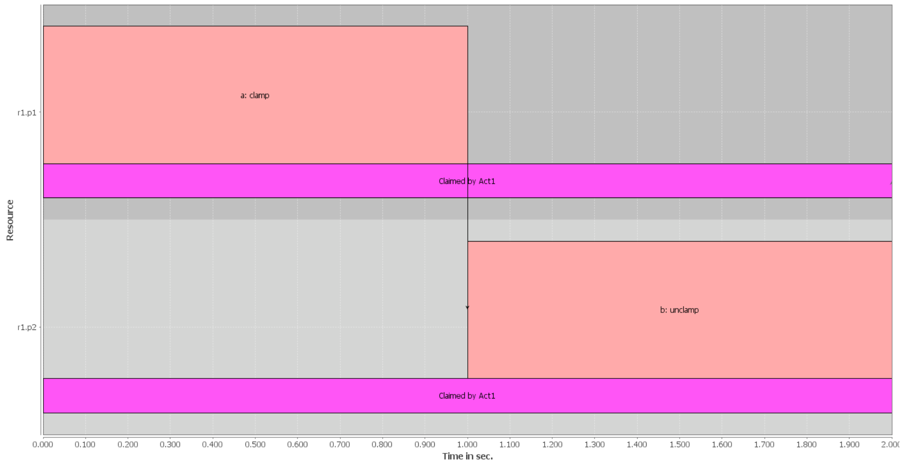


Figure 3.12: Gantt chart for activity Act_1 obtained from the LSAT model

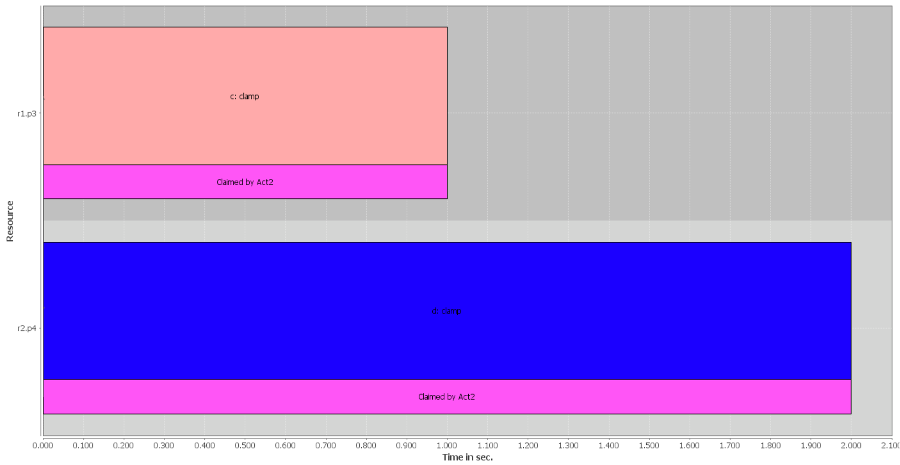


Figure 3.13: Gantt chart for activity Act_2 obtained from the LSAT model

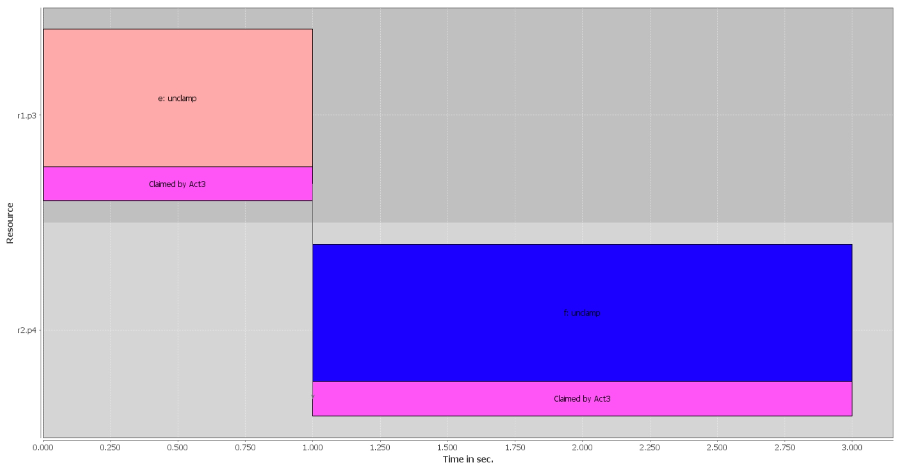


Figure 3.14: Gantt chart for activity Act_3 obtained from the LSAT model

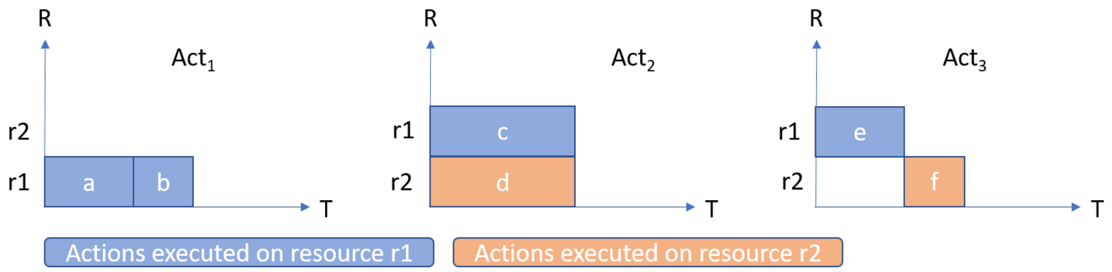


Figure 3.15: Gantt chart for activity Act_1, Act_2 and Act_3 obtained from Dezyne code

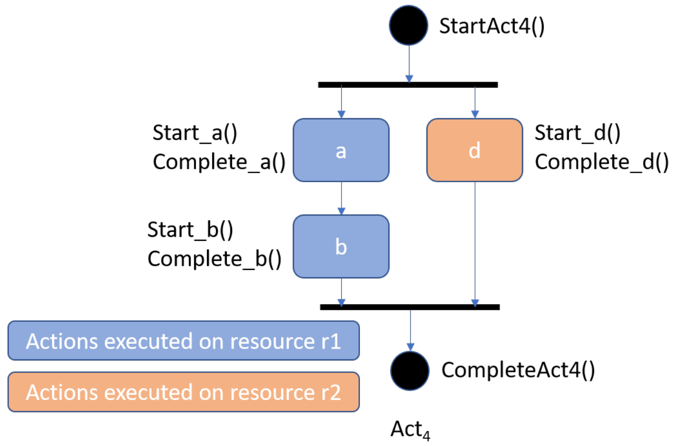


Figure 3.16: Activity diagram for activity Act_4

length increasing by only few lines.

To validate this, a new activity Act_4 is modeled. This activity is similar to activity Act_1, with one more action added to it. Act_4 has three actions *a*, *b* and *d*. Actions *a* and *b* execute on resource *r1*, whereas action *d* runs on resource *r2*. The activity diagram representing the dependencies between actions for activity Act_4 is shown in Figure 3.16.

The Dezyne code is generated for activity Act_4 which is verified in the Dezyne environment. The result of verification for the *component* of activity Act_4 is shown in Figure 3.17. The verification result for the *component* of activity Act_1 is also shown for comparison purposes. It can be seen that the number of states in the *component* of activity Act_1 is 17 while for activity Act_4 it is 39, which is almost double the number of states. However, the Dezyne code length does not increase proportionally with the increase in state space. The total lines of code in the *component* of activity Act_1 is 52 (refer to activity component on page 22) while for activity Act_4 it is 66 (as shown below). This is because for each additional action, only a single Boolean variable to represent its completion status, and one *in* and one *out* event to define the dependency on other actions are added. Hence, it is seen that while the state space increases exponentially the increase is code length does not. Nevertheless, with each action added in an activity, the readability of the generated code reduces. **Component** for activity Act_4 is shown below.

```
import IAct4.dzn;
import Ir1.dzn;
import Ir2.dzn;

component Act4_Comp {
   provides IAct4 p_Act4;
   requires Ir1 r_r1;
   requires Ir2 r_r2;

   behaviour {
```

```
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      // For every action define a boolean variable
      bool r1_a_complete = false;
      bool r1_b_complete = false;
      bool r2_d_complete = false;

      void Reset() {
         state = Activity_states_t.IDLE;
         r1_a_complete = false;
         r1_b_complete = false;
         r2_d_complete = false;
      }

      void EndActivity() {
         p_Act4.CompleteAct4();
         Reset();
      }

      [state.IDLE] {
         // Define behaviour for start event of activity
         on p_Act4.StartAct4(): {
            state = Activity_states_t.EXECUTE;

            // Insert code to start the first actions
            // For every action in initials
            r_r1.Start_a();
            r_r2.Start_d();
         }
      }

      [state.EXECUTE] {
         on r_r1.Complete_a(): {
            r1_a_complete = true;
            r_r1.Start_b();
         }

         on r_r1.Complete_b(): {
            r1_b_complete = true;
            if (r2_d_complete) {
               EndActivity();
            }
         }

         on r_r2.Complete_d(): {
            r2_d_complete = true;
            if (r1_b_complete) {
               EndActivity();
            }
         }
      }
   }
}
```

Verify Results

Act1_Component.dzn

Check	Action	Time	States	Transitions	Done	Result
IAct1						
Deadlock		0:00	7	8	100%	✓
Livelock		0:00	7	8	100%	✓
Ir1						
Deadlock		0:00	16	35	100%	✓
Livelock		0:00	16	35	100%	✓
Act1_Comp						
Deterministic		0:00	17	19	100%	✓
Illegal		0:00	17	19	100%	✓
Deadlock		0:00	17	19	100%	✓
Livelock		0:00	17	19	100%	✓
Compliance		0:00	17	19	100%	✓

Component for Act_1

Act4_Component.dzn

Check	Action	Time	States	Transitions	Done	Result
IAct4						
Deadlock		0:00	7	8	100%	✓
Livelock		0:00	7	8	100%	✓
Ir1						
Deadlock		0:00	16	35	100%	✓
Livelock		0:00	16	35	100%	✓
Ir2						
Deadlock		0:00	10	15	100%	✓
Livelock		0:00	10	15	100%	✓
Act4_Comp						
Deterministic		0:00	39	46	100%	✓
Illegal		0:00	39	46	100%	✓
Deadlock		0:00	39	46	100%	✓
Livelock		0:00	39	46	100%	✓
Compliance		0:00	39	46	100%	✓

Component for Act_4

Figure 3.17: Verification results showing the number of states for activity Act_1 and Act_4

3.9 Verification of the generated Dezyne code

Since the activity framework has no methodology for verification of activities, for this project, refinement verification at the software level is used. In this methodology, it is checked whether the functionality of an abstract system model is correctly implemented by a low-level implementation. The behavior of the abstract model is translated into the behavior of the given interfaces and structures at the software design level. This makes it possible to verify small parts of the low-level design in the context of the abstract model. [26]

The activities modeled are verified in the Dezyne environment. Verification in Dezyne focuses on verifying a component together with its provided and required interfaces. This ensures that the component behaves correctly in its environment according to the specified behavior. The following properties are verified in Dezyne: [27]

1. Completeness: It is a required that in every state of a model each event is enabled, either by being unguarded, or by having a guard that evaluates to true for the given state.
2. Deterministic: Dezyne cannot handle non-determinism. All components are required to be deterministic in nature. If a component has overlapping guards, that is, two different sets of actions for the same event are specified, this will lead to non-determinism.
3. Illegal: It is required that there are no protocol violations between a component and its required interfaces. If there is, an error is reported.
4. Range error: An integer type variable must always have a value that lies within its defined range. It not, a range error is reported.
5. Type error: If an event has a return type defined, a value of the same return type must be replied, else it leads to a type error.
6. Queue full: A Dezyne model with an interface of type provides defines a port that has a queue where notification events are stored before they are processed. It is checked that this queue does not overflow and remains non-blocking.
7. Deadlock: A deadlock occurs when none of the components in a system can make progress and the system simply does not respond. It may occur when a component is waiting for some external

Verify Results

Act1_Component.dzn

Check	Action	Time	States	Transitions	Done	Result
IAct1						
Deadlock		0:00	7	8	100%	✓
Livelock		0:00	7	8	100%	✓
Ir1						
Deadlock		0:00	16	35	100%	✓
Livelock		0:00	16	35	100%	✓
Act1_Comp						
Deterministic		0:00	17	19	100%	✓
Illegal		0:00	17	19	100%	✓
Deadlock		0:00	17	19	100%	✓
Livelock		0:00	17	19	100%	✓
Compliance		0:00	17	19	100%	✓

Figure 3.18: Verification result for activity Act_1

Verify Results

Act2_Component.dzn

Check	Action	Time	States	Transitions	Done	Result
IAct2						
Deadlock		0:00	7	8	100%	✓
Livelock		0:00	7	8	100%	✓
Ir1						
Deadlock		0:00	16	35	100%	✓
Livelock		0:00	16	35	100%	✓
Ir2						
Deadlock		0:00	10	15	100%	✓
Livelock		0:00	10	15	100%	✓
Act2_Comp						
Deterministic		0:00	24	28	100%	✓
Illegal		0:00	24	28	100%	✓
Deadlock		0:00	24	28	100%	✓
Livelock		0:00	24	28	100%	✓
Compliance		0:00	24	28	100%	✓

Figure 3.19: Verification result for activity Act_2

Verify Results

Act3_Component.dzn

Check	Action	Time	States	Transitions	Done	Result
IAct3						
Deadlock		0:00	7	8	100%	✓
Livelock		0:00	7	8	100%	✓
Ir1						
Deadlock		0:00	16	35	100%	✓
Livelock		0:00	16	35	100%	✓
Ir2						
Deadlock		0:00	10	15	100%	✓
Livelock		0:00	10	15	100%	✓
Act3_Comp						
Deterministic		0:00	17	19	100%	✓
Illegal		0:00	17	19	100%	✓
Deadlock		0:00	17	19	100%	✓
Livelock		0:00	17	19	100%	✓
Compliance		0:00	17	19	100%	✓

Figure 3.20: Verification result for activity Act_3

event which fails to occur or when two components require an action from each other before they can perform any further action themselves.

8. Compliance: The compliance property checks whether the component together with the required interfaces implements the behavior specified in the provided interfaces.

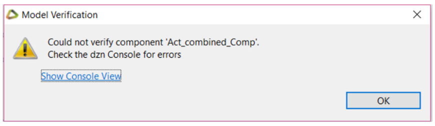
9. Livelock: A component is said to be livelocked when it is permanently busy with internal behavior and ceases to serve clients specified by the provided interface. This is similar to deadlock except that a deadlocked component does not perform any actions whereas a livelocked component might be performing lots of actions, but none of them are visible to the component's clients.

The verification results for Act_1, Act_2 and Act_3 are shown in Figure 3.18, Figure 3.19 and Figure 3.20 respectively. Since there is a state-space explosion problem when the number of actions in an activity grow too large, the verification is hampered for larger activities. One such case is *Act_combined* which activity model is given below.

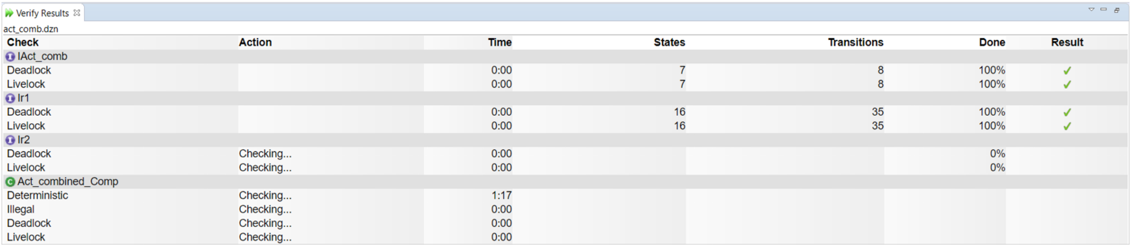
```

Resource r1
   Action type a
   Action type b
   Action type c
   Action type e
Resource r2
   Action type d
   Action type f

Activity Act_combined
   Criticality level 0
```

```
a : r1.a
b : r1.b
c : r1.c
d : r2.d
e : r1.e
f : r2.f
a1 : r1.a
b1 : r1.b
c1 : r1.c
d1 : r2.d
e1 : r1.e
f1 : r2.f
a2 : r1.a
b2 : r1.b
c2 : r1.c
d2 : r2.d
e2: r1.e
f2 : r2.f
a3 : r1.a
b3 : r1.b
c3 : r1.c
d3 : r2.d
e3 : r1.e
f3 : r2.f
Dependencies
a -> b
b -> c
b -> d
c -> e
d -> e
e -> f
f -> a1
a1 -> b1
b1 -> c1
b1 -> d1
c1 -> e1
d1 -> e1
e1 -> f1
f1 -> a2
a2 -> b2
b2 -> c2
b2 -> d2
c2 -> e2
d2 -> e2
e2 -> f2
f2 -> a3
a3 -> b3
b3 -> c3
b3 -> d3
c3 -> e3
d3 -> e3
e3 -> f3
```

As expected, verification fails for activity *Act_combined* because the verifier runs out of memory and cannot allocate memory for actions anymore (Figure 3.21). Thus, with increasing size of activities, verification is hampered.



Verification failed

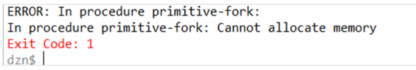


Verify Results

act_comb.dzn

Check	Action	Time	States	Transitions	Done	Result
IAct_comb						
Deadlock		0:00	7	8	100%	✓
Livelock		0:00	7	8	100%	✓
Ir1						
Deadlock		0:00	16	35	100%	✓
Livelock		0:00	16	35	100%	✓
Ir2						
Deadlock	Checking...	0:00			0%	
Livelock	Checking...	0:00			0%	
Act_combined_Comp						
Deterministic	Checking...	1:17				
Illegal	Checking...	0:00				
Deadlock	Checking...	0:00				
Livelock	Checking...	0:00				

Incomplete verification results



Verification error message

Figure 3.21: Verification result for activity *Act_combined*

Chapter 4

Case Study: Translation from Activity model to Dezyne code

The translation from the activity models to the Dezyne code is achieved using the following five steps:

1. Model an FMS using the activity framework.
2. Define the start and end events for the activity, and actions contained in that activity to specify the software component for the activity model.
3. Model the system in the Activity DSL and in the Transformation DSL by specifying the activity model and translation model for it.
4. Generate Dezyne code from the model.
5. Validate the translation made from the activity model to the Dezyne code.
6. Verify the generated code in the Dezyne environment.

4.1 Modeling an FMS using the activity framework

The first step in the translation of the activity model to the Dezyne code is to model the Factory Four model using the activity framework. The Factory Four model has four resources: a sorting line with color recognition, a multi-processing station with oven, an automated high-bay warehouse and a vacuum suction robot, executing two main sequences: the Warehouse sequence and the Color Sorter sequence.

In the *Warehouse sequence*, the vacuum suction robot sends a request to the high-bay warehouse to retrieve a widget contained in a box and waits for its acknowledgement. Simultaneously, the vacuum suction robot sends a request to the processing station and waits for its response to verify if the processing station is ready to receive the widget. Once the high-bay warehouse places the widget on its conveyor belt, it sends the response back to the vacuum suction robot informing that the widget is available to be picked up from its conveyor belt. When the multi-processing station is ready, the vacuum suction robot picks up the widget from the high-bay warehouse. The empty box at the conveyor belt is moved back to its storage location in the high-bay warehouse and, in parallel, the vacuum suction robot moves the picked widget to the processing station. Once the vacuum suction robot places the widget at the processing station, it moves back to its safe position.

In the activity framework, the behavior of moving a widget from the high-bay warehouse to the multi-processing station by the vacuum suction robot represents an **activity**. This activity is a directed acyclic graph which consists of **actions** such as pick up the widget by vacuum suction robot. The **resources** used to implement this activity are the high-bay warehouse, the vacuum suction robot, and the multi-processing station. This activity is represented using an activity diagram in Figure 4.1.

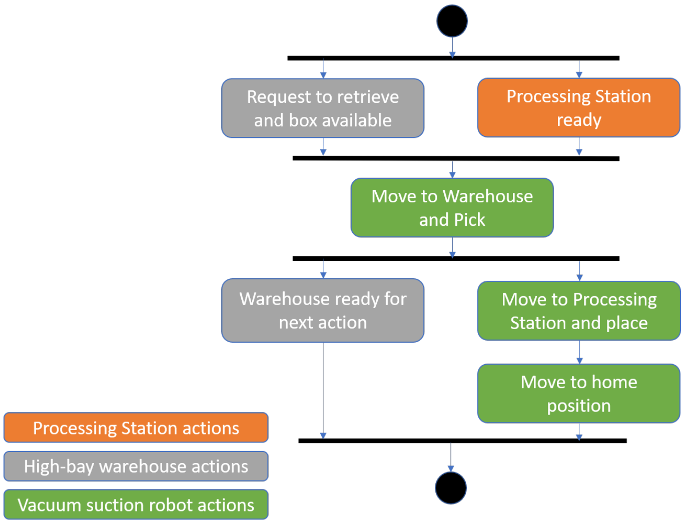


Figure 4.1: Activity diagram for the *Warehouse Activity*

In the *Color Sorter sequence*, the vacuum suction robot sends a request to the color sorter to retrieve a widget and waits for its acknowledgement. Simultaneously, the vacuum suction robot sends a request to the warehouse and waits for its response to verify if the warehouse is ready to receive the widget. Once the high-bay warehouse places the empty box on its conveyor belt, it sends the response back to the vacuum suction robot informing that the box is available for storing the widget. The vacuum suction robot then picks up the widget from the color sorter and moves the picked widget to the warehouse. Next, the vacuum suction robot places the widget at the warehouse. The box with the widget at the conveyor belt is moved back to its storage location in the high-bay warehouse and, in parallel, the vacuum suction robot moves back to its safe position.

In the activity framework, the behavior of moving a widget from the color sorter to the high-bay warehouse by the vacuum suction robot represents an **activity**. This activity is a directed acyclic graph which consists of **actions** such as place the widget by vacuum suction robot. The **resources** used to implement this activity are the high-bay warehouse, the vacuum suction robot, and the color sorter. This activity is represented as an activity diagram in Figure 4.2.

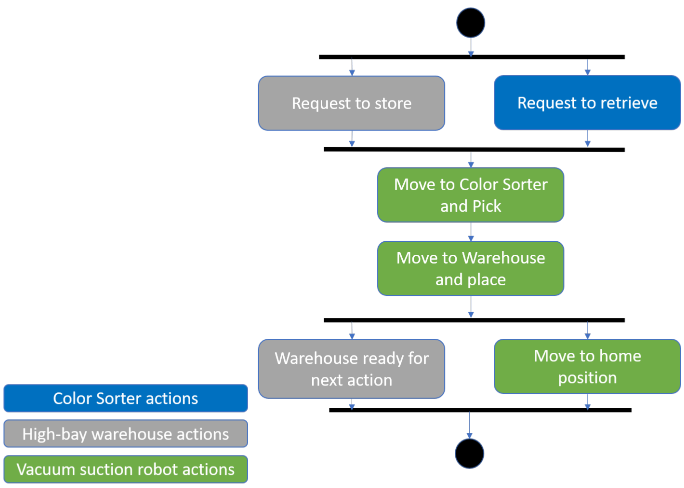


Figure 4.2: Activity diagram for the *Color Sorter Activity*

4.2 Defining the events

The start and end events for the activity and the actions within that activity are specified. Figure 4.3 illustrates the actions and their corresponding events for the *Warehouse Activity*.

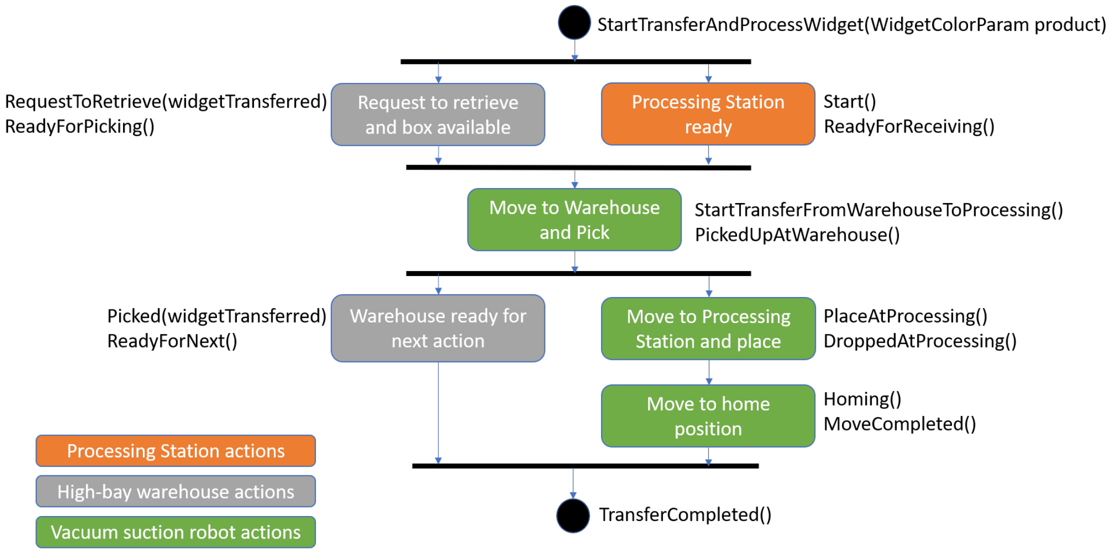


Figure 4.3: Events in Dezyne mapped to the *Warehouse Activity*

Figure 4.4 illustrates the actions and their corresponding events for the *Color Sorter Activity*.

