Bachelor Informatica

Towards the Scalability of Detecting and Correcting Incompatible Service Interfaces

Mo Diallo

June 23, 2020

Supervisor(s): Benny Åkesson, Jack Sleuters

Abstract

In cyber-physical systems with long lifetimes, updating components is a costly process. Service-oriented software architectures are used to introduce flexibility into a system. This allows for modification of the underlying component, which implements a particular version of an interface, causing the system to evolve. Service interfaces thus create an abstraction layer over the components. With this abstraction layer, systems can be made up of various components that may be replaced or modified, provided they implement the same version of the interface. This abstraction breaks when service interfaces need to be modified, meaning that all components implementing that interface may fail.

The Dynamics project proposes a methodology for detecting and correcting incompatibilities within service interfaces post-modification. The scalability of the methodology presented in Dynamics has not been evaluated. They do not have sufficient existing interfaces to use for evaluations. This thesis aims to provide a starting point for the scalability evaluation of the proposed methodology. This will be done by generating synthetic interfaces of various complexity. We make two contributions in this thesis. First, we define the notion of complexity as inputs, outputs and non-determinism. Furthermore, the relation between these parameters is studied. Second, the methodology for a ComMA interface generator using user-supplied complexity parameters is introduced.

Acknowledgements

I would like to express my gratitude to the ESI team at TNO. The time they spent on answering questions and providing their insights was extremely valuable.
I would furthermore like to thank my supervisor B. Åkesson and J. Sleuters. Their overall help in meeting with me to discuss the contents of thesis and providing me guidance during writing were vital.

Contents

CHAPTER 1

Introduction

Technological needs are ever changing. Whether that is due to regulation, business needs, or technological advancements; systems will regularly need to be updated to comply with these new requirements. Within the context of these ever changing systems, issues unfold, such as in cyber-physical systems. In cyber-physical systems with long lifetimes, it is infeasible to continuously update individual components to adhere to new requirements or technological advancements.

One method of facilitating component updates is through the implementation of service-oriented software architectures. By using service interfaces, an abstraction layer over the service's components is created. This abstraction layer makes it possible for components of the system to be replaced by other components implementing the same interface. If this component implements the same version of the interface, even if the implementation of the component or the underlying technology differ, it will remain compatible with the system.

Utilizing services interfaces makes a system resistant to changes within the individual component. This highlights one issue, the service interface itself may eventually need to be updated. Figure 1.1 shows an example of such a scenario. Assuming all implemented interfaces were previously version 1, the example shows that the server and one component have been manually updated to version 2. The three other components implement the original version of the interface. This shows that when a service interface is updated, all server and client components implementing the old interface may fail. Manually updating these components is expensive and time consuming.

In collaboration with Thales, ESI (TNO) has developed an initial approach to detect incompatibilities between different versions of interfaces. Furthermore, their approach attempts to correct incompatibilities through adapter generation. The methodology is presented in the Dynamics [1] project. The proposed solution, which introduces a five-step methodology for the detection and correction of incompatibilities, requires a scalability analysis of this approach in the context of the required use case. The work conducted in this thesis provides a starting point towards evaluating the scalability of this approach. This will be done by generating complex interfaces. The findings from this thesis may be used in future work to establish the boundaries of the proposed methodology.

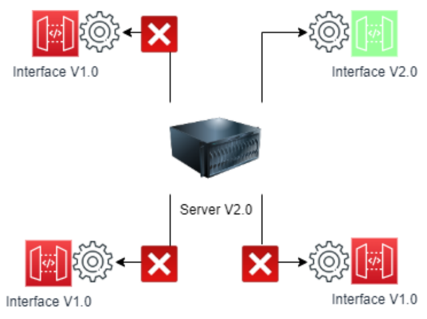


Figure 1.1: Update to service interface requires manually updating components.