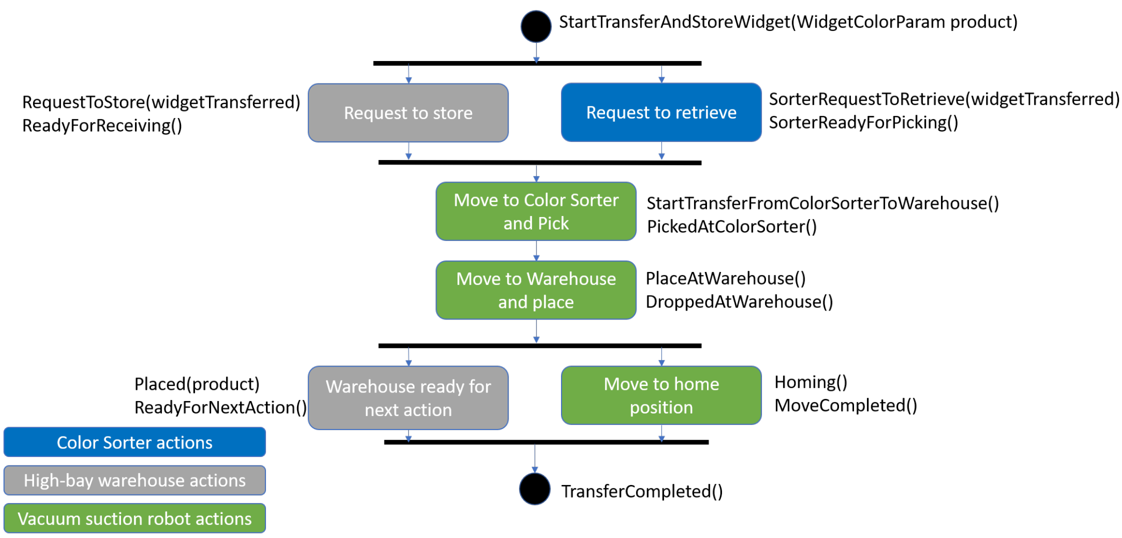


Figure 4.4: Events in Dezyne mapped to the *Color Sorter Activity*

4.3 Modeling the system in Activity DSL and Transformation DSL

Next, the Factory Four model is modeled in the DSLs by defining the activity model and the translation model.

4.3.1 Activity model

The activity model expressed in the Activity DSL for the Factory Four model is shown below.

```

Resource Robot
   Action type MoveToWarehouseAndPick
   Action type MoveToOvenAndPlace
   Action type MoveToHomePosition
   Action type MoveToColorSorterAndPick
   Action type MoveToWarehouseAndPlace
Resource Warehouse
   Action type RequestToRetrieve
   Action type RequestToStore
   Action type ReadyForNext
   Action type ReadyForNextAction
Resource ProcessingStation
   Action type Ready
Resource ColorSorter
   Action type SorterRequestToRetrieve

Activity WarehouseActivity
   Criticality level 0
   a1 : Warehouse.RequestToRetrieve
   a2 : ProcessingStation.Ready
   a3 : Robot.MoveToWarehouseAndPick
   a4 : Warehouse.ReadyForNext
   a5 : Robot.MoveToOvenAndPlace
   a6 : Robot.MoveToHomePosition
   Dependencies
   a1 -> a3
   a2 -> a3
   a3 -> a4
   a3 -> a5
   a5 -> a6

Activity ColorSorterActivity
   Criticality level 0
   b1 : Warehouse.RequestToStore
   b2 : ColorSorter.SorterRequestToRetrieve
   b3 : Robot.MoveToColorSorterAndPick
   b4 : Robot.MoveToWarehouseAndPlace
   b5 : Warehouse.ReadyForNextAction
   b6 : Robot.MoveToHomePosition
   Dependencies
   b1 -> b3
   b2 -> b3
   b3 -> b4
   b4 -> b5
   b4 -> b6
```

4.3.2 Translation model

The translation model expressed in the Transformation DSL for the Factory Four model is shown below.

```
Activity WarehouseActivity
StartEvent: StartTransferAndProcessWidget(WidgetColorParam product)
EndEvent: TransferCompleted()

Activity ColorSorterActivity
StartEvent: StartTransferAndStoreWidget(WidgetColorParam product)
EndEvent: TransferCompleted()

Resource Robot
Interface: IRobot

Action MoveToWarehouseAndPick
```

```
StartEvent: StartTransferFromWarehouseToProcessing()
EndEvent: PickedUpAtWarehouse()

Action MoveToOvenAndPlace
StartEvent: PlaceAtProcessing()
EndEvent: DroppedAtProcessing()

Action MoveToHomePosition
StartEvent: Homing()
EndEvent: MoveCompleted()

Action MoveToColorSorterAndPick
StartEvent: StartTransferFromColorSorterToWarehouse()
EndEvent: PickedAtColorSorter()

Action MoveToWarehouseAndPlace
StartEvent: PlaceAtWarehouse()
EndEvent: DroppedAtWarehouse()

Resource Warehouse
Interface: IWarehouse

Action RequestToRetrieve
StartEvent: RequestToRetrieve(WidgetColorParam widgetTransferred)
EndEvent: ReadyForPicking()

Action ReadyForNext
StartEvent: Picked(widgetTransferred)
EndEvent: ReadyForNext()

Action RequestToStore
StartEvent: RequestToStore(WidgetColorParam widgetTransferred)
EndEvent: ReadyForReceiving()

Action ReadyForNextAction
StartEvent: Placed(WidgetColorParam product)
EndEvent: ReadyForNextAction()

Resource ProcessingStation
Interface: IProcessingStation

Action Ready
StartEvent: Start()
EndEvent: ReadyForReceiving()

Resource ColorSorter
Interface: IColorSorter

Action SorterRequestToRetrieve
StartEvent: SorterRequestToRetrieve(WidgetColorParam widgetTransferred)
EndEvent: SorterReadyForPicking()
```

4.4 Generated Dezyne code

Once the specifications for the Factory Four model is defined and the events are specified, the Dezyne code can be generated. For the *Warehouse activity* and the *Color Sorter activity*, the generated Dezyne code is shown below. The interface code for the resources of Factory Four model is given in A.2.1 for reference.

1. *Warehouse activity*

(a) **Interface**

```
import Definitions.dzn;

interface IWarehouseActivity {
   // Define start and end events
   in void StartTransferAndProcessWidget(WidgetColorParam product);
   out void TransferCompleted();

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      [state.IDLE] {
         // Define behaviour for start event of activity
         on StartTransferAndProcessWidget: {
            state = Activity_states_t.EXECUTE;
         }
      }

      [state.EXECUTE] {
         on StartTransferAndProcessWidget: illegal;
         on inevitable: {
            // Return end event of activity
            TransferCompleted;
            state = Activity_states_t.IDLE;
         }
      }
   }
}
```

(b) **Component**

```
import IWarehouseActivity.dzn;
import IRobot.dzn;
import IWarehouse.dzn;
import IProcessingStation.dzn;

component WarehouseActivity_Comp {
   provides IWarehouseActivity p_WarehouseActivity;
   requires IRobot r_Robot;
   requires IWarehouse r_Warehouse;
   requires IProcessingStation r_ProcessingStation;

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      WidgetColorParam widgetTransferred;

      // For every action define a boolean variable
      bool processingStation_Ready_complete = false;
      bool robot_MoveToHomePosition_complete = false;
      bool robot_MoveToOvenAndPlace_complete = false;
      bool robot_MoveToWarehouseAndPick_complete = false;
      bool warehouse_ReadyForNextAction_complete = false;
      bool warehouse_RequestToRetrieve_complete = false;

      void Reset() {
         state = Activity_states_t.IDLE;
         processingStation_Ready_complete = false;
         robot_MoveToHomePosition_complete = false;
```

```
         robot_MoveToOvenAndPlace_complete = false;
         robot_MoveToWarehouseAndPick_complete = false;
         warehouse_ReadyForNextAction_complete = false;
         warehouse_RequestToRetrieve_complete = false;
      }

      void EndActivity() {
         p_WarehouseActivity.TransferCompleted();
         Reset();
      }

      [state.IDLE] {
         // Define behaviour for start event of activity
         on p_WarehouseActivity.StartTransferAndProcessWidget(product): {
            state = Activity_states_t.EXECUTE;

            // Insert code to start the first actions
            // For every action in initials
            r_ProcessingStation.Start();
            r_Warehouse.RequestToRetrieve(widgetTransferred);
         }
      }

      [state.EXECUTE] {
         on r_Warehouse.ReadyForPicking(): {
            warehouse_RequestToRetrieve_complete = true;
            if (processingStation_Ready_complete) {
               r_Robot.StartTransferFromWarehouseToProcessing();
            }
         }

         on r_ProcessingStation.ReadyForReceiving(): {
            processingStation_Ready_complete = true;
            if (warehouse_RequestToRetrieve_complete) {
               r_Robot.StartTransferFromWarehouseToProcessing();
            }
         }

         on r_Robot.PickedUpAtWarehouse(): {
            robot_MoveToWarehouseAndPick_complete = true;
            r_Warehouse.Picked(widgetTransferred);
            r_Robot.PlaceAtProcessing();
         }

         on r_Warehouse.ReadyForNext(): {
            warehouse_ReadyForNextAction_complete = true;
            if (robot_MoveToHomePosition_complete) {
               EndActivity();
            }
         }

         on r_Robot.DroppedAtProcessing(): {
            robot_MoveToOvenAndPlace_complete = true;
            r_Robot.Homing();
         }

         on r_Robot.MoveCompleted(): {
            robot_MoveToHomePosition_complete = true;
            if (warehouse_ReadyForNextAction_complete) {
               EndActivity();
            }
         }
      }
   }
}
```

2. *Color Sorter activity*

(a) **Interface**

```
import Definitions.dzn;

interface IColorSorterActivity {
   // Define start and end events
   in void StartTransferAndStoreWidget(WidgetColorParam product);
   out void TransferCompleted();

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      [state.IDLE] {
         // Define behaviour for start event of activity
         on StartTransferAndStoreWidget: {
            state = Activity_states_t.EXECUTE;
         }
      }

      [state.EXECUTE] {
         on StartTransferAndStoreWidget: illegal;
         on inevitable: {
            // Return end event of activity
            TransferCompleted;
            state = Activity_states_t.IDLE;
         }
      }
   }
}
```

(b) **Component**

```
import IColorSorterActivity.dzn;
import IRobot.dzn;
import IWarehouse.dzn;
import IColorSorter.dzn;

component ColorSorterActivity_Comp {
   provides IColorSorterActivity p_ColorSorterActivity;
   requires IRobot r_Robot;
   requires IWarehouse r_Warehouse;
   requires IColorSorter r_ColorSorter;

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      WidgetColorParam widgetTransferred;
      WidgetColorParam product;

      // For every action define a boolean variable
      bool colorSorter_SorterRequestToRetrieve_complete = false;
      bool robot_MoveToColorSorterAndPick_complete = false;
      bool robot_MoveToHomePosition_complete = false;
      bool robot_MoveToWarehouseAndPlace_complete = false;
      bool warehouse_ReadyForNextAction_complete = false;
      bool warehouse_RequestToStore_complete = false;

      void Reset() {
         state = Activity_states_t.IDLE;
         colorSorter_SorterRequestToRetrieve_complete = false;
```

```
         robot_MoveToColorSorterAndPick_complete = false;
         robot_MoveToHomePosition_complete = false;
         robot_MoveToWarehouseAndPlace_complete = false;
         warehouse_ReadyForNextAction_complete = false;
         warehouse_RequestToStore_complete = false;
      }

      void EndActivity() {
         p_ColorSorterActivity.TransferCompleted();
         Reset();
      }

      [state.IDLE] {
         // Define behaviour for start event of activity
         on p_ColorSorterActivity.StartTransferAndStoreWidget(product): {
            state = Activity_states_t.EXECUTE;

            // Insert code to start the first actions
            // For every action in initials
            r_Warehouse.RequestToStore(widgetTransferred);
            r_ColorSorter.SorterRequestToRetrieve(widgetTransferred);
         }
      }

      [state.EXECUTE] {
         on r_Warehouse.ReadyForReceiving(): {
            warehouse_RequestToStore_complete = true;
            if (colorSorter_SorterRequestToRetrieve_complete) {
               r_Robot.StartTransferFromColorSorterToWarehouse();
            }
         }

         on r_ColorSorter.SorterReadyForPicking(): {
            colorSorter_SorterRequestToRetrieve_complete = true;
            if (warehouse_RequestToStore_complete) {
               r_Robot.StartTransferFromColorSorterToWarehouse();
            }
         }

         on r_Robot.PickedAtColorSorter(): {
            robot_MoveToColorSorterAndPick_complete = true;
            r_Robot.PlaceAtWarehouse();
         }

         on r_Robot.DroppedAtWarehouse(): {
            robot_MoveToWarehouseAndPlace_complete = true;
            r_Warehouse.Placed(product);
            r_Robot.Homing();
         }

         on r_Warehouse.ReadyForNextAction(): {
            warehouse_ReadyForNextAction_complete = true;
            if (robot_MoveToHomePosition_complete) {
               EndActivity();
            }
         }

         on r_Robot.MoveCompleted(): {
            robot_MoveToHomePosition_complete = true;
            if (warehouse_ReadyForNextAction_complete) {
               EndActivity();
            }
         }
      }
   }
}
```

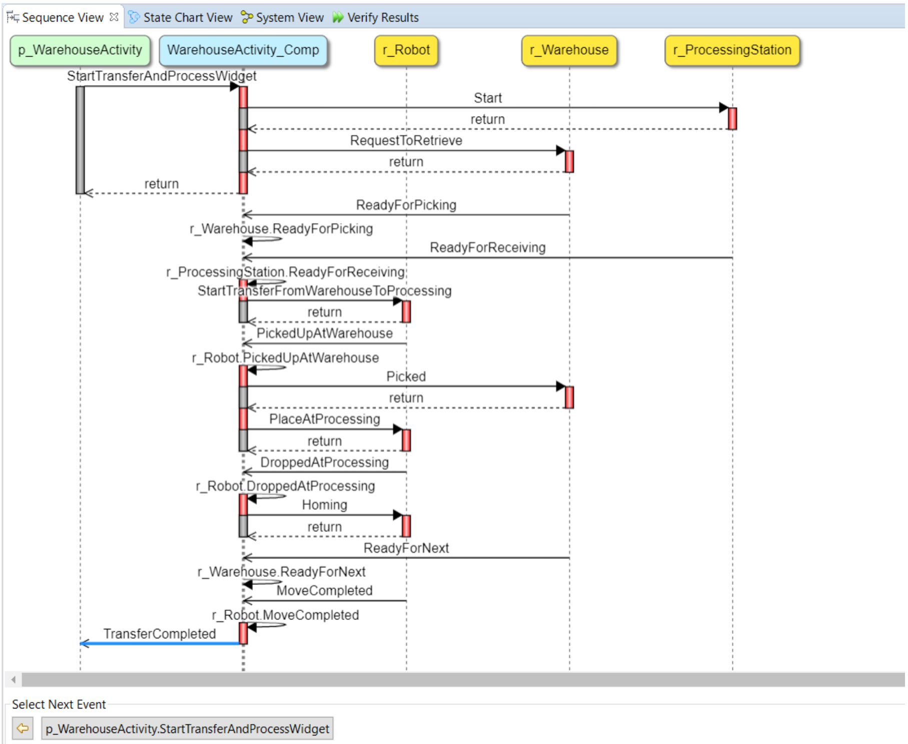


Figure 4.5: Sequence diagram for the *Warehouse Activity*

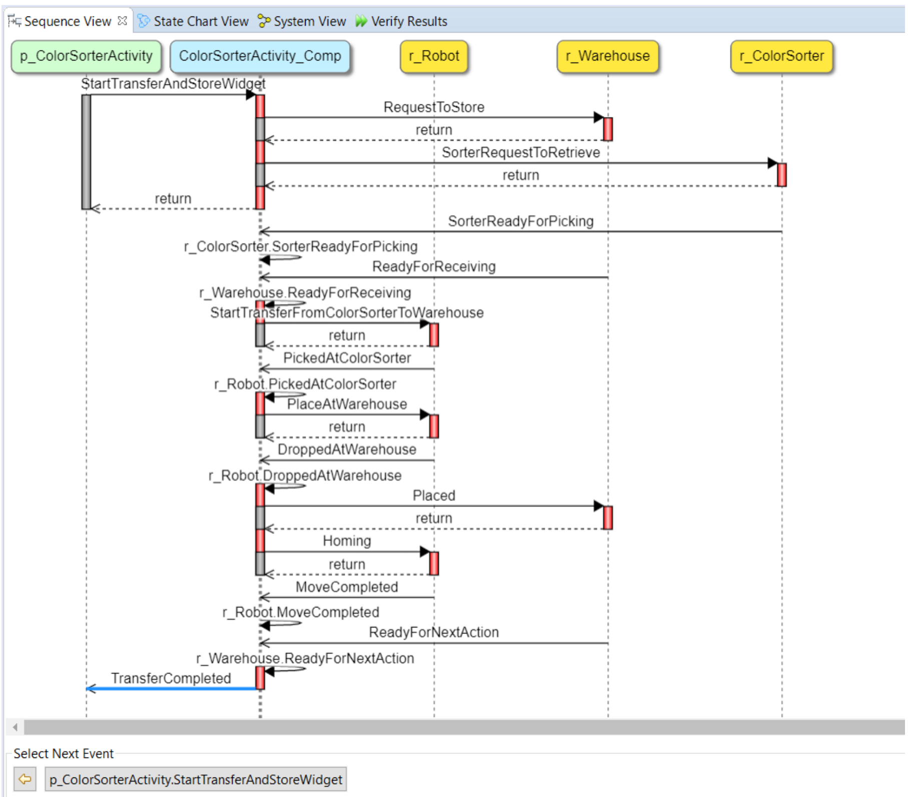


Figure 4.6: Sequence diagram for the *Color Sorter Activity*

Sequence diagrams of the generated Dezyne code

The behavior obtained from the generated code is shown as a sequence diagram for both activities in Figures 4.5 and 4.6.

4.5 Validating the translation method

To validate the transformation made for the activities of the Factory Four model, Gantt charts from the LSAT model (Figure 4.7 and 4.8) and the Gantt charts obtained from Dezyne (Figure 4.9 and 4.10) are compared. It can be seen that in both cases the behavior obtained is identical.

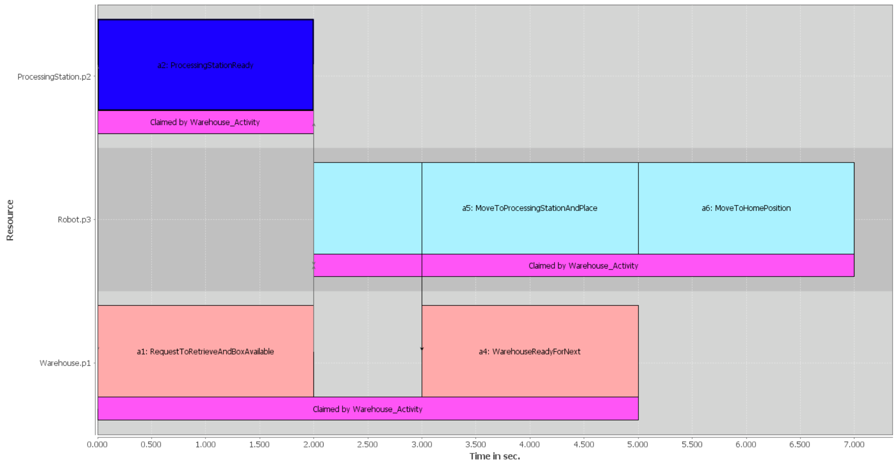


Figure 4.7: Gantt chart for the *Warehouse Activity* obtained from the LSAT model

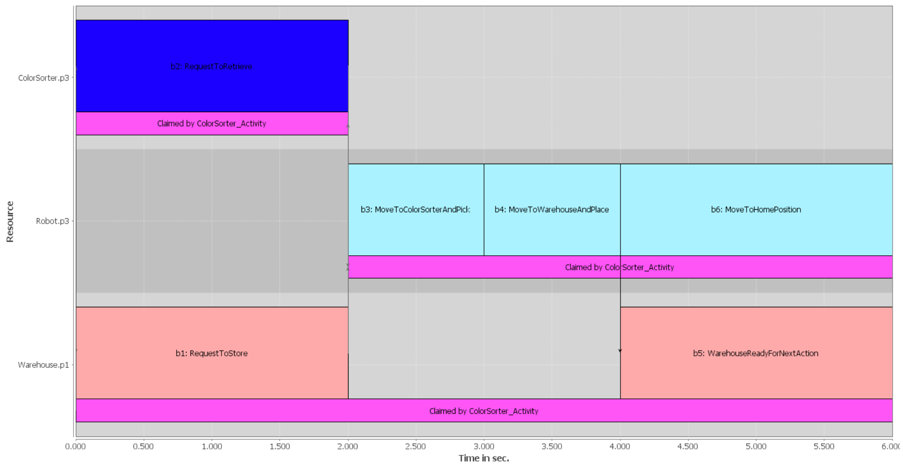


Figure 4.8: Gantt chart for the *Color Sorter Activity* obtained from the LSAT model

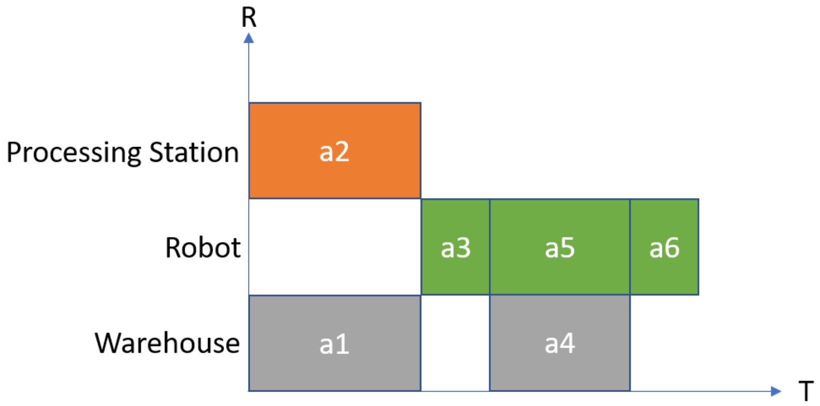


Figure 4.9: Gantt chart for the *Warehouse Activity* obtained from Dezyne

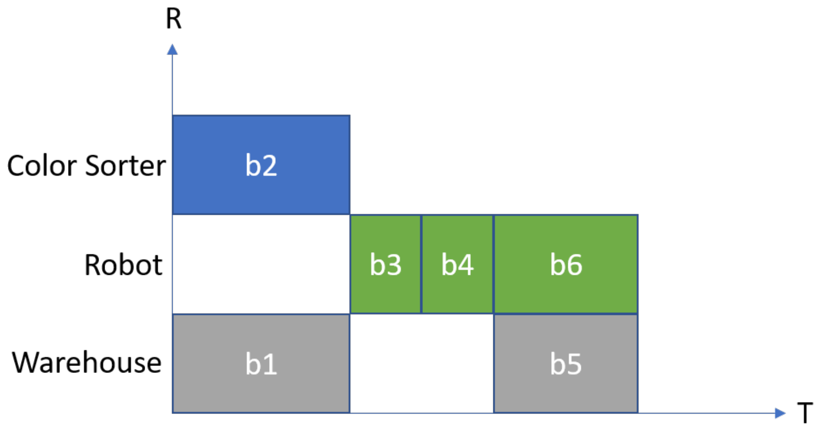
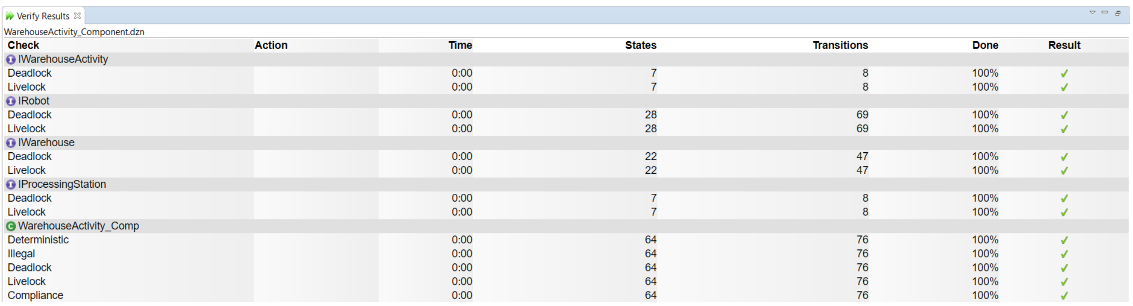


Figure 4.10: Gantt chart for the *Color Sorter Activity* obtained from Dezyne

4.6 Verifying the generated code

The verification results for the *Warehouse activity* and the *Color Sorter activity* in the Dezyne environment are shown in Figure 4.11 and Figure 4.12 respectively. It can be seen that both activities successfully pass the verification checks in the Dezyne environment.



Verify Results

WarehouseActivity_Component.dzn

Check	Action	Time	States	Transitions	Done	Result
IWarehouseActivity						
Deadlock		0:00	7	8	100%	✓
Livelock		0:00	7	8	100%	✓
IRobot						
Deadlock		0:00	28	69	100%	✓
Livelock		0:00	28	69	100%	✓
IWarehouse						
Deadlock		0:00	22	47	100%	✓
Livelock		0:00	22	47	100%	✓
IProcessingStation						
Deadlock		0:00	7	8	100%	✓
Livelock		0:00	7	8	100%	✓
WarehouseActivity_Comp						
Deterministic		0:00	64	76	100%	✓
Illegal		0:00	64	76	100%	✓
Deadlock		0:00	64	76	100%	✓
Livelock		0:00	64	76	100%	✓
Compliance		0:00	64	76	100%	✓

Figure 4.11: Verification result for the *Warehouse Activity*

Verify Results

ColorSorterActivity_Component.dzn

Check	Action	Time	States	Transitions	Done	Result
IColorSorterActivity						
Deadlock		0:00	7	8	100%	✓
Livelock		0:00	7	8	100%	✓
IRobot						
Deadlock		0:01	28	69	100%	✓
Livelock		0:01	28	69	100%	✓
IWarehouse						
Deadlock		0:00	22	47	100%	✓
Livelock		0:00	22	47	100%	✓
IColorSorter						
Deadlock		0:00	7	8	100%	✓
Livelock		0:00	7	8	100%	✓
ColorSorterActivity_Comp						
Deterministic		0:03	55	65	100%	✓
Illegal		0:03	55	65	100%	✓
Deadlock		0:03	55	65	100%	✓
Livelock		0:03	55	65	100%	✓
Compliance		0:00	55	65	100%	✓

Figure 4.12: Verification result for the *Color Sorter Activity*

Chapter 5

Handling Exceptions in an activity

Exceptions are external disturbances that may occur in an FMS while system operations are being executed. When such an exception occurs, the response of the system must be defined in order to handle it. In this work, four possible system responses are described based on three levels of criticality: low, medium and high. Since the activity framework does not have a mechanism to deal with exceptions, these cannot be specified in LSAT. The notion of an exception is introduced in the activity framework and is specified in the Activity DSL using various criticality levels. The corresponding semantics for these criticality levels is represented in the form of Gantt charts. The behavior of these criticality levels is mapped to Dezyne in the form of failure events, which are defined for an activity and the actions contained in that activity. These failure events are specified in the Transformation DSL. Further, the desired behavior in Dezyne is described for different criticality levels. Once the behavior is known, the rules for transformation are defined and an algorithm is developed for the automated translation from an activity model to the Dezyne code.

5.1 Exceptions in an activity

Exceptions are any unexpected failure occurrences that are not accounted for in a manufacturing system's normal operations. It is a condition that violates the original specifications of a system. These undefined and unanticipated conditions may occur dynamically and disrupt the normal flow of operations. When the system handles these exceptions improperly, it can lead to degradation of the system performance, may cause interruption in the production process by causing errors in the schedule plan, or even lead to system failures.

It is necessary to detect and diagnose the exceptions quickly, and recover the system by taking corrective measures to avoid fault propagation. Exception handling is the process of defining the system's response to the occurrences of exceptions. Many exceptions can be anticipated when a system is designed, and protection against these conditions can be incorporated into the specification of a system.

In an activity, an exception occurs when an action executed on a resource fails to complete execution. For instance, picking up a workpiece by the robot can fail due to its arm being stuck or the gripper being damaged. In this case, the exception handling mechanism must formulate what happens to the activity in which an exception occurred, and its effect on the other activities in the sequence. Also, the exception handling mechanism must prescribe different responses for different levels of exceptions that may occur. For instance, the response must be different for the case when a workpiece is stuck on the conveyor belt, which does not have a severe impact on the functioning of the system, as compared to when a part of the machinery fails, affecting the system's functionality.

Based on the possible levels of exceptions that may occur, caused due to the failure of an action, the response of the system envisaged is divided into three levels of criticality. These criticality levels are defined for the activity, rather than for each action, since the behavior of an activity is atomic in nature at the activity sequence level. The response of the system for different criticality levels is as follows:

1. **Low-level criticality:** A low-level criticality is defined for exceptions which have minimal impact on the functioning of the system. For instance, if a widget is deformed, it can still undergo various

operations scheduled and can be discarded at the output. These exceptions do not need immediate handling and the activity can continue execution under the determinate scenario. The only difference is that in case of failure of an action, the error in the system is logged as a warning which is handled during the next maintenance schedule. These kind of exceptions do not have any impact on the scheduling and execution of other activity sequences. The system operations continue as normal and other pipelined activities are executed according to the planned schedule.

2. **Medium-level criticality:** A medium-level criticality is defined for exceptions that have some degree of impact on the system operations. For instance, a part of the machinery stuck in movement, such as a robotic arm, unable to move from one workstation to another. In such a situation, an activity in which an exception occurs completes execution, and sends a failure message to the controller which is logged. The activity sequences already pipelined in the schedule for execution completes, but further scheduling of the activities is cancelled. This is because the other activity sequences may have some form of dependency on the activity in which an exception occurred.

3. **High-level criticality:** A high-level criticality is defined for exceptions which have severe impact on the functioning of the system. For instance, if a part of the machinery breaks down, the manufacturing operations cannot continue. The handling of such exceptions can not be delayed and must be addressed immediately. There are two ways in which a high-level criticality of an exception can be handled.

   (a) *Complete shut down of the system:* If an action fails, the activity stops in the middle of its execution, and quits immediately. The complete system shuts down, the power is cut-off, and the other activity sequences which are scheduled for execution, or are executing simultaneously, also terminate. After fixing the error, the FMS can be restarted and initialized again.

   (b) *Go to error state:* If an exception occurs, the whole system's execution is paused, and the activity execution moves to an error state. At the error state, the system waits for an input from the operator who has two options as given below. In case no input is received from the operator, after a fixed time period, the system shuts itself down.

      i. *Resume execution:* The operator rectifies the error immediately, and the activity continues execution. For instance, a workpiece stuck on a conveyor belt can be fixed manually by the operator. In this case, the control goes back to the last known state before an exception occurred, and activity resumes execution. The other activity sequences also continue as scheduled.

      ii. *Restart execution:* It might be that the operator cannot resolve the issue immediately, for instance, due to the breakdown of the machinery. In this case, the response is to shut down the whole system, and restart again when the issue is resolved.

A summary of the system's response for various criticality levels is given in table 5.1. In this project, the cases of low-level criticality and high-level criticality (complete shut down of the system) is considered. Other cases can be included in the future work.

The activity framework currently has no mechanism to handle exceptions. To generate the Dezyne code from the activity models handling exceptions, it is assumed that the activity framework must support certain features, which can be included in the framework in the future. The assumptions are as follows:

1. Low-level criticality: The activity framework must support some sort of feedback mechanism in order to know that an action, and hence the activity has failed to execute successfully.

2. High-level criticality (complete shut down of the system): In this criticality, the activity quits immediately when an action fails to execute. The activity framework must be able to deal with the action-level behavior to stop execution of an activity mid-way. Also, it must have a mechanism to support the communication between various resources in order to convey that the activity has terminated and the resources involved in the activity should stop execution.

An exception is represented in an activity as shown in Figure 5.1. Any action within an activity can fail. For activity Act_1, it is assumed that action a fails execution and an exception occurs. This is denoted

Criticality	Effect on the activity in which an exception occurs	Effect on the other activity sequences	Example case
Low-level	Continue execution with warning and log errors	Continue execution	Workpiece is damaged
Medium-level	Continue execution with warning and log errors	Continue execution of the activities that are already scheduled and stop further scheduling of the activities	Robotic arm is stuck
High-level (a) Complete shut down of the system	Halt execution immediately	Halt execution immediately	Machinery/part of machinery broke down
(b) Go to error state			
i) Resume execution	Go to the last known state before an exception occurred and continue execution from that state	Continue execution	Widget is stuck on conveyor belt
ii) Restart execution	Halt execution immediately	Halt execution immediately	Machinery/part of machinery broke down

Table 5.1: Different levels of criticality for handling exceptions

by notation **E**. Similarly, in activity Act_2 and activity Act_3, it is assumed that an exception occurs because action c and action e fail to execute respectively.

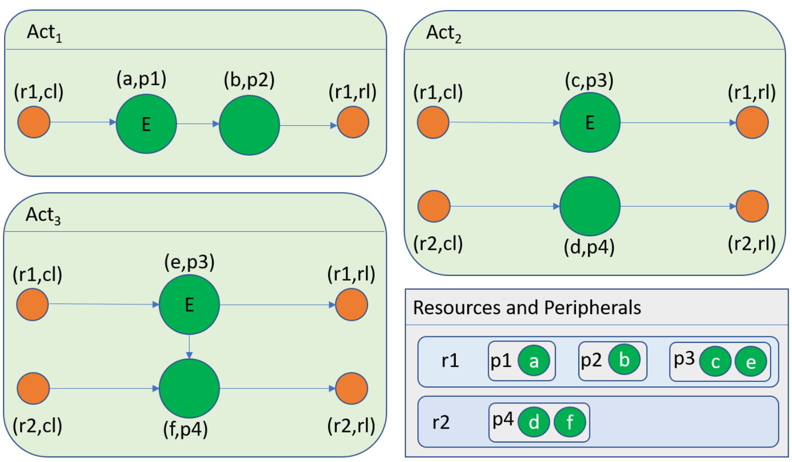


Figure 5.1: Activities Act_1, Act_2 and Act_3 with exceptions

5.2 Specification of exceptions in an activity

LSAT provides specifications only for determinate scenarios. The activity framework has no mechanism to deal with exceptions, and there is no way to specify them in LSAT.

The exceptions are incorporated in the Activity DSL using various criticality levels. These criticality levels describe the criticality of an activity. These levels are defined using the keyword **Criticality level** followed by an integer which can take values **0**, **1** or **2**. The different cases criticality levels represent is as follows:

1. **Criticality level 0:** This case represents activities modeled as determinate scenarios. It offers no exception handling mechanism.
2. **Criticality level 1:** This case represents low-level criticality to handle exceptions.
3. **Criticality level 2:** This case represents high-level criticality, more specifically, the complete shut down of the system, to handle exceptions.

The activity model for activities Act_1, Act_2 and Act_3 is shown below.

1. **Criticality level 1**

```

Resource r1
   Action type a
   Action type b
   Action type c
   Action type e
Resource r2
   Action type d
   Action type f

Activity Act1
   Criticality level 1
   A1 : r1.a
   A2 : r1.b
   Dependencies
   A1 -> A2

Activity Act2
   Criticality level 1
   A3 : r1.c
   A4 : r2.d
Dependencies

Activity Act3
   Criticality level 1
   A5 : r1.e
   A6 : r2.f
   Dependencies
   A5 -> A6
```

2. **Criticality level 2**

```

Resource r1
   Action type a
   Action type b
   Action type c
   Action type e
Resource r2
   Action type d
   Action type f

Activity Act1
   Criticality level 2
```

```
  A1 : r1.a
  A2 : r1.b
  Dependencies
  A1 -> A2

Activity Act2
  Criticality level 2
  A3 : r1.c
  A4 : r2.d
Dependencies

Activity Act3
  Criticality level 2
  A5 : r1.e
  A6 : r2.f
  Dependencies
  A5 -> A6
```

5.3 Semantics of handling exceptions in an activity

The effect of an exception on the execution of the activity is represented as Gantt charts. For criticality level 1 it is shown in Figure 5.2. It is assumed that an exception occurs in action *a* for Act_1, in action *c* for Act_2 and in action *e* for Act_3. This exception is represented as a dashed line in the actions failing to execute. Since criticality level 1 is the low-level criticality case, the system responds by logging the error and continuing execution of the activity. Hence, action *b*, action *d* and action *f* continue to execute in Act_1, Act_2 and Act_3 respectively. The activities complete execution and the resources are released after executing their corresponding actions.

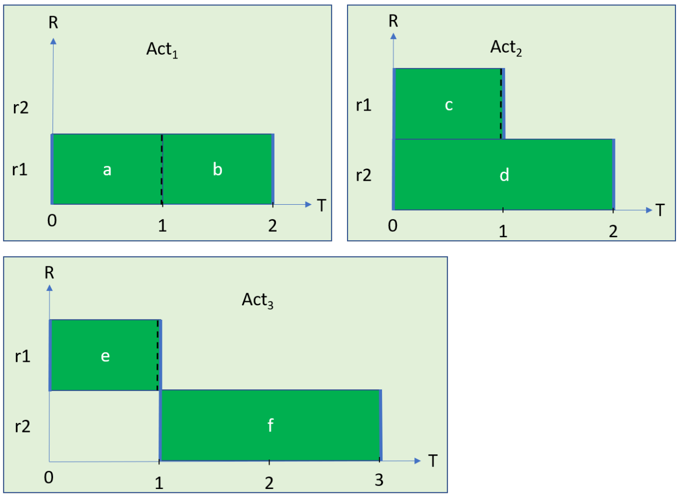


Figure 5.2: Gantt charts for activities Act_1, Act_2 and Act_3 for criticality level 1

Gantt charts for activities having exceptions of criticality level 2 is shown in Figure 5.3. This case corresponds to high-level criticality in which the system stops execution immediately and shuts down completely when an exception occurs. In Act_1, when action *a* fails execution, the activity terminates instantly and resource *r1* is released at time unit 1 without executing action *b*. Similarly, in Act_2, when action *c* fails execution on resource *r1*, action *d* running concurrently on resource *r2* is stopped mid-way. Both resources are released at time unit 1 and the activity terminates. Similarly, in Act_3, when action *e* fails execution, resource *r1* is released. Resource *r2* is not claimed to perform action *f* and the activity quits execution at time unit 1.

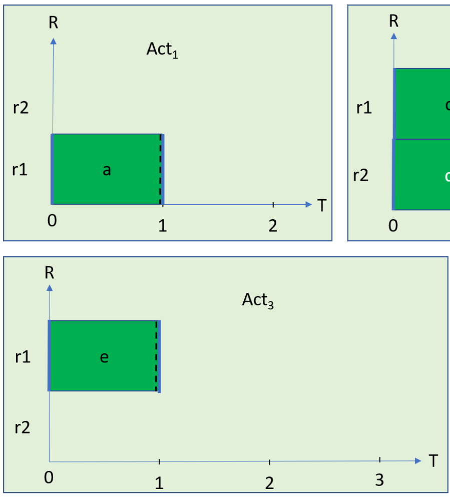


Figure 5.3: Gantt chart for activities Act_1, Act_2 and Act_3 for criticality level 2

5.4 Modeling exceptions in Dezyne

The exceptions can be modeled in Dezyne as failure events to specify that an action, and thus, the activity failed to execute successfully. If an action starts execution, it can either complete execution with a complete end event returned, or it can fail to execute. In case it fails to execute, an exception occurs and a failure end event is returned. Similarly, an activity can either complete execution with a complete end event, or end with a failure end event if an exception occurs in one of its actions. This is shown in Figure 5.4 where a failure end event is added at the action level and at the activity level.

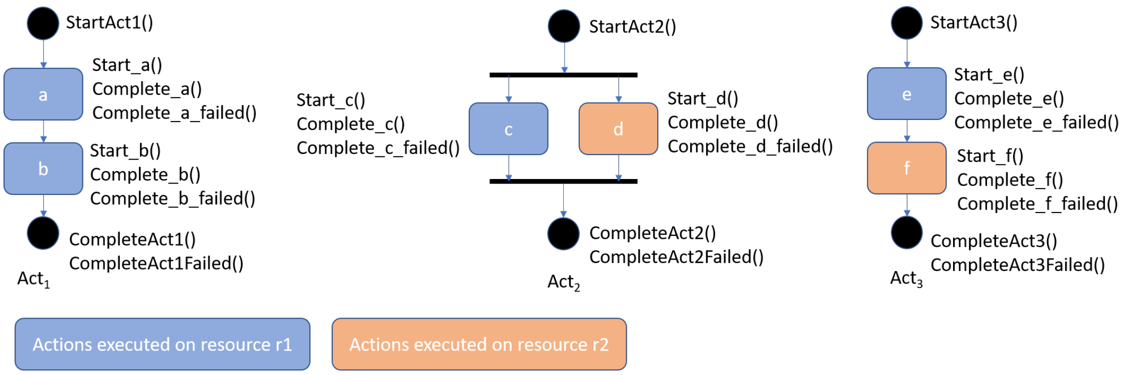


Figure 5.4: Activities Act_1, Act_2 and Act_3 with failure events

5.4.1 Behavior in Dezyne for Criticality level 1

Activity Act_1 starts with the StartAct1() event. This triggers action a to start execution and Start_a() event is called. When action a completes execution Complete_a() event is returned, which triggers action b to start execution by triggering Start_b() event. In case action a fails to execute, Complete_a_failed() event is returned, which also triggers action b to start execution by triggering Start_b() event. When action b finishes execution Complete_b() event is returned. In case action b fails to execute, Complete_b_failed() event is returned. The activity completes execution successfully only when Complete_a() event and Complete_b() event is returned, and the activity ends with CompleteAct1() event. In case one of the actions fails execution, CompleteAct1Failed() event is returned indicating the end of the activity with failure.

Similarly, activity Act_2 starts with the StartAct2() event. The start of the activity triggers execution of Start_c() and Start_d(). When action *c* completes, Complete_c() event is returned and variable c_complete is set to *true*. In case action *c* fails execution, Complete_c_failed() event is returned and variable c_complete is set to *true*. Similarly, when action *d* completes, Complete_d() event is returned and variable d_complete is set to *true*. In case action *d* fails execution, Complete_d_failed() event is returned and variable d_complete is set to *true*. When both actions corresponding Boolean completion variables are set to *true*, then the activity completes execution with CompleteAct2() event only if Complete_c() event and Complete_d() event is returned, else CompleteAct2Failed() event is returned indicating the end of the activity with failure.

Activity Act_3 starts with StartAct3() event. This triggers action *e* to start execution and Start_e() event is called. When action *e* completes execution Complete_e() event is returned, which triggers action *f* to start execution by triggering Start_f() event. In case action *e* fails to execute, Complete_e_failed() event is returned, which also triggers action *f* to start execution by triggering Start_f() event. When action *f* finishes execution Complete_f() event is returned. In case action *f* fails to execute, Complete_f_failed() event is returned. The activity completes execution successfully only when Complete_e() event and Complete_f() event is returned, and the activity ends with CompleteAct3() event. In case one of the actions fails execution, CompleteAct3Failed() event is returned indicating the end of the activity with failure.

5.4.2 Behavior in Dezyne for Criticality level 2

Activity Act_1 starts with StartAct1() event. This triggers action *a* to start execution and Start_a() event is called. If action *a* fails to execute, Complete_a_failed() event is returned. This triggers the end of the activity and CompleteAct1Failed() event is returned. Simultaneously, a reset event is sent to all the resources used by the activity to communicate that the activity has terminated. In case action *a* completes execution, Complete_a() event is returned, which triggers action *b* to start execution by triggering Start_b() event. If action *b* fails to execute, Complete_b_failed() event is returned. This triggers the end of the activity and CompleteAct1Failed() event is returned. Simultaneously, a reset event is sent to all the resources used by the activity to communicate that the activity has terminated. In case action *b* finishes execution, Complete_b() event is returned, and this triggers CompleteAct1() event indicating the end of the activity.

Activity Act_3 starts with StartAct3() event. This triggers action *e* to start execution and Start_e() event is called. If action *c* fails to execute, Complete_c_failed() event is returned. This triggers the end of the activity and CompleteAct3Failed() event is returned. Simultaneously, a reset event is sent to all the resources used by the activity to communicate that the activity has terminated. In case action *e* completes execution, Complete_e() event is returned, which triggers action *f* to start execution by triggering Start_f() event. If action *f* fails to execute, Complete_f_failed() event is returned. This triggers the end of the activity and CompleteAct3Failed() event is returned. Simultaneously, a reset event is sent to all the resources used by the activity to communicate that the activity has terminated. In case action *f* finishes execution, Complete_f() event is returned, and this triggers CompleteAct3() event indicating the end of the activity.

Similarly, activity Act_2 starts with the StartAct2() event. The start of the activity triggers execution of Start_c() event and Start_d() event. When action *c* completes, Complete_c() event is returned and variable c_complete is set to *true*. In case action *c* fails execution, Complete_c_failed() event is returned. This triggers the end of the activity and CompleteAct2Failed() event is returned. Simultaneously, a reset event is sent to all the resources used by the activity to communicate that the activity has terminated. Similarly, when action *d* completes, Complete_d() event is returned and variable d_complete is set to *true*. In case action *d* fails execution, Complete_d_failed() event is returned. This triggers the end of the activity and CompleteAct2Failed() event is returned. Simultaneously, a reset event is sent to all the resources used by the activity to communicate that the activity has terminated. When both actions complete with Complete_c() event and Complete_d() event, and their corresponding Boolean completion variables are set to *true*, then the activity completes execution with event CompleteAct2().

5.5 Rules for translation

The mapping of an activity to the Dezyne code in order to handle exceptions must follow the following set of rules:

1. Activity events ∈ {`Start_A, End_A, Failed_End_A`}.
   For each activity, to define its start and end behavior in the `activity interface`, a tuple of three events is defined: a start event `Start_A` defined as an *in* event, a complete end event `End_A` defined as an *out* event and a failed end event `Failed_End_A` defined as an *out* event.

2. Activity interface states ∈ {`idle, execute`}.
   Each `activity interface` has two states: `idle` and `execute`. The initial state is the `idle` state. When the activity starts execution with the invocation of start event `Start_A` the state changes to `execute`. Once the execution of the activity completes, the end event `End_A` is called and the state reverts to the `idle` state. If the execution of an activity fails, the failed end event `Failed_End_A` is called and the state reverts to the `idle` state.

3. Activity failed status variable ∈ {`true, false`}.
   For the activity a Boolean variable `activity_failed` is defined in `activity component` to represent failure of an action: if an action fails execution its value corresponds to *true* otherwise its *false*.

4. Action events ∈ {`start_action_event, end_action_event, failed_end_action_event`}.
   For each action, to define its start and end behavior in the `resource interface`, a tuple of three events is defined: a start event `start_action_event` defined as an *in* event, a complete end event `end_action_event` defined as an *out* event and a failed end event `failed_end_action_event` defined as an *out* event.

5. Action status variable ∈ {`true, false`}.
   For each action, a Boolean variable, `action_complete` is defined in the `activity component` to represent its execution status: if an action completes execution its value corresponds to *true* otherwise its *false*.

6. Dependencies of an action ∈ {`Initial actions, dep, post`}.
   `Initial actions` are the actions triggered by the start of the activity.
   `Dep(a)` is a set of actions that need to complete before action `a` can start.
   `Post(a)` is a set of actions triggered by the completion of action `a`.

7. Activity component states ∈ {`idle, execute`}.
   Each `activity component` has two states: `idle` and `execute`.
   (a) The initial state is the `idle` state. In the `idle` state, when the start event of the activity, `Start_A` is called, it triggers the execution of `start_action_event` of the corresponding `initial actions` and the state changes to `execute`. The behavior in `execute` varies for each criticality level.
   (b) i. *Criticality Level 1:* In the `execute` state, the sequence of actions corresponding to different resources, following the completion of `initial actions`, is executed. This sequence of actions is defined using a set of dependencies. If an action completes execution, the `end_action_event` is triggered and its corresponding Boolean completion variable is set to *true*. If an action fails execution, the `failed_end_action_event` is triggered, its corresponding Boolean completion variable and the activity failed status variable is set to *true*. Then, for both cases, the action's dependent actions (`dep`) are checked for completion, and once completed, all post actions (`post`) are triggered for execution using their corresponding `start_action_event`.
      Once all actions complete execution `EndActivity` function is invoked. This function checks the value of activity failed status variable. If its value is *true*, then the `Activity Failed` function is called which contains the failed end event of the activity, `Failed_End_A` and a `Reset` function. The `Reset` function sets the action completion variables `action_complete` and the activity failed status variable `activity_failed` back to *false*, and reverts the activity component state to `idle` state. If the activity failed status variable value is *false*, then the `End_A` event is returned and a `Reset` function is called.

ii. *Criticality Level 2:* In the `execute` state, the sequence of actions corresponding to different resources, following the completion of `initial actions`, is executed. This sequence of actions is defined using a set of dependencies. If an action completes execution, the `end_action_event` is triggered and its corresponding Boolean completion variable is set to *true*. Then, its dependent actions (`dep`) are checked for completion, and once completed, all post actions (`post`) are triggered for execution using their corresponding `start_action_event`. If an action fails execution, the `failed_end_action_event` is triggered, and the `ActivityFailed` function is called. This function calls the `Failed_End_A` event, the `Reset` function and the `ResetResources` function. The `Reset` function sets the action completion variables, `action_complete` back to *false*, and reverts the activity component state to `idle` state. The `ResetResources` function resets the resources and changes their state to `idle`.
If all actions complete execution, the `EndActivity` function is invoked. This function contains the end event of the activity, `End_A` and a `Reset` function.