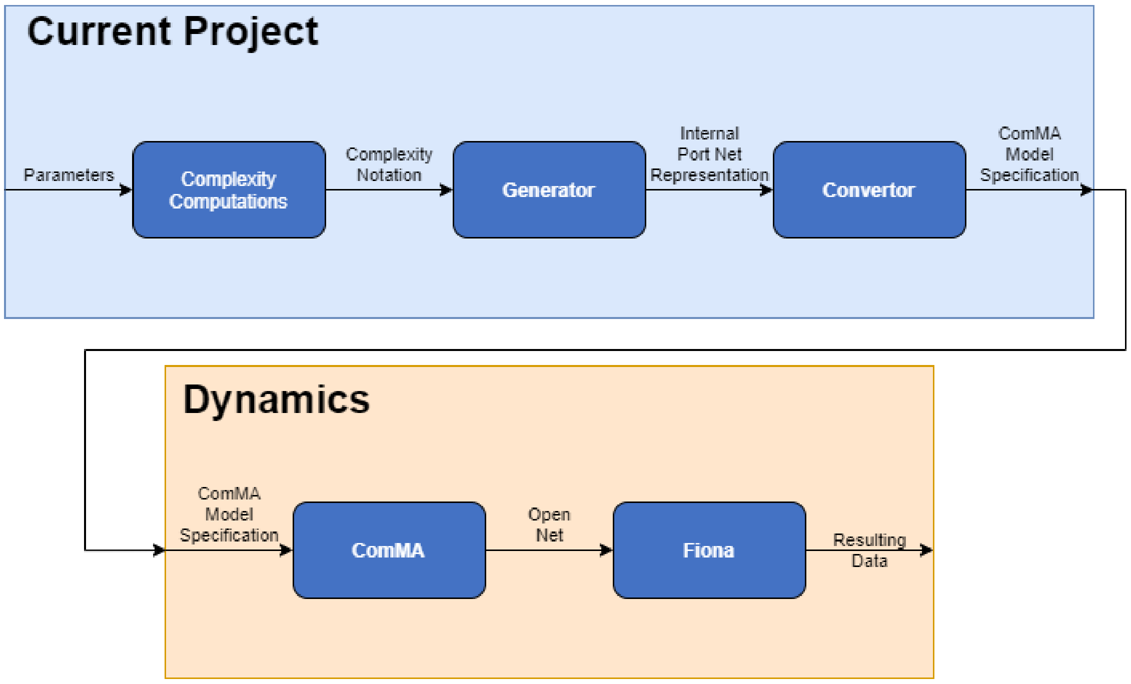


Figure 1.2: Context of thesis within the Dynamics project.

This thesis consists of three contributions, Figure 1.2 shows our contributions within the greater context.

1. We introduce and define the concept of complexity in synthetically generated interfaces as the number of inputs, outputs and the amount of non-determinism. The relation between the parameters is evaluated theoretically and experimentally.

2. A methodology for the synthetic generation of portnets, a subset of Petri Nets, using user-defined complexity parameters is presented. This utilizes previous research [2] on refinement rules for portnets. The application and properties of these rules is studied to devise a methodology for applying them in a synthetic generational context.

3. The methodology for portnet generation is extended by introducing a converter to a ComMA interface specification.

The rest of this thesis is structured as follows. Chapter 3 introduces the background this thesis builds upon. Chapter 4 introduces the methodology for generating synthetic interfaces of various complexity. This is done by defining the complexity parameters and introducing the generation scheme. In the following Chapter 5, the conversion process of a generated portnet to a ComMA interface is described. The generated methodology is evaluated in Chapter 6 by performing experiments and elaborating on the results. Chapter 7 will elaborate on the conclusions of this thesis.

CHAPTER 2

Related Work

There are a multitude of studies that focus on the synthetic generation of models for various types of analysis. "Task Graphs for Free" [6] introduces the generation of task graphs represented as directed acyclic graphs. The problem solved in the paper aims to assist researchers in re-creating examples by ensuring that there is a one-to-one relation when providing the same parameters to their proposed generator.
"SDF For Free" [14] provides a similar approach. User-supplied parameters are provided to a generator to create Synchronous DataFlow Graphs.
Both these papers align with the intent of this thesis: synthetically generating a model to perform various evaluations. In the context of our work, this is to facilitate scalability evaluations. Where they differ is that they do not directly specify which parameters influence the result of the generator. It is not intuitive how the chosen parameters correspond to the output of the generation, and thus the usage of their methodology already requires a substantial amount of knowledge in the topic.