5.6 Algorithm for translation

1. **Criticality level 1**

   (a) ***Interface***
   Inputs: name of the activity <Activity_Name>, and three events defining the start and end of the activity: <Start_A>,<End_A>, <Failed_End_A>

```
interface I<Activity_Name> {

   //Define start and end events
   in void <Start_A>();
   out void <End_A>();
   out void <Failed_End_A>();

   behaviour {
      //Define two states
      enum Activity_states_t {IDLE, EXECUTE};
      //Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      [state.IDLE] {
         //Define behaviour for start event of activity
         on <Start_A>: {
            state = Activity_states_t.EXECUTE;
         }
      }

      [state.EXECUTE] {
         on <Start_A>: illegal;
         on inevitable: {
            //Return end event of activity
            <End_A>;
            state = Activity_states_t.IDLE;
         }

         on inevitable: {
            //Return end event of activity
            <Failed_End_A>;
            state = Activity_states_t.IDLE;
         }
      }
   }
}
```

(b) ***Component***
Resource `<resource_name>` provides interface `<interface_name>` with the action `<action_name>` which starts with `<start_action_event>` and ends with `<end_action_event>` or `<failed_end_action_event>`

```
import <resource_interface_name>.dzn;

component <Activity_Name>_Comp {

   provides I<Activity_Name> p_<Activity_Name>;
   requires <resource_interface_name> r_<resource_name>;

   behaviour {

      // Define two states
      enum Activity_states_t {IDLE, EXECUTE};
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      bool activity_failed = false;

      // For every action <action_name> define a boolean variable
      bool <resource_name>_<action_name>_complete = false;

      void Reset():
      {
         state = Activity_states_t.IDLE;
         <resource_name>_<action_name>_complete = false;
         activity_failed = false;
      }

      void EndActivity() {
         if (activity_failed) {
            ActivityFailed();
         }
         else {
            p_<Activity_Name>.<End_A>;
            Reset();
         }
      }

      void ActivityFailed():
      {
         p_<Activity_Name>.<Failed_End_A>();
         Reset();
      }

      [state.IDLE] {

         // Define behaviour for start event of activity
         on p_<Activity_Name>.<Start_A>(): {
            state = Activity_states_t.EXECUTE;

            // Insert code to start the first actions
            // For every action in initials
            r_<resource_name>.<start_action_event>();

         }
      }

      [state.EXECUTE] {

         // Generate code following the activity dependencies
         // Given a tuple <a, dep, post>
         // a = an action of resource named <resource_name>
         // dep = all actions that need to be completed before a can start
         // post = all actions that need to be started when a completes
```

```

         on r_<resource_name>.<end_action_event>(): {

            // For action a, change boolean to indicate a finishes
                execution
            <resource_name>_<action_name>_complete = true;

            if (<all actions in dep have their boolean to true>) {

                  // If post is not empty, start all post actions
                  // For all actions in post
                  r_<resource_name>.<start_action_event>();

                  // If post is empty, then complete the activity
                  // If post is empty, then complete the activity
                  EndActivity();
               }
            }

            on r_<resource_name>.<failed_end_action_event>(): {

            // For action a, change boolean to indicate a finishes
                execution
            <resource_name>_<action_name>_complete = true;
             activity_failed = true;

            if (<all actions in dep have their boolean to true>) {

                  // If post is not empty, start all post actions
                  // For all actions in post
                  r_<resource_name>.<start_action_event>();


                  // If post is empty, then complete the activity
                  EndActivity();
            }
         }
      }
   }
}
```

2. **Criticality level 2**

   (a) ***Interface***
   Inputs: name of the activity `<Activity_Name>`, and three events defining the start and end of the activity: `<Start_A>,<End_A>,<Failed_End_A>`

```
interface I<Activity_Name> {

   //Define start and end events
   in void <Start_A>();
   out void <End_A>();
   out void <Failed_End_A>();

   behaviour {
      //Define two states
      enum Activity_states_t {IDLE, EXECUTE};
      //Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      [state.IDLE] {
         //Define behaviour for start event of activity
         on <Start_A>: {
            state = Activity_states_t.EXECUTE;
         }
      }
```

```
      [state.EXECUTE] {
         on <Start_A>: illegal;
         on inevitable: {
            //Return end event of activity
            <End_A>;
            state = Activity_states_t.IDLE;
         }

         on inevitable: {
            //Return end event of activity
            <Failed_End_A>;
            state = Activity_states_t.IDLE;
         }
      }
   }
}
```

(b) ***Component***

Resource `<resource_name>` provides interface `<interface_name>` with the action `<action_name>` which starts with `<start_action_event>` and ends with `<end_action_event>` or `<failed_end_action_event>`.

```
import <resource_interface_name>.dzn;

component <Activity_Name>_Comp {

   provides I<Activity_Name> p_<Activity_Name>;
   requires <resource_interface_name> r_<resource_name>;

   behaviour {

      // Define two states
      enum Activity_states_t {IDLE, EXECUTE};
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;
      // For every action <action_name> define a boolean variable
      bool <resource_name>_<action_name>_complete = false;

      void Reset()
      {
         state = Activity_states_t.IDLE;
         <resource_name>_<action_name>_complete = false;
      }

      void EndActivity() {
         p_<Activity_Name>.<End_A>;
         Reset();
      }

      void ResetResources() {
         r_<resource_name>.Reset();
      }

      void ActivityFailed() {
         p_<Activity_Name>.<Failed_End_A>();
         Reset();
         ResetResources();
      }

      [state.IDLE] {

         // Define behaviour for start event of activity
         on p_<Activity_Name>.<Start_A>(): {
            state = Activity_states_t.EXECUTE;

            // Insert code to start the first actions
```

```
                // For every action in initials
                r_<resource_name>.<start_action_event>();
            }
        }

        [state.EXECUTE] {

            // Generate code following the activity dependencies
            // Given a tuple <a, dep, post>
            // a = an action of resource named <resource_name>
            // dep = all actions that need to be completed before a can start
            // post = all actions that need to be started when a completes

            on r_<resource_name>.<end_action_event>(): {

                // For action a, change boolean to indicate a finishes
                    execution
                <resource_name>_<action_name>_complete = true;

                if (<all actions in dep have their boolean to true>) {

                        // If post is not empty, start all post actions
                        // For all actions in post
                        //If action completes successfully
                        r_<resource_name>.<start_action_event>();

                        // If post is empty, then complete the activity
                        EndActivity();
                    }
                }

                on r_<resource_name>.<failed_end_action_event>(): {
                    //If action fails execution
                    ActivityFailed();
                }
        }
    }
}
```

5.7 Adding Translation Model to the Transformation DSL

In order to handle exceptions, for an activity and each action within that activity, a **failed end event** is specified in the translation model. The translation model is common for criticality level 1 and criticality level 2, and is shown below.

```
Activity Act1
StartEvent: StartAct1()
EndEvent: CompleteAct1()
FailedEndEvent: CompleteAct1Failed()

Activity Act2
StartEvent:StartAct2()
EndEvent: CompleteAct2()
FailedEndEvent: CompleteAct2Failed()

Activity Act3
StartEvent: StartAct3()
EndEvent: CompleteAct3()
FailedEndEvent: CompleteAct3Failed()

Resource r1
Interface: Ir1

Action a
StartEvent: Start_a()
```

```
EndEvent: Complete_a()
FailedEndEvent: Complete_a_failed()

Action b
StartEvent: Start_b()
EndEvent: Complete_b()
FailedEndEvent: Complete_b_failed()

Action c
StartEvent: Start_c()
EndEvent: Complete_c()
FailedEndEvent: Complete_c_failed()

Action e
StartEvent: Start_e()
EndEvent: Complete_e()
FailedEndEvent: Complete_e_failed()

Resource r2
Interface: Ir2

Action d
StartEvent: Start_d()
EndEvent: Complete_d()
FailedEndEvent: Complete_d_failed()

Action f
StartEvent: Start_f()
EndEvent: Complete_f()
FailedEndEvent: Complete_f_failed()
```

5.8 Results

After specifying the activity model and the translation model for the example cases, Dezyne code is generated. The interface code for the resources of Act_1, Act_2 and Act_3 for criticality levels 1 and 2 is given in section A.1.2 and section A.1.3 respectively for reference.

1. **Criticality level 1**

   (a) For Act_1 the generated Dezyne code is shown below.

      i. ***Interface***

```
interface IAct1 {
   // Define start and end events
   in void StartAct1();
   out void CompleteAct1();
   out void CompleteAct1Failed();

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      [state.IDLE] {
         // Define behaviour for start event of activity
         on StartAct1: {
            state = Activity_states_t.EXECUTE;
         }
      }

      [state.EXECUTE] {
         on StartAct1: illegal;
         on inevitable: {
```

```
            // Return end event of activity
            CompleteAct1;
            state = Activity_states_t.IDLE;
         }

         on inevitable: {
            // Return failed end event of activity
            CompleteAct1Failed;
            state = Activity_states_t.IDLE;
         }
      }
   }
}
```

ii. ***Component***

```
import IAct1.dzn;
import Ir1.dzn;

component Act1_Comp {
   provides IAct1 p_Act1;
   requires Ir1 r_r1;

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      bool activity_failed = false;

      // For every action define a boolean variable
      bool r1_a_complete = false;
      bool r1_b_complete = false;

      void Reset() {
         state = Activity_states_t.IDLE;
         r1_a_complete = false;
         r1_b_complete = false;
         activity_failed = false;
      }

      void EndActivity() {
         if (activity_failed) {
            ActivityFailed();
         }
         else {
            p_Act1.CompleteAct1();
            Reset();
         }
      }

      void ActivityFailed() {
         p_Act1.CompleteAct1Failed();
         Reset();
      }

      [state.IDLE] {
         // Define behaviour for start event of activity
         on p_Act1.StartAct1(): {
            state = Activity_states_t.EXECUTE;

            // Insert code to start the first actions
            // For every action in initials
            r_r1.Start_a();
         }
      }

```

```
      [state.EXECUTE] {
         on r_r1.Complete_a(): {
            r1_a_complete = true;
            r_r1.Start_b();
         }

         on r_r1.Complete_a_failed(): {
            r1_a_complete = true;
            activity_failed = true;
            r_r1.Start_b();
         }

         on r_r1.Complete_b(): {
            r1_b_complete = true;
            EndActivity();
         }

         on r_r1.Complete_b_failed(): {
            r1_b_complete = true;
            activity_failed = true;
            EndActivity();
         }
      }
   }
}
```

(b) For Act_2 the generated Dezyne code is shown below.

i. ***Interface***

```
interface IAct2 {
   // Define start and end events
   in void StartAct2();
   out void CompleteAct2();
   out void CompleteAct2Failed();

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      [state.IDLE] {
         // Define behaviour for start event of activity
         on StartAct2: {
            state = Activity_states_t.EXECUTE;
         }
      }

      [state.EXECUTE] {
         on StartAct2: illegal;
         on inevitable: {
            // Return end event of activity
            CompleteAct2;
            state = Activity_states_t.IDLE;
         }

         on inevitable: {
            // Return failed end event of activity
            CompleteAct2Failed;
            state = Activity_states_t.IDLE;
         }
      }
   }
}
```

ii. ***Component***

```
import IAct2.dzn;
import Ir1.dzn;
import Ir2.dzn;

component Act2_Comp {
   provides IAct2 p_Act2;
   requires Ir1 r_r1;
   requires Ir2 r_r2;

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      bool activity_failed = false;

      // For every action define a boolean variable
      bool r1_c_complete = false;
      bool r2_d_complete = false;

      void Reset() {
         state = Activity_states_t.IDLE;
         r1_c_complete = false;
         r2_d_complete = false;
         activity_failed = false;
      }

      void EndActivity() {
         if (activity_failed) {
            ActivityFailed();
         }
         else {
            p_Act2.CompleteAct2();
            Reset();
         }
      }

      void ActivityFailed() {
         p_Act2.CompleteAct2Failed();
         Reset();
      }

      [state.IDLE] {
         // Define behaviour for start event of activity
         on p_Act2.StartAct2(): {
            state = Activity_states_t.EXECUTE;

            // Insert code to start the first actions
            // For every action in initials
            r_r1.Start_c();
            r_r2.Start_d();
         }
      }

      [state.EXECUTE] {
         on r_r1.Complete_c(): {
            r1_c_complete = true;
            if (r2_d_complete) {
               EndActivity();
            }
         }

         on r_r1.Complete_c_failed(): {
            r1_c_complete = true;
            activity_failed = true;
```

```
            if (r2_d_complete) {
               EndActivity();
            }
         }

         on r_r2.Complete_d(): {
            r2_d_complete = true;
            if (r1_c_complete) {
               EndActivity();
            }
         }

         on r_r2.Complete_d_failed(): {
            r2_d_complete = true;
            activity_failed = true;
            if (r1_c_complete) {
               EndActivity();
            }
         }
      }
   }
}
```

(c) For Act_3 the generated Dezyne code is shown below.

i. ***Interface***

```
interface IAct3 {
   // Define start and end events
   in void StartAct3();
   out void CompleteAct3();
   out void CompleteAct3Failed();

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      [state.IDLE] {
         // Define behaviour for start event of activity
         on StartAct3: {
            state = Activity_states_t.EXECUTE;
         }
      }

      [state.EXECUTE] {
         on StartAct3: illegal;
         on inevitable: {
            // Return end event of activity
            CompleteAct3;
            state = Activity_states_t.IDLE;
         }

         on inevitable: {
            // Return failed end event of activity
            CompleteAct3Failed;
            state = Activity_states_t.IDLE;
         }
      }
   }
}
```

ii. ***Component***

```
import IAct3.dzn;
import Ir1.dzn;
import Ir2.dzn;

```

```
component Act3_Comp {
   provides IAct3 p_Act3;
   requires Ir1 r_r1;
   requires Ir2 r_r2;

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      bool activity_failed = false;

      // For every action define a boolean variable
      bool r1_e_complete = false;
      bool r2_f_complete = false;

      void Reset() {
         state = Activity_states_t.IDLE;
         r1_e_complete = false;
         r2_f_complete = false;
         activity_failed = false;
      }

      void EndActivity() {
         if (activity_failed) {
            ActivityFailed();
         }
         else {
            p_Act3.CompleteAct3();
            Reset();
         }
      }

      void ActivityFailed() {
         p_Act3.CompleteAct3Failed();
         Reset();
      }

      [state.IDLE] {
         // Define behaviour for start event of activity
         on p_Act3.StartAct3(): {
            state = Activity_states_t.EXECUTE;

            // Insert code to start the first actions
            // For every action in initials
            r_r1.Start_e();
         }
      }

      [state.EXECUTE] {
         on r_r1.Complete_e(): {
            r1_e_complete = true;
            r_r2.Start_f();
         }

         on r_r1.Complete_e_failed(): {
            r1_e_complete = true;
            activity_failed = true;
            r_r2.Start_f();
         }

         on r_r2.Complete_f(): {
            r2_f_complete = true;
            EndActivity();
         }

```

```
            on r_r2.Complete_f_failed(): {
               r2_f_complete = true;
               activity_failed = true;
               EndActivity();
            }
         }
      }
}
```

2. **Criticality level 2**

   (a) For Act_1 the generated Dezyne code is shown below.

      i. ***Interface***

```
interface IAct1 {
   // Define start and end events
   in void StartAct1();
   out void CompleteAct1();
   out void CompleteAct1Failed();

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      [state.IDLE] {
         // Define behaviour for start event of activity
         on StartAct1: {
            state = Activity_states_t.EXECUTE;
         }
      }

      [state.EXECUTE] {
         on StartAct1: illegal;
         on inevitable: {
            // Return end event of activity
            CompleteAct1;
            state = Activity_states_t.IDLE;
         }

         on inevitable: {
            // Return failed end event of activity
            CompleteAct1Failed;
            state = Activity_states_t.IDLE;
         }
      }
   }
}
```

      ii. ***Component***

```
import IAct1.dzn;
import Ir1.dzn;

component Act1_Comp {
   provides IAct1 p_Act1;
   requires Ir1 r_r1;

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      // For every action define a boolean variable
      bool r1_a_complete = false;
```

```
      bool r1_b_complete = false;

      void Reset() {
         state = Activity_states_t.IDLE;
         r1_a_complete = false;
         r1_b_complete = false;
      }

      void EndActivity() {
         p_Act1.CompleteAct1();
         Reset();
      }
      void ResetResources() {
         r_r1.Reset();
      }

      void ActivityFailed() {
         p_Act1.CompleteAct1Failed();
         Reset();
         ResetResources();
      }

      [state.IDLE] {
         // Define behaviour for start event of activity
         on p_Act1.StartAct1(): {
            state = Activity_states_t.EXECUTE;

            // Insert code to start the first actions
            // For every action in initials
            r_r1.Start_a();
         }
      }

      [state.EXECUTE] {
         on r_r1.Complete_a(): {
            r1_a_complete = true;
            r_r1.Start_b();
         }

         on r_r1.Complete_a_failed(): {
            ActivityFailed();
         }

         on r_r1.Complete_b(): {
            r1_b_complete = true;
            EndActivity();
         }

         on r_r1.Complete_b_failed(): {
            ActivityFailed();
         }
      }
   }
}
```

(b) For Act_2 the generated Dezyne code is shown below.

i. ***Interface***

```
interface IAct2 {
   // Define start and end events
   in void StartAct2();
   out void CompleteAct2();
   out void CompleteAct2Failed();

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      [state.IDLE] {
         // Define behaviour for start event of activity
         on StartAct2: {
            state = Activity_states_t.EXECUTE;
         }
      }

      [state.EXECUTE] {
         on StartAct2: illegal;
         on inevitable: {
            // Return end event of activity
            CompleteAct2;
            state = Activity_states_t.IDLE;
         }

         on inevitable: {
            // Return failed end event of activity
            CompleteAct2Failed;
            state = Activity_states_t.IDLE;
         }
      }
   }
}
```

ii. ***Component***

```
import IAct2.dzn;
import Ir1.dzn;
import Ir2.dzn;

component Act2_Comp {
   provides IAct2 p_Act2;
   requires Ir1 r_r1;
   requires Ir2 r_r2;

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      // For every action define a boolean variable
      bool r1_c_complete = false;
      bool r2_d_complete = false;

      void Reset() {
         state = Activity_states_t.IDLE;
         r1_c_complete = false;
         r2_d_complete = false;
      }

      void EndActivity() {
         p_Act2.CompleteAct2();
```

```
         Reset();
      }
      void ResetResources() {
         r_r1.Reset();
         r_r2.Reset();
      }

      void ActivityFailed() {
         p_Act2.CompleteAct2Failed();
         Reset();
         ResetResources();
      }

      [state.IDLE] {
         // Define behaviour for start event of activity
         on p_Act2.StartAct2(): {
            state = Activity_states_t.EXECUTE;

            // Insert code to start the first actions
            // For every action in initials
            r_r1.Start_c();
            r_r2.Start_d();
         }
      }

      [state.EXECUTE] {
         on r_r1.Complete_c(): {
            r1_c_complete = true;
            if (r2_d_complete) {
               EndActivity();
            }
         }

         on r_r1.Complete_c_failed(): {
            ActivityFailed();
         }

         on r_r2.Complete_d(): {
            r2_d_complete = true;
            if (r1_c_complete) {
               EndActivity();
            }
         }

         on r_r2.Complete_d_failed(): {
            ActivityFailed();
         }
      }
   }
}
```

(c) For Act_3 the generated Dezyne code is shown below.

i. ***Interface***

```
interface IAct3 {
   // Define start and end events
   in void StartAct3();
   out void CompleteAct3();
   out void CompleteAct3Failed();

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      [state.IDLE] {
```

```
         // Define behaviour for start event of activity
         on StartAct3: {
            state = Activity_states_t.EXECUTE;
         }
      }

      [state.EXECUTE] {
         on StartAct3: illegal;
         on inevitable: {
            // Return end event of activity
            CompleteAct3;
            state = Activity_states_t.IDLE;
         }

         on inevitable: {
            // Return failed end event of activity
            CompleteAct3Failed;
            state = Activity_states_t.IDLE;
         }
      }
   }
}
```

ii. ***Component***

```
import IAct3.dzn;
import Ir1.dzn;
import Ir2.dzn;

component Act3_Comp {
   provides IAct3 p_Act3;
   requires Ir1 r_r1;
   requires Ir2 r_r2;

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      // For every action define a boolean variable
      bool r1_e_complete = false;
      bool r2_f_complete = false;

      void Reset() {
         state = Activity_states_t.IDLE;
         r1_e_complete = false;
         r2_f_complete = false;
      }

      void EndActivity() {
         p_Act3.CompleteAct3();
         Reset();
      }
      void ResetResources() {
         r_r1.Reset();
         r_r2.Reset();
      }

      void ActivityFailed() {
         p_Act3.CompleteAct3Failed();
         Reset();
         ResetResources();
      }

      [state.IDLE] {
         // Define behaviour for start event of activity
         on p_Act3.StartAct3(): {
```

```
                state = Activity_states_t.EXECUTE;

                // Insert code to start the first actions
                // For every action in initials
                r_r1.Start_e();
            }
        }

        [state.EXECUTE] {
            on r_r1.Complete_e(): {
                r1_e_complete = true;
                r_r2.Start_f();
            }

            on r_r1.Complete_e_failed(): {
                ActivityFailed();
            }

            on r_r2.Complete_f(): {
                r2_f_complete = true;
                EndActivity();
            }

            on r_r2.Complete_f_failed(): {
                ActivityFailed();
            }
        }
    }
}
```

Sequence diagrams of the generated Dezyne code

1. **Criticality level 1**
   The behavior of the activities Act_1, Act_2 and Act_3 is shown as sequence diagram in Figure 5.5, Figure 5.6 and Figure 5.7 respectively. It is assumed that action a, action c and action e fail in activities Act_1, Act_2 and Act_3 respectively.

2. **Criticality level 2**
   The behavior of activities Act_1, Act_2 and Act_3 is shown as sequence diagram in Figure 5.8, Figure 5.9 and Figure 5.10 respectively. It is assumed that action a, action c and action e fail in activities Act_1, Act_2 and Act_3 respectively.

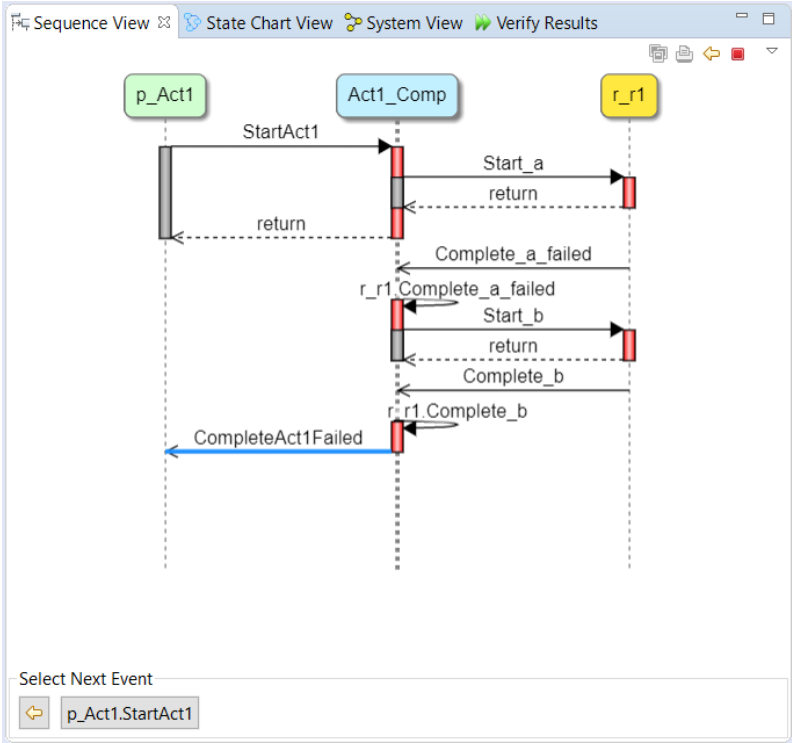


Figure 5.5: Sequence diagram for the activity Act_1

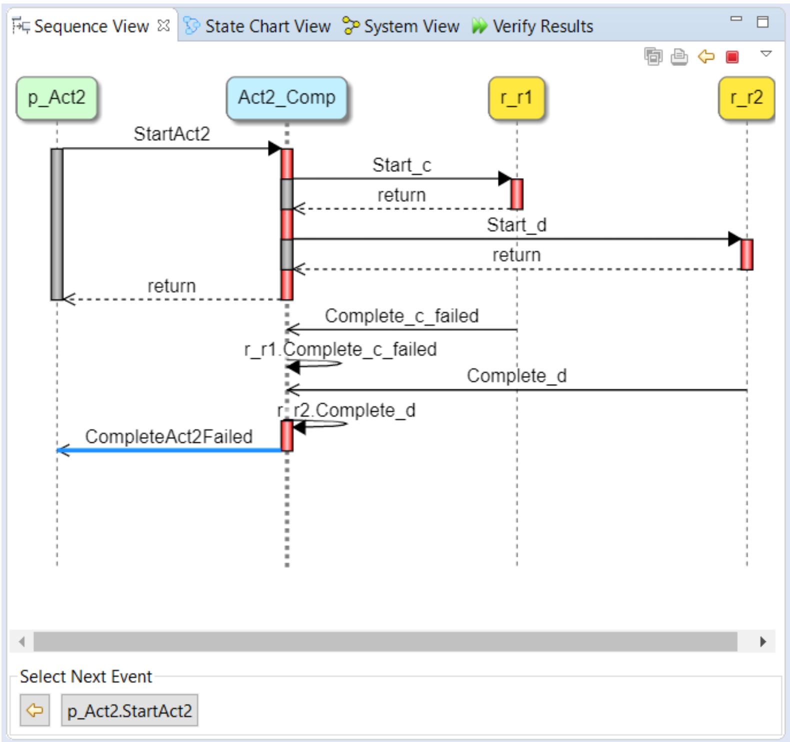


Figure 5.6: Sequence diagram for the activity Act_2

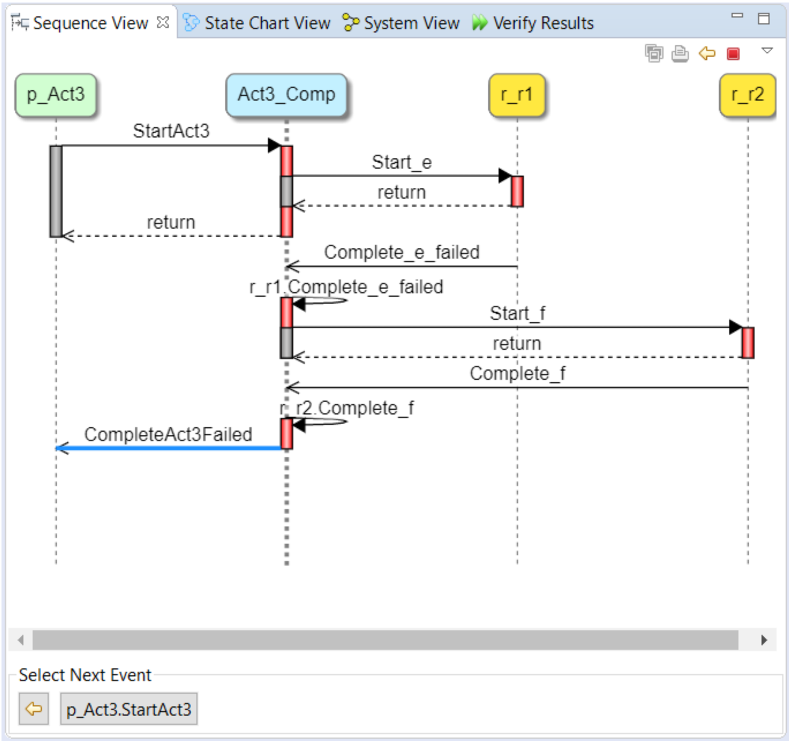


Figure 5.7: Sequence diagram for the activity Act_3

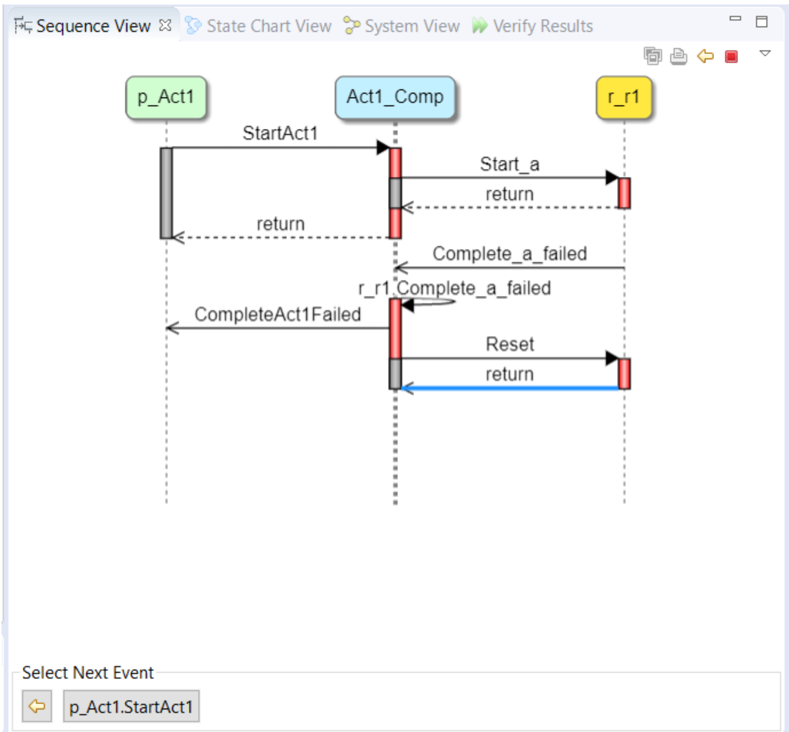


Figure 5.8: Sequence diagram for the activity Act_1

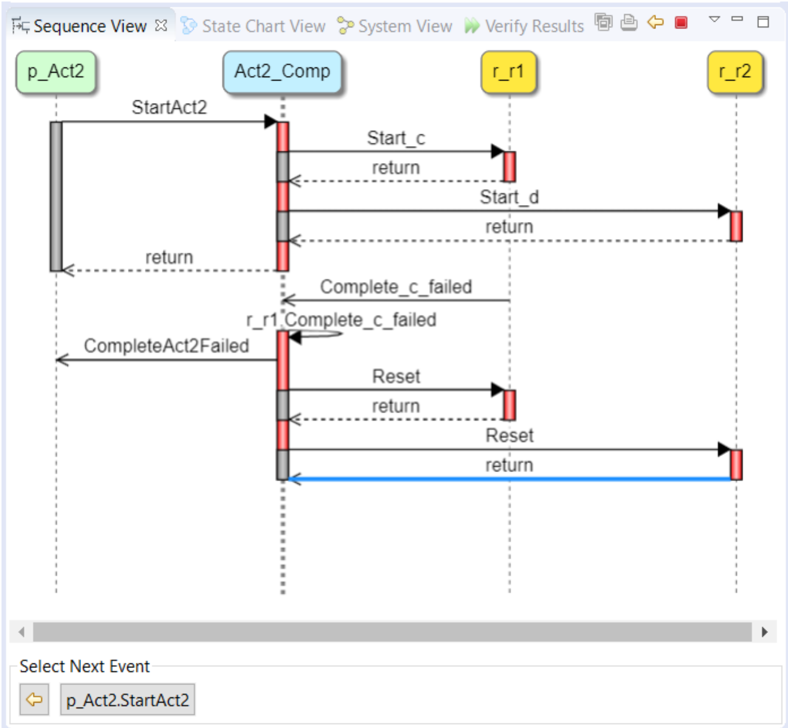


Figure 5.9: Sequence diagram for the activity Act_2

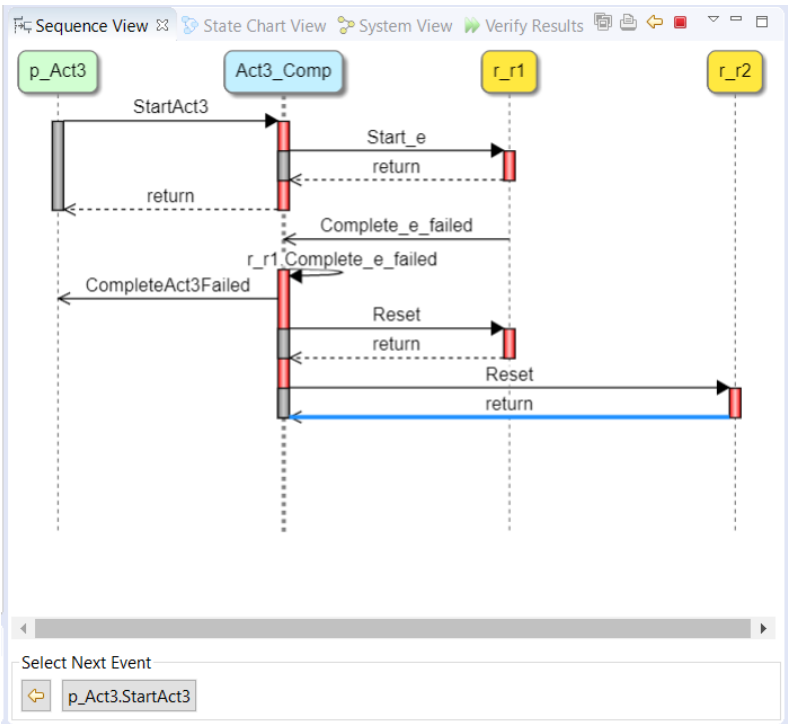


Figure 5.10: Sequence diagram for the activity Act_3

5.9 Verification of the generated Dezyne code

The activities modeled are verified in the Dezyne environment for properties described in section 3.9.

1. **Criticality level 1**
   The verification results for Act_1, Act_2 and Act_3 is shown in Figure 5.11, Figure 5.12 and Figure 5.13 respectively.

Sequence View | State Chart View | System View | Verify Results

Act1_Component.dzn

Check	Action	Time	States	Transitions	Done	Result
IAct1						
Deadlock		0:00	8	10	100%	✓
Livelock		0:00	8	10	100%	✓
Ir1						
Deadlock		0:00	20	43	100%	✓
Livelock		0:00	20	43	100%	✓
Act1_Comp						
Deterministic		0:00	30	36	100%	✓
Illegal		0:00	30	36	100%	✓
Deadlock		0:00	30	36	100%	✓
Livelock		0:00	30	36	100%	✓
Compliance		0:00	30	36	100%	✓

Figure 5.11: Verification result for the activity Act_1

Sequence View | State Chart View | System View | Verify Results

Act2_Component.dzn

Check	Action	Time	States	Transitions	Done	Result
IAct2						
Deadlock		0:00	8	10	100%	✓
Livelock		0:00	8	10	100%	✓
Ir1						
Deadlock		0:00	20	43	100%	✓
Livelock		0:00	20	43	100%	✓
Ir2						
Deadlock		0:00	12	19	100%	✓
Livelock		0:00	12	19	100%	✓
Act2_Comp						
Deterministic		0:00	45	57	100%	✓
Illegal		0:00	45	57	100%	✓
Deadlock		0:00	45	57	100%	✓
Livelock		0:00	45	57	100%	✓
Compliance		0:00	45	57	100%	✓

Figure 5.12: Verification result for the activity Act_2

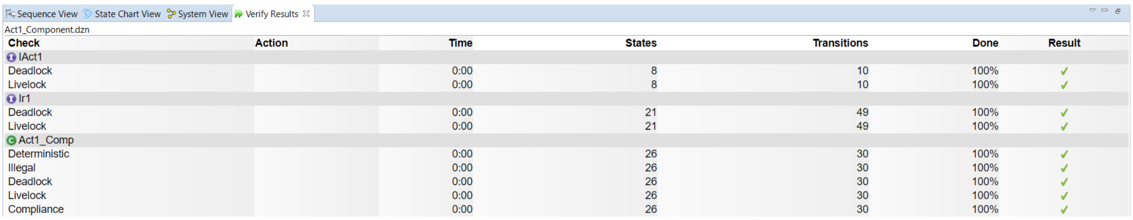
Sequence View | State Chart View | System View | Verify Results

Act3_Component.dzn

Check	Action	Time	States	Transitions	Done	Result
IAct3						
Deadlock		0:00	8	10	100%	✓
Livelock		0:00	8	10	100%	✓
Ir1						
Deadlock		0:00	20	43	100%	✓
Livelock		0:00	20	43	100%	✓
Ir2						
Deadlock		0:00	12	19	100%	✓
Livelock		0:00	12	19	100%	✓
Act3_Comp						
Deterministic		0:00	30	36	100%	✓
Illegal		0:00	30	36	100%	✓
Deadlock		0:00	30	36	100%	✓
Livelock		0:00	30	36	100%	✓
Compliance		0:00	30	36	100%	✓

Figure 5.13: Verification result for the activity Act_3

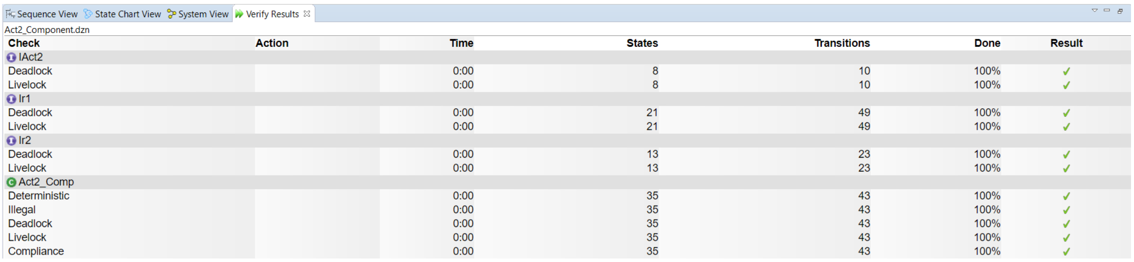
2. **Criticality level 2**
   The verification results for Act_1, Act_2 and Act_3 is shown in Figure 5.14, Figure 5.15 and Figure 5.16 respectively.


Sequence View | State Chart View | System View | Verify Results

Act1_Component.dzn

Check	Action	Time	States	Transitions	Done	Result
IAct1						
Deadlock		0:00	8	10	100%	✓
Livelock		0:00	8	10	100%	✓
Ir1						
Deadlock		0:00	21	49	100%	✓
Livelock		0:00	21	49	100%	✓
Act1_Comp						
Deterministic		0:00	26	30	100%	✓
Illegal		0:00	26	30	100%	✓
Deadlock		0:00	26	30	100%	✓
Livelock		0:00	26	30	100%	✓
Compliance		0:00	26	30	100%	✓

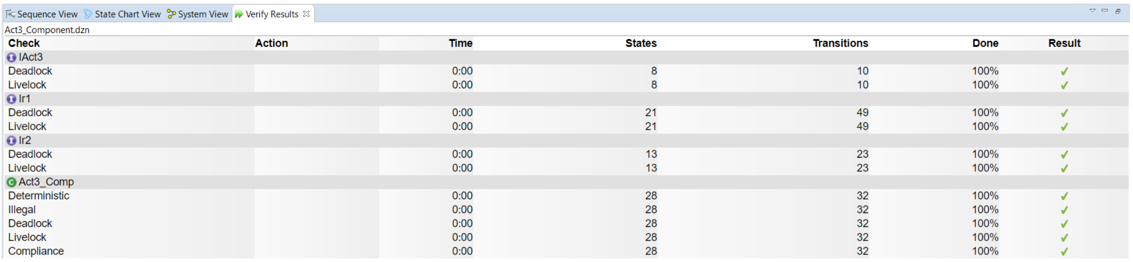
Figure 5.14: Verification result for the activity Act_1


Sequence View | State Chart View | System View | Verify Results

Act2_Component.dzn

Check	Action	Time	States	Transitions	Done	Result
IAct2						
Deadlock		0:00	8	10	100%	✓
Livelock		0:00	8	10	100%	✓
Ir1						
Deadlock		0:00	21	49	100%	✓
Livelock		0:00	21	49	100%	✓
Ir2						
Deadlock		0:00	13	23	100%	✓
Livelock		0:00	13	23	100%	✓
Act2_Comp						
Deterministic		0:00	35	43	100%	✓
Illegal		0:00	35	43	100%	✓
Deadlock		0:00	35	43	100%	✓
Livelock		0:00	35	43	100%	✓
Compliance		0:00	35	43	100%	✓

Figure 5.15: Verification result for the activity Act_2


Sequence View | State Chart View | System View | Verify Results

Act3_Component.dzn

Check	Action	Time	States	Transitions	Done	Result
IAct3						
Deadlock		0:00	8	10	100%	✓
Livelock		0:00	8	10	100%	✓
Ir1						
Deadlock		0:00	21	49	100%	✓
Livelock		0:00	21	49	100%	✓
Ir2						
Deadlock		0:00	13	23	100%	✓
Livelock		0:00	13	23	100%	✓
Act3_Comp						
Deterministic		0:00	28	32	100%	✓
Illegal		0:00	28	32	100%	✓
Deadlock		0:00	28	32	100%	✓
Livelock		0:00	28	32	100%	✓
Compliance		0:00	28	32	100%	✓

Figure 5.16: Verification result for the activity Act_3

Chapter 6

Case Study: Handling Exceptions

The translation from the activity models to the Dezyne code is achieved using the following five steps:

1. Define the events for handling exceptions, that is the failure event for the activity, and the actions contained in that activity.
2. Incorporate exceptions in the Activity DSL and in the Transformation DSL by specifying criticality levels in the activity model and adding failed events to the translation model.
3. Generate the Dezyne code from the model.
4. Verify the generated code in the Dezyne environment.

6.1 Defining the events for handling exceptions

When an exception occurs, the failure events need to be defined, in addition to the start and end events for the activity's software counterpart, and actions contained within that activity. The events defined for the *Warehouse Activity* is shown in Figure 6.1.

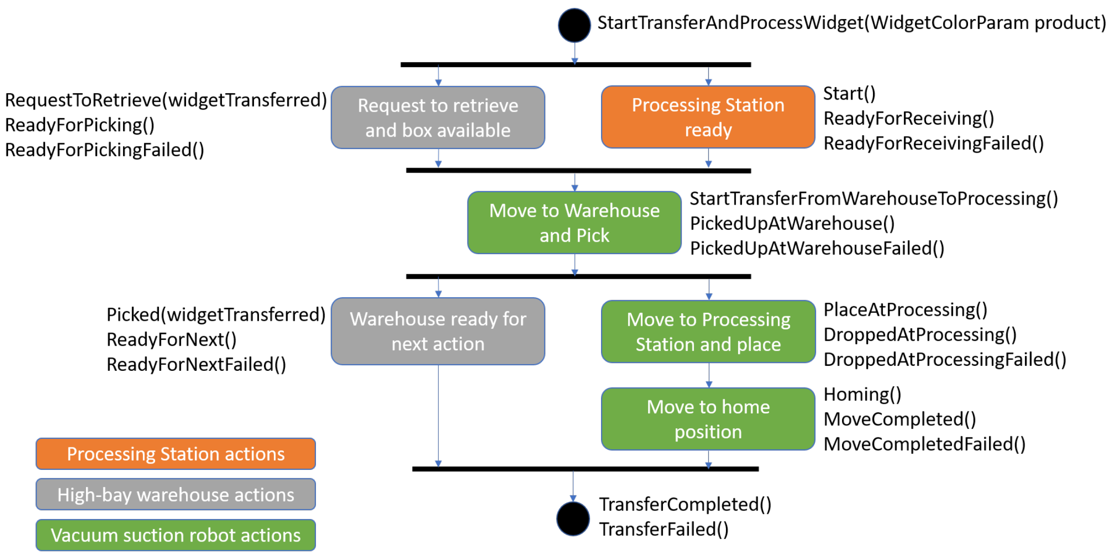


Figure 6.1: Events in Dezyne mapped to the *Warehouse Activity*

Similarly, the events defined for the *Color Sorter Activity* is shown in Figure 6.2.

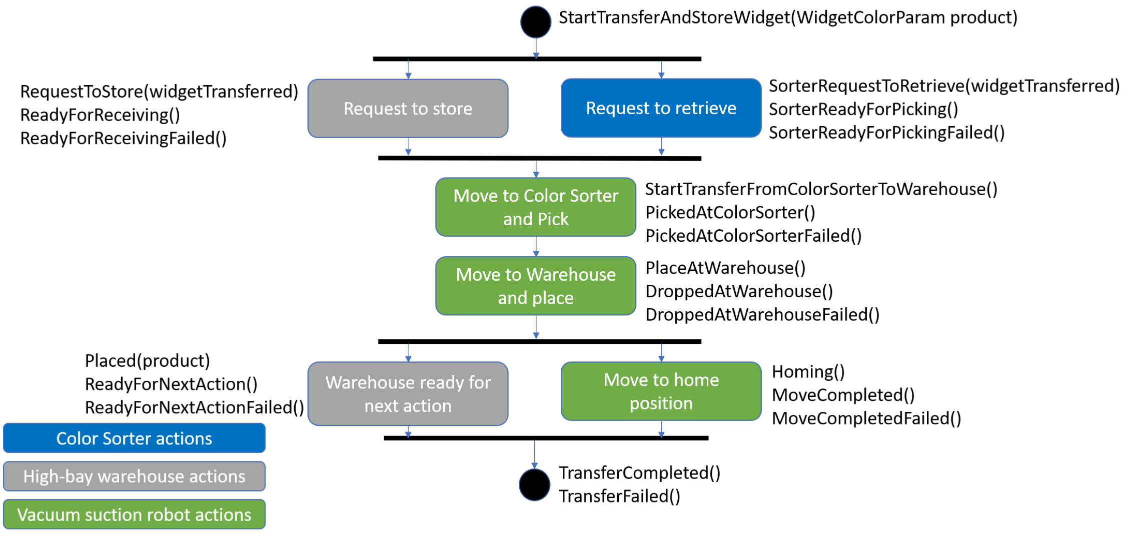


Figure 6.2: Events in Dezyne mapped to the *Color Sorter Activity*

6.2 Incorporating the exceptions in the Domain Specific Language

Since exceptions can be handled in two different ways, there are two criticality levels that can be defined: criticality level 1 and criticality level 2 in the Activity DSL.

6.2.1 Activity model

Criticality level 1

The activity model of the Factory Four model for criticality level 1 is shown below.

```
Resource Robot
   Action type MoveToWarehouseAndPick
   Action type MoveToOvenAndPlace
   Action type MoveToHomePosition
   Action type MoveToColorSorterAndPick
   Action type MoveToWarehouseAndPlace
Resource Warehouse
   Action type RequestToRetrieve
   Action type RequestToStore
   Action type ReadyForNext
   Action type ReadyForNextAction
Resource ProcessingStation
   Action type Ready
Resource ColorSorter
   Action type SorterRequestToRetrieve

Activity WarehouseActivity
   Criticality level 1
   a1 : Warehouse.RequestToRetrieve
   a2 : ProcessingStation.Ready
   a3 : Robot.MoveToWarehouseAndPick
   a4 : Warehouse.ReadyForNext
   a5 : Robot.MoveToOvenAndPlace
   a6 : Robot.MoveToHomePosition
   Dependencies
   a1 -> a3
   a2 -> a3
   a3 -> a4
   a3 -> a5
   a5 -> a6
```

```

Activity ColorSorterActivity
   Criticality level 1
   b1 : Warehouse.RequestToStore
   b2 : ColorSorter.SorterRequestToRetrieve
   b3 : Robot.MoveToColorSorterAndPick
   b4 : Robot.MoveToWarehouseAndPlace
   b5 : Warehouse.ReadyForNextAction
   b6 : Robot.MoveToHomePosition
   Dependencies
   b1 -> b3
   b2 -> b3
   b3 -> b4
   b4 -> b5
   b4 -> b6
```

Criticality level 2

The activity model of the Factory Four model for criticality level 2 is shown below.

```
Resource Robot
   Action type MoveToWarehouseAndPick
   Action type MoveToOvenAndPlace
   Action type MoveToHomePosition
   Action type MoveToColorSorterAndPick
   Action type MoveToWarehouseAndPlace
Resource Warehouse
   Action type RequestToRetrieve
   Action type RequestToStore
   Action type ReadyForNext
   Action type ReadyForNextAction
Resource ProcessingStation
   Action type Ready
Resource ColorSorter
   Action type SorterRequestToRetrieve

Activity WarehouseActivity
   Criticality level 2
   a1 : Warehouse.RequestToRetrieve
   a2 : ProcessingStation.Ready
   a3 : Robot.MoveToWarehouseAndPick
   a4 : Warehouse.ReadyForNext
   a5 : Robot.MoveToOvenAndPlace
   a6 : Robot.MoveToHomePosition
   Dependencies
   a1 -> a3
   a2 -> a3
   a3 -> a4
   a3 -> a5
   a5 -> a6

Activity ColorSorterActivity
   Criticality level 2
   b1 : Warehouse.RequestToStore
   b2 : ColorSorter.SorterRequestToRetrieve
   b3 : Robot.MoveToColorSorterAndPick
   b4 : Robot.MoveToWarehouseAndPlace
   b5 : Warehouse.ReadyForNextAction
   b6 : Robot.MoveToHomePosition
   Dependencies
   b1 -> b3
   b2 -> b3
   b3 -> b4
   b4 -> b5
   b4 -> b6
```

6.2.2 Translation model

The translation model of the Factory Four model is expressed in the Transformation DSL. The translation model is same for criticality level 1 and criticality level 2, which is shown below.

```
Activity WarehouseActivity
StartEvent: StartTransferAndProcessWidget(WidgetColorParam product)
EndEvent: TransferCompleted()
FailedEndEvent: TransferFailed()

Activity ColorSorterActivity
StartEvent: StartTransferAndStoreWidget(WidgetColorParam product)
EndEvent: TransferCompleted()
FailedEndEvent: TransferFailed()

Resource Robot
Interface: IRobot

Action MoveToWarehouseAndPick
StartEvent: StartTransferFromWarehouseToProcessing()
EndEvent: PickedUpAtWarehouse()
FailedEndEvent: PickedUpAtWarehouseFailed()

Action MoveToOvenAndPlace
StartEvent: PlaceAtProcessing()
EndEvent: DroppedAtProcessing()
FailedEndEvent: DroppedAtProcessingFailed()

Action MoveToHomePosition
StartEvent: Homing()
EndEvent: MoveCompleted()
FailedEndEvent: MoveCompletedFailed()

Action MoveToColorSorterAndPick
StartEvent: StartTransferFromColorSorterToWarehouse()
EndEvent: PickedAtColorSorter()
FailedEndEvent: PickedAtColorSorterFailed()

Action MoveToWarehouseAndPlace
StartEvent: PlaceAtWarehouse()
EndEvent: DroppedAtWarehouse()
FailedEndEvent: DroppedAtWarehouseFailed()

Resource Warehouse
Interface: IWarehouse

Action RequestToRetrieve
StartEvent: RequestToRetrieve(WidgetColorParam widgetTransferred)
EndEvent: ReadyForPicking()
FailedEndEvent:ReadyForPickingFailed()

Action ReadyForNext
StartEvent: Picked(widgetTransferred)
EndEvent: ReadyForNext()
FailedEndEvent: ReadyForNextFailed()

Action ReadyForNextAction
StartEvent: Placed(WidgetColorParam product)
EndEvent: ReadyForNextAction()
FailedEndEvent: ReadyForNextActionFailed()

Action RequestToStore
StartEvent: RequestToStore(WidgetColorParam widgetTransferred)
EndEvent: ReadyForReceiving()
FailedEndEvent: ReadyForReceivingFailed()

Resource ProcessingStation
Interface: IProcessingStation
```

```

Action Ready
StartEvent: Start()
EndEvent: ReadyForReceiving()
FailedEndEvent:ReadyForReceivingFailed()

Resource ColorSorter
Interface: IColorSorter

Action SorterRequestToRetrieve
StartEvent: SorterRequestToRetrieve(WidgetColorParam widgetTransferred)
EndEvent: SorterReadyForPicking()
FailedEndEvent: SorterReadyForPickingFailed()
```

6.3 Generated Dezyne code

Once the criticality levels for Factory Four model is defined and the events are specified, Dezyne code is generated.

6.3.1 Criticality level 1

For criticality level 1, the generated Dezyne code is shown below. The interface code for the resources of Factory Four model for criticality level 1 is given in A.2.2 for reference.