In contrast to the previous papers, this generator aims to provide users with intuitive parameters that directly affect what is generated. Furthermore, these parameters are also directly responsible for adding complexity within the generated interface, allowing one to argue between the complexity of two different interfaces utilizing only the given parameters.
Furthermore, the result of the generator that we devise will be completely random. The same parameter values may result in completely different structures.

CHAPTER 3

Project Background

The goal of our work is to generate part of the required tooling for complexity analysis of the work conducted in Dynamics [1]. We generate the original ComMA model, as seen in Figure 3.1. This chapter introduces the various background concepts this project builds upon. In Section 3.1 the Dynamics project, for which this work is conducted, is introduced. In Section 3.2 we present one of the key concepts of the Dynamics project, ComMA interface specification. This is further extended in Section 3.3 by the introduction of Petri Nets. Finally, the focus of this generator, portnets and their refinement rules, are presented in Section 3.4.

3.1 Dynamics

The Dynamics [1] project provides a solution to the problem of "complex systems that need to continuously evolve".
The contributions made in the Dynamics project result in the five-step methodology for the detection and correction of service interface incompatibilities.

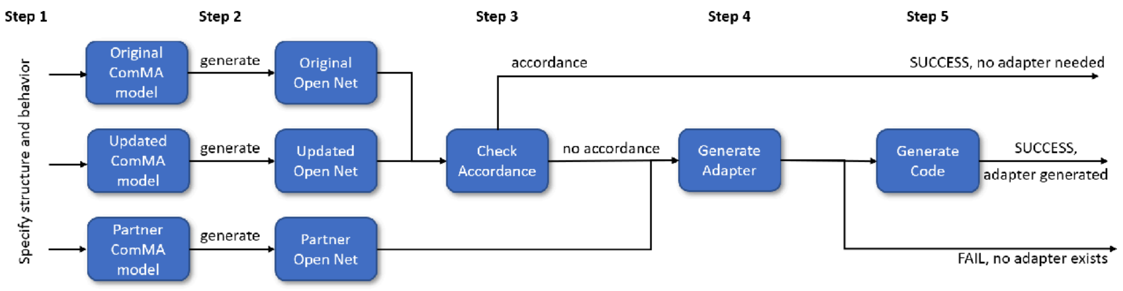


Figure 3.1: Five-step methodology proposed by ESI (TNO)

The five-step methodology for detecting and correcting incompatible service interfaces is shown in figure 3.1. A brief summarized version of the explanation of each step is as follows:

1. The first step within the proposed solution uses research done previously with Philips [8]. The essence of the research defines modeling interface behavior and structure using ComMA (Component Modeling and Analysis) framework.

2. In the second step, from the earlier created ComMA models an Open Net (Petri Net) [11] is generated.

3. From the third step, detection and correction happens. This starts with accordance checking [10] using the tool Fiona [12]. If these two interfaces are in accordance, all clients of the old interface are compatible with the updated interface. Two different interfaces do not have to be incompatible. For example, one could have two different interfaces where

the updated one includes an optional feature. These two interfaces would then still be in accordance.
In contrast, if for example a new mandatory signal is added, they will not be in accordance and clients will no longer work properly.

4. If during step three the two interfaces were found not to be in accordance, what remains is the generation of a corresponding adapter. This is described in the proposed solution as an approach based on controller synthesis [7].

5. As final step, if an adapter can be generated, the corresponding C++ code for the INAET-ICS architecture [4] is generated.

3.2 Component Modeling and Analysis (ComMA)

ComMA [8] is an approach utilized for interface specification. The problem statement described in the paper discusses that interfaces are often only described by their signature. They go on to state that timing and data constraints usually stay implicit. The paper introduces an approach that avoids these problems by formalizing interface specifications.