1. *Warehouse Activity*

   (a) ***Interface***

```
import Definitions.dzn;
interface IWarehouseActivity {
   // Define start and end events
   in void StartTransferAndProcessWidget(WidgetColorParam product);
   out void TransferCompleted();
   out void TransferFailed();

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      [state.IDLE] {
         // Define behaviour for start event of activity
         on StartTransferAndProcessWidget: {
            state = Activity_states_t.EXECUTE;
         }
      }

      [state.EXECUTE] {
         on StartTransferAndProcessWidget: illegal;
         on inevitable: {
            // Return end event of activity
            TransferCompleted;
            state = Activity_states_t.IDLE;
         }

         on inevitable: {
            // Return failed end event of activity
            TransferFailed;
            state = Activity_states_t.IDLE;
         }
      }
   }
}
```

(b) ***Component***

```
import IWarehouseActivity.dzn;
import IRobot.dzn;
import IWarehouse.dzn;
import IProcessingStation.dzn;

component WarehouseActivity_Comp {
   provides IWarehouseActivity p_WarehouseActivity;
   requires IRobot r_Robot;
   requires IWarehouse r_Warehouse;
   requires IProcessingStation r_ProcessingStation;

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      WidgetColorParam widgetTransferred;

      bool activity_failed = false;

      // For every action define a boolean variable
      bool processingStation_Ready_complete = false;
      bool robot_MoveToHomePosition_complete = false;
      bool robot_MoveToOvenAndPlace_complete = false;
      bool robot_MoveToWarehouseAndPick_complete = false;
      bool warehouse_ReadyForNext_complete = false;
      bool warehouse_RequestToRetrieve_complete = false;

      void Reset() {
         state = Activity_states_t.IDLE;
         processingStation_Ready_complete = false;
         robot_MoveToHomePosition_complete = false;
         robot_MoveToOvenAndPlace_complete = false;
         robot_MoveToWarehouseAndPick_complete = false;
         warehouse_ReadyForNext_complete = false;
         warehouse_RequestToRetrieve_complete = false;
         activity_failed = false;
      }

      void EndActivity() {
         if (activity_failed) {
            ActivityFailed();
         }
         else {
            p_WarehouseActivity.TransferCompleted();
            Reset();
         }
      }

      void ActivityFailed() {
         p_WarehouseActivity.TransferFailed();
         Reset();
      }

      [state.IDLE] {
         // Define behaviour for start event of activity
         on p_WarehouseActivity.StartTransferAndProcessWidget(product): {
            state = Activity_states_t.EXECUTE;

            // Insert code to start the first actions
            // For every action in initials
            r_ProcessingStation.Start();
            r_Warehouse.RequestToRetrieve(widgetTransferred);
         }
      }
```

```
    [state.EXECUTE] {
       on r_Warehouse.ReadyForPicking(): {
          warehouse_RequestToRetrieve_complete = true;
          if (processingStation_Ready_complete) {
             r_Robot.StartTransferFromWarehouseToProcessing();
          }
       }

       on r_Warehouse.ReadyForPickingFailed(): {
          warehouse_RequestToRetrieve_complete = true;
          activity_failed = true;
          if (processingStation_Ready_complete) {
             r_Robot.StartTransferFromWarehouseToProcessing();
          }
       }

       on r_ProcessingStation.ReadyForReceiving(): {
          processingStation_Ready_complete = true;
          if (warehouse_RequestToRetrieve_complete) {
             r_Robot.StartTransferFromWarehouseToProcessing();
          }
       }

       on r_ProcessingStation.ReadyForReceivingFailed(): {
          processingStation_Ready_complete = true;
          activity_failed = true;
          if (warehouse_RequestToRetrieve_complete) {
             r_Robot.StartTransferFromWarehouseToProcessing();
          }
       }

       on r_Robot.PickedUpAtWarehouse(): {
          robot_MoveToWarehouseAndPick_complete = true;
          r_Warehouse.Picked(widgetTransferred);
          r_Robot.PlaceAtProcessing();
       }

       on r_Robot.PickedUpAtWarehouseFailed(): {
          robot_MoveToWarehouseAndPick_complete = true;
          activity_failed = true;
          r_Warehouse.Picked(widgetTransferred);
          r_Robot.PlaceAtProcessing();
       }

       on r_Warehouse.ReadyForNext(): {
          warehouse_ReadyForNext_complete = true;
          if (robot_MoveToHomePosition_complete) {
             EndActivity();
          }
       }

       on r_Warehouse.ReadyForNextFailed(): {
          warehouse_ReadyForNext_complete = true;
          activity_failed = true;
          if (robot_MoveToHomePosition_complete) {
             EndActivity();
          }
       }

       on r_Robot.DroppedAtProcessing(): {
          robot_MoveToOvenAndPlace_complete = true;
          r_Robot.Homing();
       }

       on r_Robot.DroppedAtProcessingFailed(): {
          robot_MoveToOvenAndPlace_complete = true;
```

```
                activity_failed = true;
                r_Robot.Homing();
            }

            on r_Robot.MoveCompleted(): {
                robot_MoveToHomePosition_complete = true;
                if (warehouse_ReadyForNext_complete) {
                    EndActivity();
                }
            }

            on r_Robot.MoveCompletedFailed(): {
                robot_MoveToHomePosition_complete = true;
                activity_failed = true;
                if (warehouse_ReadyForNext_complete) {
                    EndActivity();
                }
            }
        }
    }
}
```

2. *Color Sorter Activity*

(a) ***Interface***

```
import Definitions.dzn;
interface IColorSorterActivity {
    // Define start and end events
    in void StartTransferAndStoreWidget(WidgetColorParam product);
    out void TransferCompleted();
    out void TransferFailed();

    behaviour {
        // Define two states
        enum Activity_states_t { IDLE, EXECUTE };
        // Set initial state to IDLE
        Activity_states_t state = Activity_states_t.IDLE;

        [state.IDLE] {
            // Define behaviour for start event of activity
            on StartTransferAndStoreWidget: {
                state = Activity_states_t.EXECUTE;
            }
        }

        [state.EXECUTE] {
            on StartTransferAndStoreWidget: illegal;
            on inevitable: {
                // Return end event of activity
                TransferCompleted;
                state = Activity_states_t.IDLE;
            }

            on inevitable: {
                // Return failed end event of activity
                TransferFailed;
                state = Activity_states_t.IDLE;
            }
        }
    }
}
```

(b) ***Component***

```

import IColorSorterActivity.dzn;
import IRobot.dzn;
import IWarehouse.dzn;
import IColorSorter.dzn;

component ColorSorterActivity_Comp {
  provides IColorSorterActivity p_ColorSorterActivity;
  requires IRobot r_Robot;
  requires IWarehouse r_Warehouse;
  requires IColorSorter r_ColorSorter;

  behaviour {
    // Define two states
    enum Activity_states_t { IDLE, EXECUTE };
    // Set initial state to IDLE
    Activity_states_t state = Activity_states_t.IDLE;

    WidgetColorParam widgetTransferred;
    WidgetColorParam product;

    bool activity_failed = false;

    // For every action define a boolean variable
    bool colorSorter_SorterRequestToRetrieve_complete = false;
    bool robot_MoveToColorSorterAndPick_complete = false;
    bool robot_MoveToHomePosition_complete = false;
    bool robot_MoveToWarehouseAndPlace_complete = false;
    bool warehouse_ReadyForNextAction_complete = false;
    bool warehouse_RequestToStore_complete = false;

    void Reset() {
      state = Activity_states_t.IDLE;
      colorSorter_SorterRequestToRetrieve_complete = false;
      robot_MoveToColorSorterAndPick_complete = false;
      robot_MoveToHomePosition_complete = false;
      robot_MoveToWarehouseAndPlace_complete = false;
      warehouse_ReadyForNextAction_complete = false;
      warehouse_RequestToStore_complete = false;
      activity_failed = false;
    }

    void EndActivity() {
      if (activity_failed) {
        ActivityFailed();
      }
      else {
        p_ColorSorterActivity.TransferCompleted();
        Reset();
      }
    }

    void ActivityFailed() {
      p_ColorSorterActivity.TransferFailed();
      Reset();
    }

    [state.IDLE] {
      // Define behaviour for start event of activity
      on p_ColorSorterActivity.StartTransferAndStoreWidget(product): {
        state = Activity_states_t.EXECUTE;

        // Insert code to start the first actions
        // For every action in initials
        r_ColorSorter.SorterRequestToRetrieve(widgetTransferred);
        r_Warehouse.RequestToStore(widgetTransferred);
```

```
      }
    }

    [state.EXECUTE] {
      on r_Warehouse.ReadyForReceiving(): {
        warehouse_RequestToStore_complete = true;
        if (colorSorter_SorterRequestToRetrieve_complete) {
          r_Robot.StartTransferFromColorSorterToWarehouse();
        }
      }

      on r_Warehouse.ReadyForReceivingFailed(): {
        warehouse_RequestToStore_complete = true;
        activity_failed = true;
        if (colorSorter_SorterRequestToRetrieve_complete) {
          r_Robot.StartTransferFromColorSorterToWarehouse();
        }
      }

      on r_ColorSorter.SorterReadyForPicking(): {
        colorSorter_SorterRequestToRetrieve_complete = true;
        if (warehouse_RequestToStore_complete) {
          r_Robot.StartTransferFromColorSorterToWarehouse();
        }
      }

      on r_ColorSorter.SorterReadyForPickingFailed(): {
        colorSorter_SorterRequestToRetrieve_complete = true;
        activity_failed = true;
        if (warehouse_RequestToStore_complete) {
          r_Robot.StartTransferFromColorSorterToWarehouse();
        }
      }

      on r_Robot.PickedAtColorSorter(): {
        robot_MoveToColorSorterAndPick_complete = true;
        r_Robot.PlaceAtWarehouse();
      }

      on r_Robot.PickedAtColorSorterFailed(): {
        robot_MoveToColorSorterAndPick_complete = true;
        activity_failed = true;
        r_Robot.PlaceAtWarehouse();
      }

      on r_Robot.DroppedAtWarehouse(): {
        robot_MoveToWarehouseAndPlace_complete = true;
        r_Warehouse.Placed(product);
        r_Robot.Homing();
      }

      on r_Robot.DroppedAtWarehouseFailed(): {
        robot_MoveToWarehouseAndPlace_complete = true;
        activity_failed = true;
        r_Warehouse.Placed(product);
        r_Robot.Homing();
      }

      on r_Warehouse.ReadyForNextAction(): {
        warehouse_ReadyForNextAction_complete = true;
        if (robot_MoveToHomePosition_complete) {
          EndActivity();
        }
      }

      on r_Warehouse.ReadyForNextActionFailed(): {
        warehouse_ReadyForNextAction_complete = true;
```

```
                activity_failed = true;
                if (robot_MoveToHomePosition_complete) {
                    EndActivity();
                }
            }

            on r_Robot.MoveCompleted(): {
                robot_MoveToHomePosition_complete = true;
                if (warehouse_ReadyForNextAction_complete) {
                    EndActivity();
                }
            }

            on r_Robot.MoveCompletedFailed(): {
                robot_MoveToHomePosition_complete = true;
                activity_failed = true;
                if (warehouse_ReadyForNextAction_complete) {
                    EndActivity();
                }
            }
        }
    }
}
```

6.3.2 Criticality level 2

For criticality level 2, the generated Dezyne code is shown below. The interface code for the resources of the Factory Four model for criticality level 2 is given in A.2.3 for reference.

1. *Warehouse Activity*

   (a) ***Interface***

```
import Definitions.dzn;

interface IWarehouseActivity {
    // Define start and end events
    in void StartTransferAndProcessWidget(WidgetColorParam product);
    out void TransferCompleted();
    out void TransferFailed();

    behaviour {
        // Define two states
        enum Activity_states_t { IDLE, EXECUTE };
        // Set initial state to IDLE
        Activity_states_t state = Activity_states_t.IDLE;

        [state.IDLE] {
            // Define behaviour for start event of activity
            on StartTransferAndProcessWidget: {
                state = Activity_states_t.EXECUTE;
            }
        }

        [state.EXECUTE] {
            on StartTransferAndProcessWidget: illegal;
            on inevitable: {
                // Return end event of activity
                TransferCompleted;
                state = Activity_states_t.IDLE;
            }

            on inevitable: {
                // Return failed end event of activity
                TransferFailed;
                state = Activity_states_t.IDLE;
```

```
            }
        }
    }
}
```

(b) ***Component***

```
import IWarehouseActivity.dzn;
import IRobot.dzn;
import IWarehouse.dzn;
import IProcessingStation.dzn;

component WarehouseActivity_Comp {
   provides IWarehouseActivity p_WarehouseActivity;
   requires IRobot r_Robot;
   requires IWarehouse r_Warehouse;
   requires IProcessingStation r_ProcessingStation;

   behaviour {
      // Define two states
      enum Activity_states_t { IDLE, EXECUTE };
      // Set initial state to IDLE
      Activity_states_t state = Activity_states_t.IDLE;

      WidgetColorParam widgetTransferred;

      // For every action define a boolean variable
      bool processingStation_Ready_complete = false;
      bool robot_MoveToHomePosition_complete = false;
      bool robot_MoveToOvenAndPlace_complete = false;
      bool robot_MoveToWarehouseAndPick_complete = false;
      bool warehouse_ReadyForNext_complete = false;
      bool warehouse_RequestToRetrieve_complete = false;

      void Reset() {
         state = Activity_states_t.IDLE;
         processingStation_Ready_complete = false;
         robot_MoveToHomePosition_complete = false;
         robot_MoveToOvenAndPlace_complete = false;
         robot_MoveToWarehouseAndPick_complete = false;
         warehouse_ReadyForNext_complete = false;
         warehouse_RequestToRetrieve_complete = false;
      }

      void EndActivity() {
         p_WarehouseActivity.TransferCompleted();
         Reset();
      }
      void ResetResources() {
         r_Robot.Reset();
         r_Warehouse.Reset();
         r_ProcessingStation.Reset();
      }

      void ActivityFailed() {
         p_WarehouseActivity.TransferFailed();
         Reset();
         ResetResources();
      }

      [state.IDLE] {
         // Define behaviour for start event of activity
         on p_WarehouseActivity.StartTransferAndProcessWidget(product): {
            state = Activity_states_t.EXECUTE;

            // Insert code to start the first actions
            // For every action in initials
            r_ProcessingStation.Start();
```

```
                r_Warehouse.RequestToRetrieve(widgetTransferred);
            }
        }

        [state.EXECUTE] {
            on r_Warehouse.ReadyForPicking(): {
                warehouse_RequestToRetrieve_complete = true;
                if (processingStation_Ready_complete) {
                    r_Robot.StartTransferFromWarehouseToProcessing();
                }
            }

            on r_Warehouse.ReadyForPickingFailed(): {
                ActivityFailed();
            }

            on r_ProcessingStation.ReadyForReceiving(): {
                processingStation_Ready_complete = true;
                if (warehouse_RequestToRetrieve_complete) {
                    r_Robot.StartTransferFromWarehouseToProcessing();
                }
            }

            on r_ProcessingStation.ReadyForReceivingFailed(): {
                ActivityFailed();
            }

            on r_Robot.PickedUpAtWarehouse(): {
                robot_MoveToWarehouseAndPick_complete = true;
                r_Warehouse.Picked(widgetTransferred);
                r_Robot.PlaceAtProcessing();
            }

            on r_Robot.PickedUpAtWarehouseFailed(): {
                ActivityFailed();
            }

            on r_Warehouse.ReadyForNext(): {
                warehouse_ReadyForNext_complete = true;
                if (robot_MoveToHomePosition_complete) {
                    EndActivity();
                }
            }

            on r_Warehouse.ReadyForNextFailed(): {
                ActivityFailed();
            }

            on r_Robot.DroppedAtProcessing(): {
                robot_MoveToOvenAndPlace_complete = true;
                r_Robot.Homing();
            }

            on r_Robot.DroppedAtProcessingFailed(): {
                ActivityFailed();
            }

            on r_Robot.MoveCompleted(): {
                robot_MoveToHomePosition_complete = true;
                if (warehouse_ReadyForNext_complete) {
                    EndActivity();
                }
            }

            on r_Robot.MoveCompletedFailed(): {
                ActivityFailed();
            }
```

```
        }
    }
}
```

2. *Color Sorter Activity*

    (a) ***Interface***

```
import Definitions.dzn;

interface IColorSorterActivity {
    // Define start and end events
    in void StartTransferAndStoreWidget(WidgetColorParam product);
    out void TransferCompleted();
    out void TransferFailed();

    behaviour {
        // Define two states
        enum Activity_states_t { IDLE, EXECUTE };
        // Set initial state to IDLE
        Activity_states_t state = Activity_states_t.IDLE;

        [state.IDLE] {
            // Define behaviour for start event of activity
            on StartTransferAndStoreWidget: {
                state = Activity_states_t.EXECUTE;
            }
        }

        [state.EXECUTE] {
            on StartTransferAndStoreWidget: illegal;
            on inevitable: {
                // Return end event of activity
                TransferCompleted;
                state = Activity_states_t.IDLE;
            }

            on inevitable: {
                // Return failed end event of activity
                TransferFailed;
                state = Activity_states_t.IDLE;
            }
        }
    }
}
```

    (b) ***Component***

```
import IColorSorterActivity.dzn;
import IRobot.dzn;
import IWarehouse.dzn;
import IColorSorter.dzn;

component ColorSorterActivity_Comp {
    provides IColorSorterActivity p_ColorSorterActivity;
    requires IRobot r_Robot;
    requires IWarehouse r_Warehouse;
    requires IColorSorter r_ColorSorter;

    behaviour {
        // Define two states
        enum Activity_states_t { IDLE, EXECUTE };
        // Set initial state to IDLE
        Activity_states_t state = Activity_states_t.IDLE;

        WidgetColorParam widgetTransferred;
        WidgetColorParam product;
```

```

    // For every action define a boolean variable
    bool colorSorter_SorterRequestToRetrieve_complete = false;
    bool robot_MoveToColorSorterAndPick_complete = false;
    bool robot_MoveToHomePosition_complete = false;
    bool robot_MoveToWarehouseAndPlace_complete = false;
    bool warehouse_ReadyForNextAction_complete = false;
    bool warehouse_RequestToStore_complete = false;

    void Reset() {
       state = Activity_states_t.IDLE;
       colorSorter_SorterRequestToRetrieve_complete = false;
       robot_MoveToColorSorterAndPick_complete = false;
       robot_MoveToHomePosition_complete = false;
       robot_MoveToWarehouseAndPlace_complete = false;
       warehouse_ReadyForNextAction_complete = false;
       warehouse_RequestToStore_complete = false;
    }

    void EndActivity() {
       p_ColorSorterActivity.TransferCompleted();
       Reset();
    }
    void ResetResources() {
       r_Robot.Reset();
       r_Warehouse.Reset();
       r_ColorSorter.Reset();
    }

    void ActivityFailed() {
       p_ColorSorterActivity.TransferFailed();
       Reset();
       ResetResources();
    }

    [state.IDLE] {
       // Define behaviour for start event of activity
       on p_ColorSorterActivity.StartTransferAndStoreWidget(product): {
          state = Activity_states_t.EXECUTE;

          // Insert code to start the first actions
          // For every action in initials
          r_Warehouse.RequestToStore(widgetTransferred);
          r_ColorSorter.SorterRequestToRetrieve(widgetTransferred);
       }
    }

    [state.EXECUTE] {
       on r_Warehouse.ReadyForReceiving(): {
          warehouse_RequestToStore_complete = true;
          if (colorSorter_SorterRequestToRetrieve_complete) {
             r_Robot.StartTransferFromColorSorterToWarehouse();
          }
       }

       on r_Warehouse.ReadyForReceivingFailed(): {
          ActivityFailed();
       }

       on r_ColorSorter.SorterReadyForPicking(): {
          colorSorter_SorterRequestToRetrieve_complete = true;
          if (warehouse_RequestToStore_complete) {
             r_Robot.StartTransferFromColorSorterToWarehouse();
          }
       }

       on r_ColorSorter.SorterReadyForPickingFailed(): {
```

```
            ActivityFailed();
        }

        on r_Robot.PickedAtColorSorter(): {
            robot_MoveToColorSorterAndPick_complete = true;
            r_Robot.PlaceAtWarehouse();
        }

        on r_Robot.PickedAtColorSorterFailed(): {
            ActivityFailed();
        }

        on r_Robot.DroppedAtWarehouse(): {
            robot_MoveToWarehouseAndPlace_complete = true;
            r_Warehouse.Placed(product);
            r_Robot.Homing();
        }

        on r_Robot.DroppedAtWarehouseFailed(): {
            ActivityFailed();
        }

        on r_Warehouse.ReadyForNextAction(): {
            warehouse_ReadyForNextAction_complete = true;
            if (robot_MoveToHomePosition_complete) {
                EndActivity();
            }
        }

        on r_Warehouse.ReadyForNextActionFailed(): {
            ActivityFailed();
        }

        on r_Robot.MoveCompleted(): {
            robot_MoveToHomePosition_complete = true;
            if (warehouse_ReadyForNextAction_complete) {
                EndActivity();
            }
        }

        on r_Robot.MoveCompletedFailed(): {
            ActivityFailed();
        }
    }
  }
}
```

Sequence diagrams of the generated Dezyne code

1. **Criticality level 1**
The sequence diagram for the *Warehouse activity* and the *Color Sorter activity* is shown in Figures 6.3 and 6.4. It is assumed that actions *Processing Station ready* and *Move to Warehouse and Pick* fail in the *Warehouse activity*. Similarly, in the *Color Sorter activity* it is assumed that actions *Move to Warehouse and place* and *Warehouse ready for next action* fail to execute.

2. **Criticality level 2**
The sequence diagrams for the *Warehouse activity* and the *Color Sorter activity* are shown in Figure 6.5 and 6.6. It is assumed that action *Request to retrieve and box available* fails in the *Warehouse activity*. Similarly, in the *Color Sorter activity* it is assumed that action *Move to Color Sorter and Pick* fails to execute.

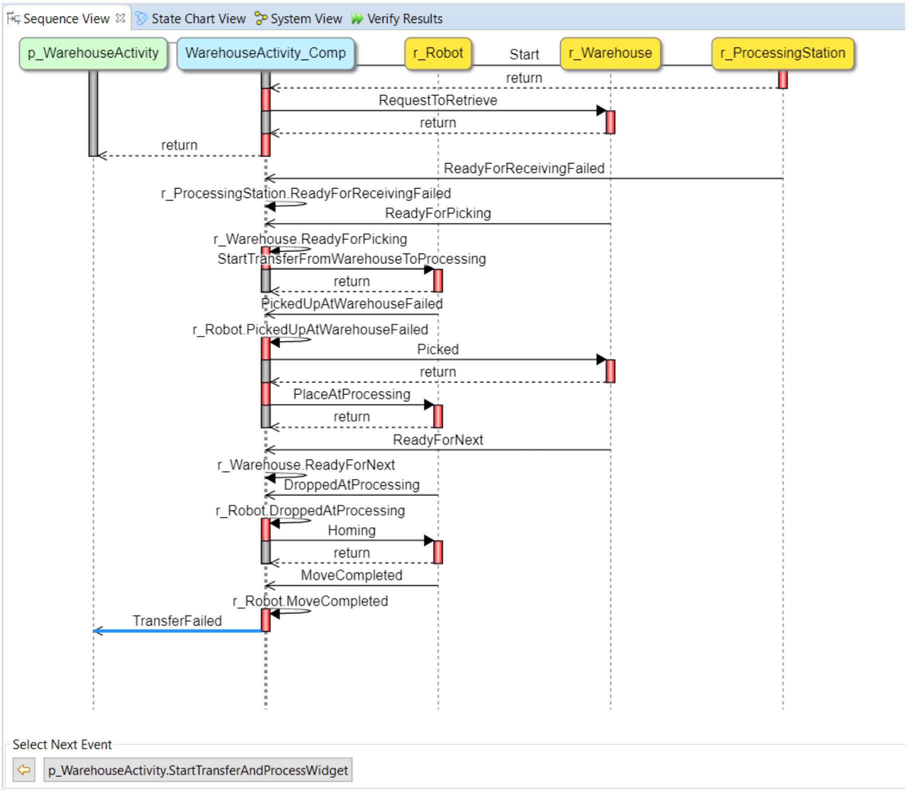


Figure 6.3: Sequence diagram for the *Warehouse Activity* for criticality level 1

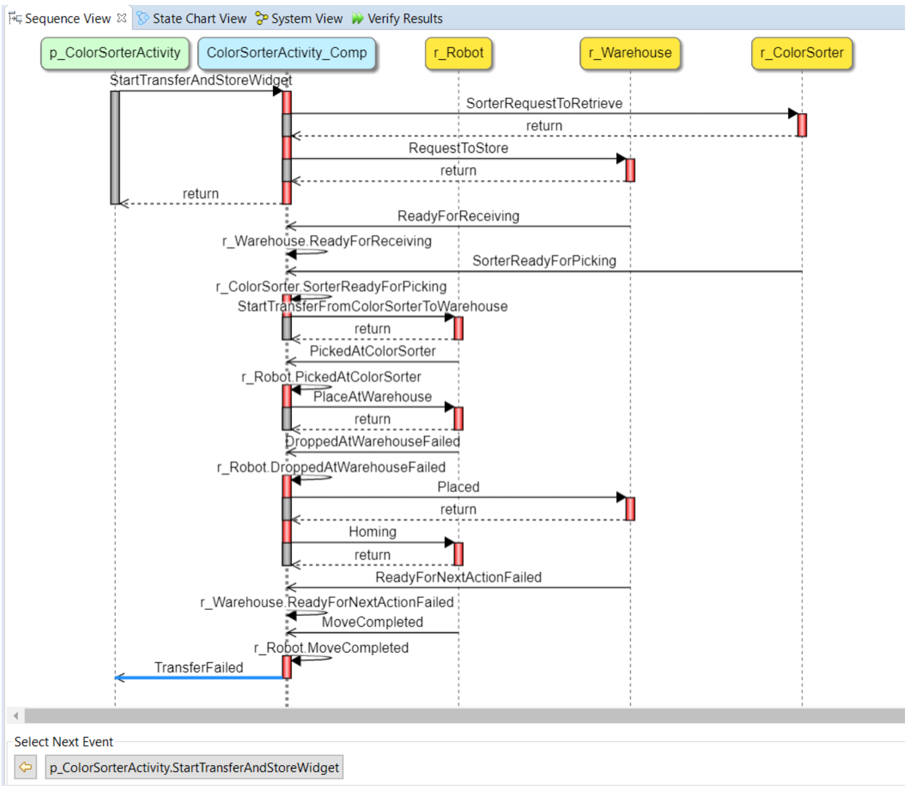


Figure 6.4: Sequence diagram for the *Color Sorter activity* for criticality level 1

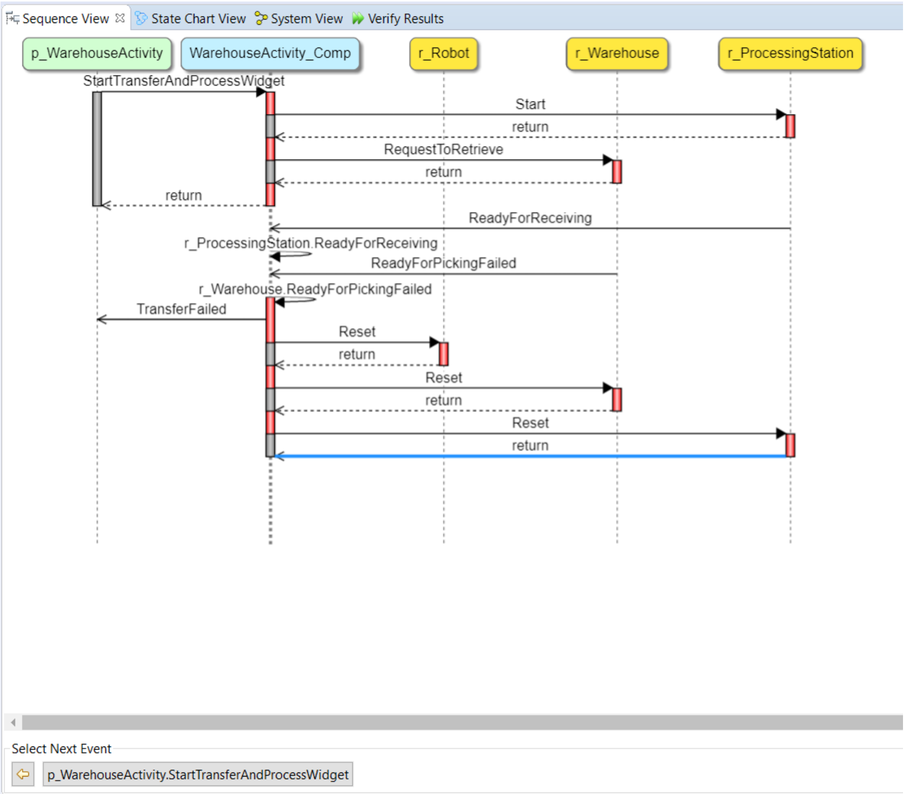


Figure 6.5: Sequence diagram for the *Warehouse Activity* for criticality level 2

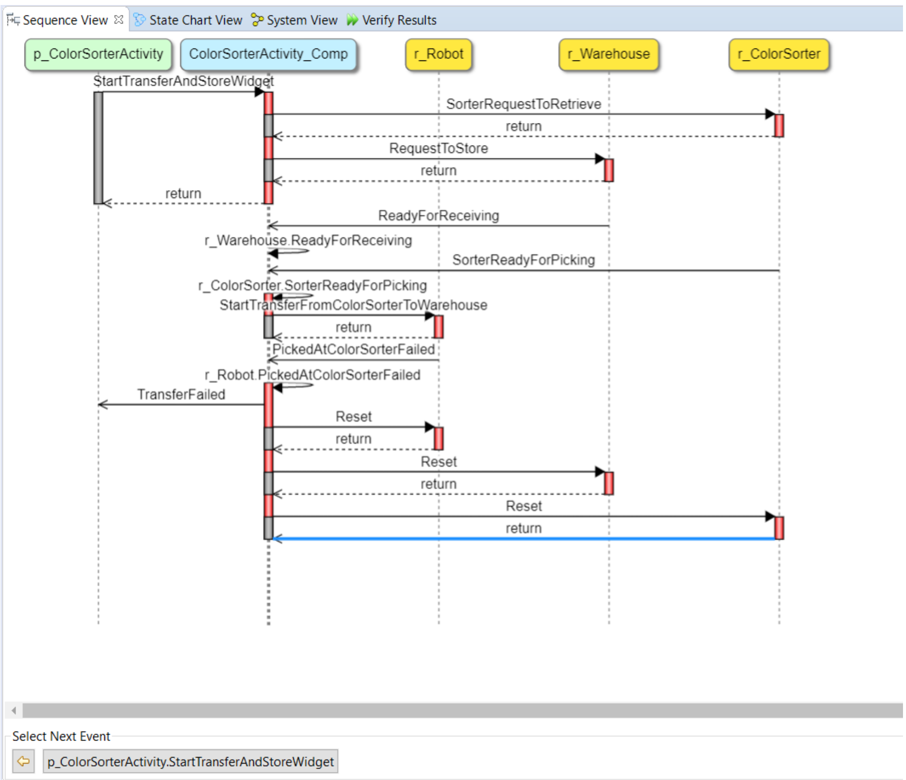


Figure 6.6: Sequence diagram for the *Color Sorter activity* for criticality level 2

6.4 Verifying the generated code

The verification results for the *Warehouse activity* and the *Color Sorter activity* in the Dezyne environment is shown below for different criticality levels.

1. **Criticality level 1**
   For criticality level 1, the verification results are shown in Figure 6.7 and Figure 6.8 for the *Warehouse activity* and the *Color Sorter activity* respectively.

Sequence View | State Chart View | System View | Verify Results

WarehouseActivity_Component.dzn

Check	Action	Time	States	Transitions	Done	Result
IWarehouseActivity						
Deadlock		0:00	8	10	100%	✓
Livelock		0:00	8	10	100%	✓
IRobot						
Deadlock		0:00	33	79	100%	✓
Livelock		0:00	33	79	100%	✓
IWarehouse						
Deadlock		0:00	26	55	100%	✓
Livelock		0:00	26	55	100%	✓
IProcessingStation						
Deadlock		0:00	8	10	100%	✓
Livelock		0:00	8	10	100%	✓
WarehouseActivity_Comp						
Deterministic		0:01	149	193	100%	✓
Illegal		0:01	149	193	100%	✓
Deadlock		0:01	149	193	100%	✓
Livelock		0:01	149	193	100%	✓
Compliance		0:00	149	193	100%	✓

Figure 6.7: Verification result for the *Warehouse Activity*

Sequence View | State Chart View | System View | Verify Results

ColorSorterActivity_Component.dzn

Check	Action	Time	States	Transitions	Done	Result
IColorSorterActivity						
Deadlock		0:00	8	10	100%	✓
Livelock		0:00	8	10	100%	✓
IRobot						
Deadlock		0:00	33	79	100%	✓
Livelock		0:00	33	79	100%	✓
IWarehouse						
Deadlock		0:00	26	55	100%	✓
Livelock		0:00	26	55	100%	✓
IColorSorter						
Deadlock		0:00	8	10	100%	✓
Livelock		0:00	8	10	100%	✓
ColorSorterActivity_Comp						
Deterministic		0:01	125	161	100%	✓
Illegal		0:01	125	161	100%	✓
Deadlock		0:01	125	161	100%	✓
Livelock		0:01	125	161	100%	✓
Compliance		0:00	125	161	100%	✓

Figure 6.8: Verification result for the *Color Sorter Activity*

2. **Criticality level 2**
   The verification results for the *Warehouse activity* and the *Color Sorter activity* in the Dezyne environment is shown in Figure 6.9 and Figure 6.10 respectively.

Sequence View | State Chart View | System View | Verify Results

WarehouseActivity_Component.dzn

Check	Action	Time	States	Transitions	Done	Result
IWarehouseActivity						
Deadlock		0:00	8	10	100%	✓
Livelock		0:00	8	10	100%	✓
IRobot						
Deadlock		0:00	34	89	100%	✓
Livelock		0:00	34	89	100%	✓
IWarehouse						
Deadlock		0:00	27	63	100%	✓
Livelock		0:00	27	63	100%	✓
IProcessingStation						
Deadlock		0:00	9	13	100%	✓
Livelock		0:00	9	13	100%	✓
WarehouseActivity_Comp						
Deterministic		0:01	89	113	100%	✓
Illegal		0:01	89	113	100%	✓
Deadlock		0:01	89	113	100%	✓
Livelock		0:01	89	113	100%	✓
Compliance		0:00	89	113	100%	✓

Figure 6.9: Verification result for the *Warehouse Activity*

Sequence View State Chart View System View Verify Results

ColorSorterActivity_Component.dzn

Check	Action	Time	States	Transitions	Done	Result
IColorSorterActivity						
Deadlock		0:00	8	10	100%	✓
Livelock		0:00	8	10	100%	✓
IRobot						
Deadlock		0:00	34	89	100%	✓
Livelock		0:00	34	89	100%	✓
IWarehouse						
Deadlock		0:00	27	63	100%	✓
Livelock		0:00	27	63	100%	✓
IColorSorter						
Deadlock		0:00	9	13	100%	✓
Livelock		0:00	9	13	100%	✓
ColorSorterActivity_Comp						
Deterministic		0:01	80	100	100%	✓
Illegal		0:01	80	100	100%	✓
Deadlock		0:01	80	100	100%	✓
Livelock		0:01	80	100	100%	✓
Compliance		0:00	80	100	100%	✓

Figure 6.10: Verification result for the *Color Sorter Activity*

Chapter 7

Conclusions and future work

The following sections conclude the work discussed in this report and suggest improvements for the solution which can be included in the future work.

7.1 Conclusion

The answers to the following research questions for an FMS is provided as follows:

1. How to generate Dezyne code from activity models?
   An FMS is modeled using the activity framework in terms of resources, actions and activities, which can be specified in LSAT. However, LSAT has features which are not required for Dezyne code generation. Hence, a textual Domain Specific Language is developed which acts as an intermediary towards Dezyne code generation from activity models. In the Activity DSL, FMS specifications can be described using the features from the LSAT. Next, the semantics of the activities is represented in terms of Gantt charts to explain what it means to execute an activity. Then modeling concepts used in Dezyne are explained. A model in Dezyne is specified in terms interfaces, components and events. These events are mapped to the activity and actions contained within that activity. Next, transformation rules are defined to obtain the transformation from the activity model to the Dezyne code and an algorithm is developed. Using this algorithm, Dezyne code is generated from the activities. The correctness of the transformation is validated as well as the scalability of the results in terms of increasing state-space. Finally, the generated code is verified in the Dezyne environment for any errors.

2. How to handle exceptions in the activity framework?
   To handle exceptions in an FMS, various responses of the system are described in terms of varying degrees of criticality. The notion of an exception is introduced in an activity and its specification is incorporated in the Activity DSL in terms of criticality levels. Next, the semantics of an activity is explained in terms of Gantt charts for the low-level criticality and the high-level criticality (complete shut down of the system) cases. The exceptions are then modeled in Dezyne using failure events which are defined for the activity and its actions. The behavior that the Dezyne model must exhibit for various criticality levels is also defined. Once the behavior is known, the rules of transformation are defined and an algorithm is developed, which is added to the Transformation DSL for automated Dezyne code generation to handle exceptions in an activity model. In the end, generated code is verified in the Dezyne environment for any errors.

7.2 Future Work

The scope of this project is limited to defining resources, actions and activities, for defining the specifications of an FMS. This scope can be extended to include activity sequences and peripherals.

At the level of activity sequences, multiple activities are scheduled and deployed at the runtime. This necessitates that the resources are shared by multiple activities during execution. This dynamic sharing of resources is implemented using claims and releases in the activity framework. When a resource is claimed by an activity, it can execute its actions. However, while a resource is claimed, other activities must wait for the resource to be released in order to execute their actions. Also, in each activity a resource can be claimed and released only once. This leads to performance optimizations in terms of throughput and makespan. Respecting the semantics for the sharing of resources by multiple activities at the component level is a challenge. Models in Dezyne have a strictly layered architecture. In addition, the run-to-completion semantics of incoming events enforce that any sequence of incoming events is serialized.

The solution can be further extended to include peripherals. Peripherals are physical components that constitutes a resource. Modeling an FMS which defines specifications in terms of peripherals comes with its own sets of challenges. The component level design must ensure that all actions mapped to the same peripheral must be sequentially ordered to avoid self-concurrency. In addition, different peripherals within a resource must be allowed to execute actions simultaneously.

The activities with two or more instances of the same action type can be defined in the Activity DSL. However, the systematic translation for such cases is not included in the Dezyne code generation. In future, the work can be extended to handle activities with multiple instances of the same action type in Dezyne by changing states such that each state handles only one instance of that action.

Further, the solution can be extended to handle exceptions of medium-level and high-level criticality (error state case). For this, the activity framework must be extended to include a mechanism for exception handling.

Considering the state-space explosion problem with addition of each action in an activity, there are scalability challenges in the verification process of the generated Dezyne code. Therefore, the verification process must be raised from the software component level to the functional level specifications.

Bibliography

[1] Edward Ashford Lee and Sanjit A. Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach, Second Edition, MIT Press*. 2017. 1

[2] Jim Browne, Didier Dubois, K. Rathmill, Suresh Sethi, and Kathryn Stecke. Classification of flexible manufacturing systems. *The FMS Magazine*, 2, 01 1984. 1

[3] Twan Basten, João Bastos, Róbinson Medina, Bram van der Sanden, Marc C. W. Geilen, Dip Goswami, Michel A. Reniers, Sander Stuijk, and Jeroen P. M. Voeten. *Scenarios in the Design of Flexible Manufacturing Systems*, pages 181–224. Springer International Publishing, Cham, 2020. 2, 8

[4] L.J. van der Sanden. *Performance analysis and optimization of supervisory controllers*. PhD thesis, Department of Electrical Engineering, Technische Universiteit Eindhoven, 11 2018. 2, 8, 9

[5] L.J. van der Sanden, J.P. Nogueira Bastos, J.P.M. Voeten, M.C.W. Geilen, M.A. Reniers, T. Basten, J. Jacobs, and R.R.H. Schiffelers. Compositional specification of functionality and timing of manufacturing systems. In *Proceedings of the 2016 Forum on specification and Design Languages, FDL 2016, Bremen, Germany*, 09 2016. 2, 8, 14

[6] ESI. Logistics-Specification and Analysis Tool (L-SAT). URL: `https://www.esi.nl/solutions/l-sat/`. 2

[7] Eindhoven University of Technology. CIF 3. URL: `http://cif.se.wtb.tue.nl/`. 3

[8] Logistics Specification and Analysis Tool (L-SAT) User Guide. Technical report. URL: `https://www.esi.nl/solutions/l-sat/userguide/index.html`. 3, 11

[9] Verum. About dezyne. URL: `https://www.verum.com/dezyne/`. 4

[10] Rutger van Beusekom, Jan Friso Groote, Paul F. Hoogendijk, Robert Howe, Wieger Wesselink, Rob Wieringa, and Tim A. C. Willemse. Formalising the Dezyne Modelling Language in mCRL2. In *FMICS-AVoCS*, pages 217–233, 2017. 4

[11] Saurav Paul and Ronald Wiericx. Smooth adoption of Verum's Dezyne to model software for a service tool. Technical report, ICT- Hitech Unit. URL: `https://ict.eu/wp-content/uploads/2018/03/ICT-GROUP_Casestudy-Dezyne_EN_cor.pdf`. 4

[12] FischerTechnik. Factory Simulation 24V-Education. URL: `https://www.fischertechnik.de/en/products/teaching/training-models/536634-edu-factory-simulation-24v-education`. 5

[13] Training Factory Industry 4.0 Activity booklet. Technical report, FischerTechnik. 5

[14] D.E. Nadales Agut, D.A. Beek, van, and J.E. Rooda. Syntax and semantics of the compositional interchange format for hybrid systems. *Journal of Logic and Algebraic Programming*, 82(1):1–52, 2013. 8

[15] Conrad Bock. Sysml and uml 2 support for activity modeling. *Systems Engineering*, 9:160 – 186, 06 2006. 8

[16] OMG Unified Modeling Language (Version 2.5.1). Technical report, 12 2017. 8

[17] Conrad Bock. Uml 2 activity and action models part 6: Structured activities. *Journal of Object Technology*, 4:43–66, 05 2005. 8

[18] Harald Störrle and J. Hausmann. Towards a formal semantics of uml 2.0 activities. In *Software Engineering*, 2005. 8

[19] Object constraint language. Technical report, 02 2014. 9

[20] Sunitha Ev and Philip Samuel. Object constraint language for code generation from activity models. *Information and Software Technology*, 103, 07 2018. 9

[21] Federico Ciccozzi, Antonio Cicchetti, and Mikael Sjödin. Full code generation from uml models for complex embedded systems. 11 2012. 9

[22] Action language for foundational uml. Technical report, 06 2017. 9

[23] Girish Managoli. What developers need to know about domain-specific languages. URL: `https://opensource.com/article/20/2/domain-specific-languages`. 13

[24] Verum. Dezyne features. URL: `https://verum.com/features-2/`. 16

[25] Verum. Dezyne reference manual. URL: `https://dezyne.org/dezyne/manual/dezyne.html`. 16

[26] Ken McMillan. Refinement verification. URL: `http://santos.cs.ksu.edu/smv-doc/tutorial/node8.html`. 31

[27] Verum. Dezyne features. URL: `https://dezyne.org/dezyne/manual/dezyne.html#Verifying-Models`. 31

Appendix A

Dezyne Code for Resources

A.1 Resources of activities Act_1, Act_2 and Act_3

The Dezyne code for interfaces of resource *r1* and *r2* is given below for different levels of criticality.

A.1.1 Critical level 0

1. Resource *r1*
   Interface

```
interface Ir1
{
   in void Start_a();
   out void Complete_a();

   in void Start_b();
   out void Complete_b();

   in void Start_c();
   out void Complete_c();

   in void Start_e();
   out void Complete_e();

   behaviour {
      enum Activity_states_t { IDLE, EXECUTE_a, EXECUTE_b, EXECUTE_c,
          EXECUTE_e };
      Activity_states_t state = Activity_states_t.IDLE;

      [state.IDLE] {
         on Start_a:
         {
            state = Activity_states_t.EXECUTE_a;
         }

         on Start_b:
         {
            state = Activity_states_t.EXECUTE_b;
         }

         on Start_c:
         {
            state = Activity_states_t.EXECUTE_c;
         }

         on Start_e:
         {
            state = Activity_states_t.EXECUTE_e;
```

```
            }
        }

        [state.EXECUTE_a]
        {
            on Start_a, Start_b, Start_c, Start_e: illegal;
            on inevitable:
            {
                Complete_a;
                state = Activity_states_t.IDLE;
            }
        }

        [state.EXECUTE_b]
        {
            on Start_a, Start_b, Start_c, Start_e: illegal;
            on inevitable:
            {
                Complete_b;
                state = Activity_states_t.IDLE;
            }
        }

        [state.EXECUTE_c]
        {
            on Start_a, Start_b, Start_c, Start_e: illegal;
            on inevitable:
            {
                Complete_c;
                state = Activity_states_t.IDLE;
            }
        }

        [state.EXECUTE_e]
        {
            on Start_a, Start_b, Start_c, Start_e: illegal;
            on inevitable:
            {
                Complete_e;
                state = Activity_states_t.IDLE;
            }
        }
    }
}
```

2. Resource *r2*
   Interface

```
interface Ir2
{
    in void Start_d();
    out void Complete_d();

    in void Start_f();
    out void Complete_f();

    behaviour {
        enum Activity_states_t { IDLE, EXECUTE_d, EXECUTE_f};
        Activity_states_t state = Activity_states_t.IDLE;

        [state.IDLE] {
            on Start_d:
            {
                state = Activity_states_t.EXECUTE_d;
            }

            on Start_f:
```

```
            {
                state = Activity_states_t.EXECUTE_f;
            }
        }

        [state.EXECUTE_d]
        {
            on Start_d, Start_f: illegal;
            on inevitable:
            {
                Complete_d;
                state = Activity_states_t.IDLE;
            }
        }

        [state.EXECUTE_f]
        {
            on Start_d, Start_f: illegal;
            on inevitable:
            {
                Complete_f;
                state = Activity_states_t.IDLE;
            }
        }
    }
}
```

A.1.2 Critical level 1

1. Resource *r1*
   Interface

```
interface Ir1
{
    in void Start_a();
    out void Complete_a();
    out void Complete_a_failed();

    in void Start_b();
    out void Complete_b();
    out void Complete_b_failed();

    in void Start_c();
    out void Complete_c();
    out void Complete_c_failed();

    in void Start_e();
    out void Complete_e();
    out void Complete_e_failed();

    behaviour {
        enum Activity_states_t { IDLE, EXECUTE_a, EXECUTE_b, EXECUTE_c,
             EXECUTE_e };
        Activity_states_t state = Activity_states_t.IDLE;

        [state.IDLE] {
            on Start_a:
            {
                state = Activity_states_t.EXECUTE_a;
            }

            on Start_b:
            {
                state = Activity_states_t.EXECUTE_b;
            }

```

```
            on Start_c:
            {
                state = Activity_states_t.EXECUTE_c;
            }

            on Start_e:
            {
                state = Activity_states_t.EXECUTE_e;
            }
        }

        [state.EXECUTE_a]
        {
            on Start_a, Start_b, Start_c, Start_e: illegal;
            on inevitable:
            {
                Complete_a;
                state = Activity_states_t.IDLE;
            }
            on inevitable:
            {
                Complete_a_failed;
                state = Activity_states_t.IDLE;
            }
        }

        [state.EXECUTE_b]
        {
            on Start_a, Start_b, Start_c, Start_e: illegal;
            on inevitable:
            {
                Complete_b;
                state = Activity_states_t.IDLE;
            }
            on inevitable:
            {
                Complete_b_failed;
                state = Activity_states_t.IDLE;
            }
        }

        [state.EXECUTE_c]
        {
            on Start_a, Start_b, Start_c, Start_e: illegal;
            on inevitable:
            {
                Complete_c;
                state = Activity_states_t.IDLE;
            }
            on inevitable:
            {
                Complete_c_failed;
                state = Activity_states_t.IDLE;
            }
        }

        [state.EXECUTE_e]
        {
            on Start_a, Start_b, Start_c, Start_e: illegal;
            on inevitable:
            {
                Complete_e;
                state = Activity_states_t.IDLE;
            }
            on inevitable:
            {
                Complete_e_failed;
```

```
            state = Activity_states_t.IDLE;
         }
      }
   }
}
```

2. Resource *r2*

   Interface

```
interface Ir2
{
   in void Start_d();
   out void Complete_d();
   out void Complete_d_failed();

   in void Start_f();
   out void Complete_f();
   out void Complete_f_failed();

   behaviour {
      enum Activity_states_t { IDLE, EXECUTE_d, EXECUTE_f};
      Activity_states_t state = Activity_states_t.IDLE;

      [state.IDLE] {
         on Start_d:
         {
            state = Activity_states_t.EXECUTE_d;
         }

         on Start_f:
         {
            state = Activity_states_t.EXECUTE_f;
         }
      }

      [state.EXECUTE_d]
      {
         on Start_d, Start_f: illegal;
         on inevitable:
         {
            Complete_d;
            state = Activity_states_t.IDLE;
         }
         on inevitable:
         {
            Complete_d_failed;
            state = Activity_states_t.IDLE;
         }
      }

      [state.EXECUTE_f]
      {
         on Start_d, Start_f: illegal;
         on inevitable:
         {
            Complete_f;
            state = Activity_states_t.IDLE;
         }
         on inevitable:
         {
            Complete_f_failed;
            state = Activity_states_t.IDLE;
         }
      }
   }
}
```

A.1.3 Critical level 2

1. Resource *r1*
   Interface

```
interface Ir1
{
   in void Start_a();
   out void Complete_a();
   out void Complete_a_failed();

   in void Start_b();
   out void Complete_b();
   out void Complete_b_failed();

   in void Start_c();
   out void Complete_c();
   out void Complete_c_failed();

   in void Start_e();
   out void Complete_e();
   out void Complete_e_failed();

   in void Reset();

   behaviour {
      enum Activity_states_t { IDLE, EXECUTE_a, EXECUTE_b, EXECUTE_c,
          EXECUTE_e };
      Activity_states_t state = Activity_states_t.IDLE;

      on Reset: {
         state = Activity_states_t.IDLE;
      }

      [state.IDLE] {
         on Start_a:
         {
            state = Activity_states_t.EXECUTE_a;
         }

         on Start_b:
         {
            state = Activity_states_t.EXECUTE_b;
         }

         on Start_c:
         {
            state = Activity_states_t.EXECUTE_c;
         }

         on Start_e:
         {
            state = Activity_states_t.EXECUTE_e;
         }
      }

      [state.EXECUTE_a]
      {
         on Start_a, Start_b, Start_c, Start_e: illegal;
         on inevitable:
         {
            Complete_a;
            state = Activity_states_t.IDLE;
         }
         on inevitable:
         {
            Complete_a_failed;
            state = Activity_states_t.IDLE;
```

```
            }
        }

        [state.EXECUTE_b]
        {
            on Start_a, Start_b, Start_c, Start_e: illegal;
            on inevitable:
            {
                Complete_b;
                state = Activity_states_t.IDLE;
            }
            on inevitable:
            {
                Complete_b_failed;
                state = Activity_states_t.IDLE;
            }
        }

        [state.EXECUTE_c]
        {
            on Start_a, Start_b, Start_c, Start_e: illegal;
            on inevitable:
            {
                Complete_c;
                state = Activity_states_t.IDLE;
            }
            on inevitable:
            {
                Complete_c_failed;
                state = Activity_states_t.IDLE;
            }
        }

        [state.EXECUTE_e]
        {
            on Start_a, Start_b, Start_c, Start_e: illegal;
            on inevitable:
            {
                Complete_e;
                state = Activity_states_t.IDLE;
            }
            on inevitable:
            {
                Complete_e_failed;
                state = Activity_states_t.IDLE;
            }
        }
    }
}
```

2. Resource *r2*
   Interface

```
interface Ir2
{
    in void Start_d();
    out void Complete_d();
    out void Complete_d_failed();

    in void Start_f();
    out void Complete_f();
    out void Complete_f_failed();

    in void Reset();

    behaviour {
        enum Activity_states_t { IDLE, EXECUTE_d, EXECUTE_f};
```

```
        Activity_states_t state = Activity_states_t.IDLE;

        on Reset: {
            state = Activity_states_t.IDLE;
        }

        [state.IDLE] {
            on Start_d:
            {
                state = Activity_states_t.EXECUTE_d;
            }

            on Start_f:
            {
                state = Activity_states_t.EXECUTE_f;
            }
        }

        [state.EXECUTE_d]
        {
            on Start_d, Start_f: illegal;
            on inevitable:
            {
                Complete_d;
                state = Activity_states_t.IDLE;
            }
            on inevitable:
            {
                Complete_d_failed;
                state = Activity_states_t.IDLE;
            }
        }

        [state.EXECUTE_f]
        {
            on Start_d, Start_f: illegal;
            on inevitable:
            {
                Complete_f;
                state = Activity_states_t.IDLE;
            }
            on inevitable:
            {
                Complete_f_failed;
                state = Activity_states_t.IDLE;
            }
        }
    }
}
```

A.2 Resources of Factory Four simulation model

Similarly, the Dezyne code for interfaces of resources in the Factory Four simulation model, that is, the *high-bay warehouse*, the *vacuum suction robot*, the *color sorter* and the *processing station* is given below for different levels of criticality.