ComMA is described as a state machine-based DSL. The framework supports interface specification through providing an interface structure, interface behavior and data/timing constraints. Furthermore, ComMA provides the ability to monitor the interface behavior during execution. This way it can be checked that a system conforms to the interface specification.

In our work we will focus on how interface behavior is specified with ComMA.

Listing 3.1: Simple ComMA light switch example

```

machine LightSwitch {
    initial state OFF {
        transition trigger: LSOn
            next state: ACK_ON
    }

    state ACK_ON {
        transition
            do:
                LTAck
            next state: ON
    }

    state ON {
        transition trigger: LTOff
            next state: OFF

    }
}
```

ComMA represents interface behavior as a traditional state machine. Listing 3.1 shows a simple example of a light switch interface. Of importance within the context of our work are the basics: signals, notifications, transitions and states. Signals are client to server communication and notifications are server to client communication.

The provided example assumes a server-sided viewpoint for a remote-controlled light switch.

When the light switch is in the ***OFF*** state, upon receiving the signal ***LSOn***, it goes into the ***ACK_ON*** state. Within this state an acknowledgement follows by sending the **LTAck** notification. The server will after these actions be in the ***ON*** state. Eventually, the client can choose to switch the light off again by sending **LTOff**, which results in the server going back to the ***OFF*** state. Figure 3.2 shows the corresponding communicating finite-state machine [5], "?" denotes an incoming signal, while "!" denotes an outgoing notification.

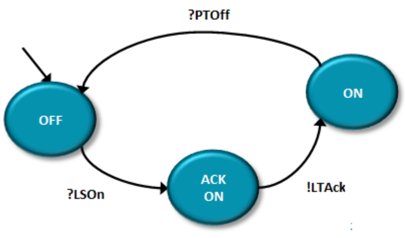


Figure 3.2: Communicating finite-state machine representation of a light switch.

3.3 Petri nets

Petri Nets, or place/transitions nets, are a particular kind of directed graphs utilized for the modelling of distributed systems [13]. Using the terminology of graphs, a Petri Net can be considered as having two different types of nodes, places and transitions. The directed edges within graphs are referred to as arcs. How this can be visualized is shown in Figure 3.3.

When modelling systems using Petri Nets, places can be considered as conditions. The truth value of these conditions are considered markings. The marking of a place p, M_p, is a natural number representing the tokens in that place. Referring back to the modeling example, a place with three markings can be said to have fulfilled three of its pre-conditions.

Transitions within Petri Nets can be considered as events. These transitions have a number of input places and output places, connected by arcs between them. The *firing* (activation) of a transition is dependent on it being *enabled*. An enabled transition requires all connected input places to contain a token. The example in Figure 3.3 shows tokens only in the places labeled as "Input" and "Initial Place". Therefore, the first transition is enabled, while the second transition is disabled. When a transition is fired, it consumes one token from all of its input places and produces a token in all the output places. Therefore, in the given example the firing of transition 1 would produce a token in place $P2$.

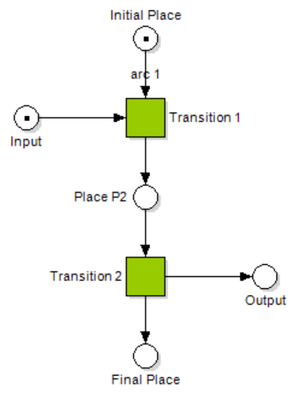


Figure 3.3: Terminology of Petri Nets

Petri Nets are used within the Dynamics [1] project to detect incompatibilities. There are three primary reasons stated within the paper. The first reason is due to it being possible to translate interfaces specified in ComMA to a Petri Net. The second reason is due to Petri Nets supporting the modeling of synchronous and asynchronous communication. The final reason is due to already existing analysis methods for detecting and correction of incompatibilities [16].