A.2.1 Critical level 0

1. Resource *high-bay warehouse*
   Interface

```
import Definitions.dzn;

interface IWarehouse
{
   in void RequestToRetrieve(WidgetColorParam product);
   out void ReadyForPicking();

   in void Picked(WidgetColorParam product);
   out void ReadyForNext();

   in void RequestToStore(WidgetColorParam product);
   out void ReadyForReceiving();

   in void Placed(WidgetColorParam product);
   out void ReadyForNextAction();

   behaviour
   {
      enum WarehouseSRHState { READY, RETRIEVE_FULL_BOX, STORE_EMPTY_BOX,
          WAIT_WIDGET_PICKED, RETRIEVE_EMPTY_BOX, STORE_FULL_BOX,
          WAIT_WIDGET_PLACED };
      WarehouseSRHState state = WarehouseSRHState.READY;

      [state.READY]
      {
         on Picked, Placed : illegal;
         on RequestToRetrieve:
         {
            state = WarehouseSRHState.RETRIEVE_FULL_BOX;
         }
         on RequestToStore:
         {
            state = WarehouseSRHState.RETRIEVE_EMPTY_BOX;
         }
      }

      [state.RETRIEVE_FULL_BOX]
      {
         on  Picked, RequestToRetrieve, RequestToStore, Placed : illegal;
         on inevitable:
         {
            ReadyForPicking;
            state = WarehouseSRHState.WAIT_WIDGET_PICKED;
         }
      }

      [state.RETRIEVE_EMPTY_BOX]
      {
         on  Picked, RequestToRetrieve, RequestToStore, Placed : illegal;
         on inevitable:
         {
            ReadyForReceiving;
            state = WarehouseSRHState.WAIT_WIDGET_PLACED;
         }
      }

      [state.WAIT_WIDGET_PICKED]
      {
         on RequestToRetrieve, RequestToStore, Placed: illegal;
         on Picked:
         {
            state = WarehouseSRHState.STORE_EMPTY_BOX;
         }
      }

```

```
      [state. WAIT_WIDGET_PLACED]
      {
         on RequestToRetrieve, RequestToStore, Picked: illegal;
         on Placed:
         {
            state = WarehouseSRHState.STORE_FULL_BOX;
         }
      }

      [state.STORE_EMPTY_BOX]
      {
         on  Picked, RequestToRetrieve, RequestToStore, Placed : illegal;
         on inevitable:
         {
            ReadyForNext;
            state = WarehouseSRHState.READY;
         }
      }

      [state.STORE_FULL_BOX]
      {
         on  Picked, RequestToRetrieve, RequestToStore, Placed : illegal;
         on inevitable:
         {
            ReadyForNextAction;
            state = WarehouseSRHState.READY;
         }
      }
   }
}
```

2. Resource *vacuum suction robot*
   Interface

```
import Definitions.dzn;

interface IRobot
{
    in void StartTransferFromWarehouseToProcessing();
    out void PickedUpAtWarehouse();

    in void PlaceAtProcessing();
    out void DroppedAtProcessing();

   in void StartTransferFromColorSorterToWarehouse();
   out void PickedAtColorSorter();

   in void PlaceAtWarehouse();
   out void DroppedAtWarehouse();

    in void Homing();
    out void MoveCompleted();

    behaviour
    {
        enum RobotSRHState{ IDLE, PICKATWAREHOUSE, PICKED, PLACEATPROCESSINGSTATION, HOMING, HOMED, PICKATCOLORSORTER, PLACED, PLACEATWAREHOUSE };
        RobotSRHState state = RobotSRHState.IDLE;

        [state.IDLE]
        {
         on PlaceAtProcessing, Homing, PlaceAtWarehouse : illegal;
            on StartTransferFromWarehouseToProcessing:
            {
                state = RobotSRHState.PICKATWAREHOUSE;
            }
```

```
            on StartTransferFromColorSorterToWarehouse:
            {
                state = RobotSRHState.PICKATCOLORSORTER;
            }
        }

        [state.PICKATWAREHOUSE]
        {
         on StartTransferFromWarehouseToProcessing, Homing, PlaceAtProcessing,
              PlaceAtWarehouse, StartTransferFromColorSorterToWarehouse:
             illegal;
         on inevitable:
            {
                 PickedUpAtWarehouse;
                 state = RobotSRHState.PICKED;
            }
        }

      [state.PICKATCOLORSORTER]
      {
         on StartTransferFromWarehouseToProcessing, Homing, PlaceAtProcessing,
              PlaceAtWarehouse, StartTransferFromColorSorterToWarehouse:
             illegal;
         on inevitable:
         {
            PickedAtColorSorter;
            state = RobotSRHState.PLACED;
         }
      }

        [state.PICKED]
        {
         on StartTransferFromWarehouseToProcessing, Homing, PlaceAtWarehouse,
             StartTransferFromColorSorterToWarehouse: illegal;
            on PlaceAtProcessing:
            {
                 state = RobotSRHState.PLACEATPROCESSINGSTATION;
            }
        }

        [state.PLACED]
        {
         on StartTransferFromWarehouseToProcessing, Homing, PlaceAtProcessing,
              StartTransferFromColorSorterToWarehouse: illegal;
         on PlaceAtWarehouse:
         {
            state = RobotSRHState.PLACEATWAREHOUSE;
         }
        }

      [state.PLACEATPROCESSINGSTATION]
      {
         on StartTransferFromWarehouseToProcessing, Homing, PlaceAtProcessing,
              PlaceAtWarehouse, StartTransferFromColorSorterToWarehouse:
             illegal;
         on inevitable:
         {
            DroppedAtProcessing;
            state = RobotSRHState.HOMING;
         }
      }

      [state.PLACEATWAREHOUSE]
      {
         on StartTransferFromWarehouseToProcessing, Homing, PlaceAtProcessing,
              PlaceAtWarehouse, StartTransferFromColorSorterToWarehouse:
             illegal;
```

```
        on inevitable:
        {
            DroppedAtWarehouse;
            state = RobotSRHState.HOMING;
        }
    }

    [state.HOMING]
    {
        on StartTransferFromWarehouseToProcessing, PlaceAtProcessing,
            PlaceAtWarehouse, StartTransferFromColorSorterToWarehouse :
            illegal;
        on Homing:
        {
            state = RobotSRHState.HOMED;
        }
    }

      [state.HOMED]
      {
        on StartTransferFromWarehouseToProcessing, Homing, PlaceAtProcessing,
             PlaceAtWarehouse, StartTransferFromColorSorterToWarehouse:
            illegal;
        on inevitable:
            {
                MoveCompleted;
                 state = RobotSRHState.IDLE;
            }
      }
    }
}
```

3. Resource *color sorter*
   Interface

```
import Definitions.dzn;

interface IColorSorter
{
   in void SorterRequestToRetrieve(WidgetColorParam widgetColor);
   out void SorterReadyForPicking();

   behaviour
   {
      enum SorterState {IDLE, EXECUTE};
      SorterState state = SorterState.IDLE;

      [state.IDLE]
      {
         on SorterRequestToRetrieve:
         {
            state = SorterState.EXECUTE;
         }
      }

      [state.EXECUTE]
      {
         on SorterRequestToRetrieve: illegal;
         on inevitable:
         {
            SorterReadyForPicking;
            state = SorterState.IDLE;
         }
      }
   }
}
```

4. Resource *processing station*
   Interface

```
import Definitions.dzn;

interface IProcessingStation
{
   in void Start();
   out void ReadyForReceiving();

   behaviour
   {
      enum OvenState {IDLE, EXECUTE};
      OvenState state = OvenState.IDLE;

      [state.IDLE]
      {
         on Start:
         {
            state = OvenState.EXECUTE;
         }
      }

      [state.EXECUTE]
      {
         on Start: illegal;
         on inevitable:
         {
            ReadyForReceiving;
            state = OvenState.IDLE;
         }
      }
   }
}
```

A.2.2 Critical level 1

1. Resource *high-bay warehouse*
   Interface

```
import Definitions.dzn;

interface IWarehouse
{
   in void RequestToRetrieve(WidgetColorParam product);
   out void ReadyForPicking();
   out void ReadyForPickingFailed();

   in void Picked(WidgetColorParam product);
   out void ReadyForNext();
   out void ReadyForNextFailed();

   in void RequestToStore(WidgetColorParam product);
   out void ReadyForReceiving();
   out void ReadyForReceivingFailed();

   in void Placed(WidgetColorParam product);
   out void ReadyForNextAction();
   out void ReadyForNextActionFailed();

   behaviour
   {
      enum WarehouseSRHState { READY, RETRIEVE_FULL_BOX, STORE_EMPTY_BOX,
          WAIT_WIDGET_PICKED, RETRIEVE_EMPTY_BOX, STORE_FULL_BOX,
          WAIT_WIDGET_PLACED };
      WarehouseSRHState state = WarehouseSRHState.READY;
```

```

      [state.READY]
      {
         on Picked, Placed: illegal;
         on RequestToRetrieve:
         {
            state = WarehouseSRHState.RETRIEVE_FULL_BOX;
         }

         on RequestToStore:
         {
            state = WarehouseSRHState.RETRIEVE_EMPTY_BOX;
         }
      }

      [state.RETRIEVE_FULL_BOX]
      {
         on  Picked, Placed, RequestToRetrieve, RequestToStore: illegal;
         on inevitable:
         {
            ReadyForPicking;
            state = WarehouseSRHState.WAIT_WIDGET_PICKED;
         }
         on inevitable:
         {
            ReadyForPickingFailed;
            state = WarehouseSRHState.WAIT_WIDGET_PICKED;
         }
      }

      [state.RETRIEVE_EMPTY_BOX]
      {
         on  Picked, RequestToRetrieve, RequestToStore, Placed : illegal;
         on inevitable:
         {
            ReadyForReceiving;
            state = WarehouseSRHState.WAIT_WIDGET_PLACED;
         }
         on inevitable:
         {
            ReadyForReceivingFailed;
            state = WarehouseSRHState.WAIT_WIDGET_PLACED;
         }
      }

      [state.WAIT_WIDGET_PICKED]
      {
         on RequestToRetrieve, Placed, RequestToStore: illegal;
         on Picked:
         {
            state = WarehouseSRHState.STORE_EMPTY_BOX;
         }
      }

      [state. WAIT_WIDGET_PLACED]
      {
         on RequestToRetrieve, RequestToStore, Picked: illegal;
         on Placed:
         {
            state = WarehouseSRHState.STORE_FULL_BOX;
         }
      }

      [state.STORE_EMPTY_BOX]
      {
         on Picked, Placed, RequestToRetrieve, RequestToStore: illegal;
         on inevitable:
```

```
         {
            ReadyForNext;
            state = WarehouseSRHState.READY;
         }
         on inevitable:
         {
            ReadyForNextFailed;
            state = WarehouseSRHState.READY;
         }
      }

      [state.STORE_FULL_BOX]
      {
         on  Picked, RequestToRetrieve, RequestToStore, Placed : illegal;
         on inevitable:
         {
            ReadyForNextAction;
            state = WarehouseSRHState.READY;
         }
         on inevitable:
         {
            ReadyForNextActionFailed;
            state = WarehouseSRHState.READY;
         }
      }
   }
}
```

2. Resource *vacuum suction robot*
   Interface

```
import Definitions.dzn;

interface IRobot
{
    in void StartTransferFromWarehouseToProcessing();
    out void PickedUpAtWarehouse();
    out void PickedUpAtWarehouseFailed();

    in void PlaceAtProcessing();
    out void DroppedAtProcessing();
    out void DroppedAtProcessingFailed();

    in void StartTransferFromColorSorterToWarehouse();
   out void PickedAtColorSorter();
   out void PickedAtColorSorterFailed();

   in void PlaceAtWarehouse();
   out void DroppedAtWarehouse();
   out void DroppedAtWarehouseFailed();

   in void Homing();
    out void MoveCompleted();
    out void MoveCompletedFailed();

    behaviour
    {
        enum RobotSRHState{ IDLE, PICKATWAREHOUSE, PICKED,
            PLACEATPROCESSINGSTATION, HOMING, HOMED, PICKATCOLORSORTER, PLACED
            , PLACEATWAREHOUSE };
        RobotSRHState state = RobotSRHState.IDLE;

        [state.IDLE]
        {
         on PlaceAtProcessing, Homing, PlaceAtWarehouse: illegal;
            on StartTransferFromWarehouseToProcessing:
            {
```

```
                state = RobotSRHState.PICKATWAREHOUSE;
            }
            on StartTransferFromColorSorterToWarehouse:
            {
               state = RobotSRHState.PICKATCOLORSORTER;
            }
        }

        [state.PICKATWAREHOUSE]
        {
         on StartTransferFromWarehouseToProcessing,
             StartTransferFromColorSorterToWarehouse, Homing,
             PlaceAtProcessing, PlaceAtWarehouse: illegal;
         on inevitable:
            {
                PickedUpAtWarehouse;
                state = RobotSRHState.PICKED;
            }
            on inevitable:
            {
               PickedUpAtWarehouseFailed;
               state = RobotSRHState.PICKED;
            }
        }

      [state.PICKATCOLORSORTER]
      {
         on StartTransferFromWarehouseToProcessing,
             StartTransferFromColorSorterToWarehouse, Homing,
             PlaceAtProcessing, PlaceAtWarehouse,
             StartTransferFromColorSorterToWarehouse: illegal;
         on inevitable:
         {
            PickedAtColorSorter;
            state = RobotSRHState.PLACED;
         }
         on inevitable:
         {
            PickedAtColorSorterFailed;
            state = RobotSRHState.PLACED;
         }
      }

        [state.PICKED]
        {
         on StartTransferFromWarehouseToProcessing,
             StartTransferFromColorSorterToWarehouse, Homing, PlaceAtWarehouse
             : illegal;
            on PlaceAtProcessing:
            {
                state = RobotSRHState.PLACEATPROCESSINGSTATION;
            }
        }

        [state.PLACED]
        {
         on StartTransferFromWarehouseToProcessing,
             StartTransferFromColorSorterToWarehouse, Homing,
             PlaceAtProcessing, StartTransferFromColorSorterToWarehouse:
             illegal;
         on PlaceAtWarehouse:
         {
            state = RobotSRHState.PLACEATWAREHOUSE;
         }
        }

      [state.PLACEATPROCESSINGSTATION]
```

```
      {
         on StartTransferFromWarehouseToProcessing,
             StartTransferFromColorSorterToWarehouse, Homing,
             PlaceAtProcessing, PlaceAtWarehouse: illegal;
         on inevitable:
         {
            DroppedAtProcessing;
            state = RobotSRHState.HOMING;
         }
         on inevitable:
         {
            DroppedAtProcessingFailed;
            state = RobotSRHState.HOMING;
         }
      }

      [state.PLACEATWAREHOUSE]
      {
         on StartTransferFromWarehouseToProcessing,
             StartTransferFromColorSorterToWarehouse, Homing,
             PlaceAtProcessing, PlaceAtWarehouse,
             StartTransferFromColorSorterToWarehouse: illegal;
         on inevitable:
         {
            DroppedAtWarehouse;
            state = RobotSRHState.HOMING;
         }
         on inevitable:
         {
            DroppedAtWarehouseFailed;
            state = RobotSRHState.HOMING;
         }
      }

      [state.HOMING]
      {
         on StartTransferFromWarehouseToProcessing,
             StartTransferFromColorSorterToWarehouse, PlaceAtProcessing,
             PlaceAtWarehouse: illegal;
         on Homing:
         {
            state = RobotSRHState.HOMED;
         }
      }

        [state.HOMED]
        {
         on StartTransferFromWarehouseToProcessing,
             StartTransferFromColorSorterToWarehouse, PlaceAtProcessing,
             Homing, PlaceAtWarehouse: illegal;
         on inevitable:
            {
               MoveCompleted;
                state = RobotSRHState.IDLE;
            }
            on inevitable:
            {
               MoveCompletedFailed;
                state = RobotSRHState.IDLE;
            }
        }
    }
}
```

3. Resource *color sorter*
   Interface

```
import Definitions.dzn;

interface IColorSorter
{
   in void SorterRequestToRetrieve(WidgetColorParam widgetColor);
   out void SorterReadyForPicking();
   out void SorterReadyForPickingFailed();

   behaviour
   {
      enum SorterState {IDLE, EXECUTE};
      SorterState state = SorterState.IDLE;

      [state.IDLE]
      {
         on SorterRequestToRetrieve:
         {
            state = SorterState.EXECUTE;
         }
      }

      [state.EXECUTE]
      {
         on SorterRequestToRetrieve: illegal;
         on inevitable:
         {
            SorterReadyForPicking;
            state = SorterState.IDLE;
         }
         on inevitable:
         {
            SorterReadyForPickingFailed;
            state = SorterState.IDLE;
         }
      }
   }
}
```

4. Resource *processing station*
   Interface

```
import Definitions.dzn;

interface IProcessingStation
{
   in void Start();
   out void ReadyForReceiving();
   out void ReadyForReceivingFailed();

   behaviour
   {
      enum OvenState {IDLE, EXECUTE};
      OvenState state = OvenState.IDLE;

      [state.IDLE]
      {
         on Start:
         {
            state = OvenState.EXECUTE;
         }
      }

      [state.EXECUTE]
      {
```

```
            on Start: illegal;
            on inevitable:
            {
               ReadyForReceiving;
               state = OvenState.IDLE;
            }
            on inevitable:
            {
               ReadyForReceivingFailed;
               state = OvenState.IDLE;
            }
         }
      }
}
```

A.2.3 Critical level 2

1. Resource *high-bay warehouse*
   Interface

```
import Definitions.dzn;

interface IWarehouse
{
   in void RequestToRetrieve(WidgetColorParam product);
   out void ReadyForPicking();
   out void ReadyForPickingFailed();

   in void Picked(WidgetColorParam product);
   out void ReadyForNext();
   out void ReadyForNextFailed();

   in void RequestToStore(WidgetColorParam product);
   out void ReadyForReceiving();
   out void ReadyForReceivingFailed();

   in void Placed(WidgetColorParam product);
   out void ReadyForNextAction();
   out void ReadyForNextActionFailed();

   in void Reset();

   behaviour
   {
      enum WarehouseSRHState { READY, RETRIEVE_FULL_BOX, STORE_EMPTY_BOX,
          WAIT_WIDGET_PICKED, RETRIEVE_EMPTY_BOX, STORE_FULL_BOX,
          WAIT_WIDGET_PLACED };
      WarehouseSRHState state = WarehouseSRHState.READY;

      on Reset: {
         state = WarehouseSRHState.READY;
      }

      [state.READY]
      {
         on Picked, Placed: illegal;
         on RequestToRetrieve:
         {
            state = WarehouseSRHState.RETRIEVE_FULL_BOX;
         }

         on RequestToStore:
         {
            state = WarehouseSRHState.RETRIEVE_EMPTY_BOX;
         }
      }
```

```
    [state.RETRIEVE_FULL_BOX]
    {
       on  Picked, Placed, RequestToRetrieve, RequestToStore: illegal;
       on inevitable:
       {
          ReadyForPicking;
          state = WarehouseSRHState.WAIT_WIDGET_PICKED;
       }
       on inevitable:
       {
          ReadyForPickingFailed;
          state = WarehouseSRHState.WAIT_WIDGET_PICKED;
       }
    }

    [state.RETRIEVE_EMPTY_BOX]
    {
       on  Picked, RequestToRetrieve, RequestToStore, Placed : illegal;
       on inevitable:
       {
          ReadyForReceiving;
          state = WarehouseSRHState.WAIT_WIDGET_PLACED;
       }
       on inevitable:
       {
          ReadyForReceivingFailed;
          state = WarehouseSRHState.WAIT_WIDGET_PLACED;
       }
    }

    [state.WAIT_WIDGET_PICKED]
    {
       on RequestToRetrieve, Placed, RequestToStore: illegal;
       on Picked:
       {
          state = WarehouseSRHState.STORE_EMPTY_BOX;
       }
    }

    [state. WAIT_WIDGET_PLACED]
    {
       on RequestToRetrieve, RequestToStore, Picked: illegal;
       on Placed:
       {
          state = WarehouseSRHState.STORE_FULL_BOX;
       }
    }

    [state.STORE_EMPTY_BOX]
    {
       on Picked, Placed, RequestToRetrieve, RequestToStore: illegal;
       on inevitable:
       {
          ReadyForNext;
          state = WarehouseSRHState.READY;
       }
       on inevitable:
       {
          ReadyForNextFailed;
          state = WarehouseSRHState.READY;
       }
    }

    [state.STORE_FULL_BOX]
    {
       on  Picked, RequestToRetrieve, RequestToStore, Placed : illegal;
```

```
         on inevitable:
         {
            ReadyForNextAction;
            state = WarehouseSRHState.READY;
         }
         on inevitable:
         {
            ReadyForNextActionFailed;
            state = WarehouseSRHState.READY;
         }
      }
   }
}
```

2. Resource *vacuum suction robot*
   Interface

```
import Definitions.dzn;

interface IRobot
{
    in void StartTransferFromWarehouseToProcessing();
    out void PickedUpAtWarehouse();
    out void PickedUpAtWarehouseFailed();

    in void PlaceAtProcessing();
    out void DroppedAtProcessing();
    out void DroppedAtProcessingFailed();

    in void StartTransferFromColorSorterToWarehouse();
   out void PickedAtColorSorter();
   out void PickedAtColorSorterFailed();

   in void PlaceAtWarehouse();
   out void DroppedAtWarehouse();
   out void DroppedAtWarehouseFailed();

    in void Homing();
    out void MoveCompleted();
    out void MoveCompletedFailed();

    in void Reset();

    behaviour
    {
        enum RobotSRHState{ IDLE, PICKATWAREHOUSE, PICKED,
            PLACEATPROCESSINGSTATION, HOMING, HOMED, PICKATCOLORSORTER, PLACED
            , PLACEATWAREHOUSE };
        RobotSRHState state = RobotSRHState.IDLE;

        on Reset: {
         state = RobotSRHState.IDLE;
        }

        [state.IDLE]
        {
         on PlaceAtProcessing, Homing, PlaceAtWarehouse: illegal;
            on StartTransferFromWarehouseToProcessing:
            {
                state = RobotSRHState.PICKATWAREHOUSE;
            }
            on StartTransferFromColorSorterToWarehouse:
            {
               state = RobotSRHState.PICKATCOLORSORTER;
            }
        }

```

```
        [state.PICKATWAREHOUSE]
        {
         on StartTransferFromWarehouseToProcessing,
             StartTransferFromColorSorterToWarehouse, Homing,
             PlaceAtProcessing, PlaceAtWarehouse: illegal;
         on inevitable:
            {
                PickedUpAtWarehouse;
                state = RobotSRHState.PICKED;
            }
            on inevitable:
            {
               PickedUpAtWarehouseFailed;
               state = RobotSRHState.PICKED;
            }
        }

      [state.PICKATCOLORSORTER]
      {
         on StartTransferFromWarehouseToProcessing,
             StartTransferFromColorSorterToWarehouse, Homing,
             PlaceAtProcessing, PlaceAtWarehouse,
             StartTransferFromColorSorterToWarehouse: illegal;
         on inevitable:
         {
            PickedAtColorSorter;
            state = RobotSRHState.PLACED;
         }
         on inevitable:
         {
            PickedAtColorSorterFailed;
            state = RobotSRHState.PLACED;
         }
      }

        [state.PICKED]
        {
         on StartTransferFromWarehouseToProcessing,
             StartTransferFromColorSorterToWarehouse, Homing, PlaceAtWarehouse
             : illegal;
            on PlaceAtProcessing:
            {
                state = RobotSRHState.PLACEATPROCESSINGSTATION;
            }
        }

        [state.PLACED]
        {
         on StartTransferFromWarehouseToProcessing,
             StartTransferFromColorSorterToWarehouse, Homing,
             PlaceAtProcessing, StartTransferFromColorSorterToWarehouse:
             illegal;
         on PlaceAtWarehouse:
         {
            state = RobotSRHState.PLACEATWAREHOUSE;
         }
        }

      [state.PLACEATPROCESSINGSTATION]
      {
         on StartTransferFromWarehouseToProcessing,
             StartTransferFromColorSorterToWarehouse, Homing,
             PlaceAtProcessing, PlaceAtWarehouse: illegal;
         on inevitable:
         {
            DroppedAtProcessing;
            state = RobotSRHState.HOMING;
```

```
            }
            on inevitable:
            {
               DroppedAtProcessingFailed;
               state = RobotSRHState.HOMING;
            }
         }

         [state.PLACEATWAREHOUSE]
         {
            on StartTransferFromWarehouseToProcessing,
                StartTransferFromColorSorterToWarehouse, Homing,
                PlaceAtProcessing, PlaceAtWarehouse,
                StartTransferFromColorSorterToWarehouse: illegal;
            on inevitable:
            {
               DroppedAtWarehouse;
               state = RobotSRHState.HOMING;
            }
            on inevitable:
            {
               DroppedAtWarehouseFailed;
               state = RobotSRHState.HOMING;
            }
         }

         [state.HOMING]
         {
            on StartTransferFromWarehouseToProcessing,
                StartTransferFromColorSorterToWarehouse, PlaceAtProcessing,
                PlaceAtWarehouse: illegal;
            on Homing:
            {
               state = RobotSRHState.HOMED;
            }
         }

           [state.HOMED]
           {
            on StartTransferFromWarehouseToProcessing,
                StartTransferFromColorSorterToWarehouse, PlaceAtProcessing,
                Homing, PlaceAtWarehouse: illegal;
            on inevitable:
               {
                  MoveCompleted;
                   state = RobotSRHState.IDLE;
               }
               on inevitable:
               {
                  MoveCompletedFailed;
                   state = RobotSRHState.IDLE;
               }
           }
      }
}
```

3. Resource *color sorter*
   Interface

```
import Definitions.dzn;

interface IColorSorter
{
   in void SorterRequestToRetrieve(WidgetColorParam widgetColor);
   out void SorterReadyForPicking();
   out void SorterReadyForPickingFailed();

```

```
    in void Reset();

    behaviour
    {
        enum SorterState {IDLE, EXECUTE};
        SorterState state = SorterState.IDLE;

        on Reset : {
            state = SorterState.IDLE;
        }

        [state.IDLE]
        {
            on SorterRequestToRetrieve:
            {
                state = SorterState.EXECUTE;
            }
        }

        [state.EXECUTE]
        {
            on SorterRequestToRetrieve: illegal;
            on inevitable:
            {
                SorterReadyForPicking;
                state = SorterState.IDLE;
            }
            on inevitable:
            {
                SorterReadyForPickingFailed;
                state = SorterState.IDLE;
            }
        }
    }
}
```

4. Resource *processing station*
   Interface

```
import Definitions.dzn;

interface IProcessingStation
{
    in void Start();
    out void ReadyForReceiving();
    out void ReadyForReceivingFailed();

    in void Reset();

    behaviour
    {
        enum OvenState {IDLE, EXECUTE};
        OvenState state = OvenState.IDLE;

        on Reset : {
            state = OvenState.IDLE;
        }

        [state.IDLE]
        {
            on Start:
            {
                state = OvenState.EXECUTE;
            }
        }

        [state.EXECUTE]
```

```
      {
         on Start: illegal;
         on inevitable:
         {
            ReadyForReceiving;
            state = OvenState.IDLE;
         }
         on inevitable:
         {
            ReadyForReceivingFailed;
            state = OvenState.IDLE;
         }
      }
   }
}
```