3.4 Portnet Refinement Rules

This background section covers the refinement rules introduced in [2]. We introduce the concept of portnets, a special case of workflow nets [15] [17]. In the context of this work, it is not integral to know what workflow nets are, hence this explanation will be omitted.

There are several properties that valid portnets must adhere to. First, we introduce the concept of markings: each portnet consists of a single initial marking $P_{initial}$ for which goes that no arc in the net directs to it. Therefore, given an arc as the tuple $< a_{origin}, a_{destination} >$ then $\{a : a_{destination} == P_{initial}\} = \emptyset$. Furthermore, a portnet must have a final marking, meaning a place with no outgoing arcs, formally: $\{a : a_{origin} == P_{initial}\} = \emptyset$. We now come towards *weak termination*. In [2] the concept of weak termination is introduced. In Figure 3.4 a valid portnet is given that weakly terminates. The paper states the definition as “The weak termination property states that in each reachable state of the system, the system always has the possibility to reach a final state”.
Using the portnet, that means that from any place, the system will reach the final marking. Consequently, all tokens within the portnet will be in the final marking.

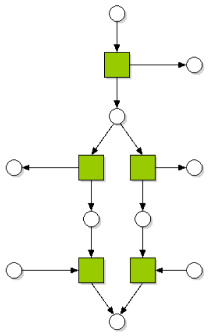


Figure 3.4: Valid weakly-terminating portnet

Another concern within portnets is the choice property. The property states that for any given place, all transitions directly following that place must have the same direction of communication. Figure 3.5 shows an example in which this does not happen.

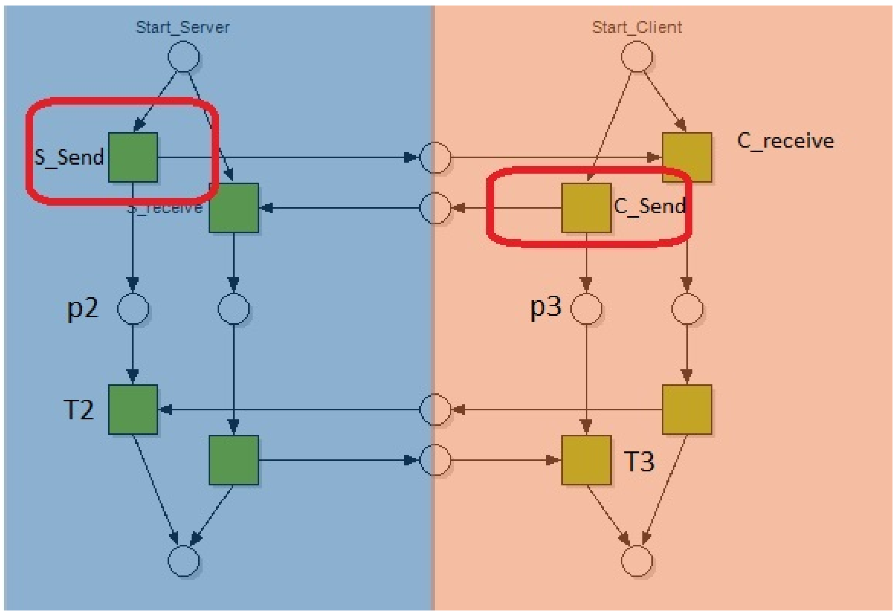


Figure 3.5: Mirrored client example of choice property violation.

If both the client and server decide to send, in C_{Send} and S_{Send} respectively, a deadlock occurs. The transitions directly following, $T2$ and $T3$, are receive for both the client and server. Consequently, neither the client nor server can ever reach the final marking.

To continue, portnets must also satisfy the leg property. The leg property refers to the conditions each leg in the portnet must satisfy. A leg is defined as being a path within the portnet that deviates from another path by a split and is concluded by a join. Furthermore, the existence of a single path from the initial place to the final marking is also considered as being a separate leg. The leg property requires each path to have at least two transitions in alternating direction: an input and output. Therefore, each leg must have one transition that is an input and one that is an output.
The example in Figure 3.6 shows a portnet that violates the leg property.

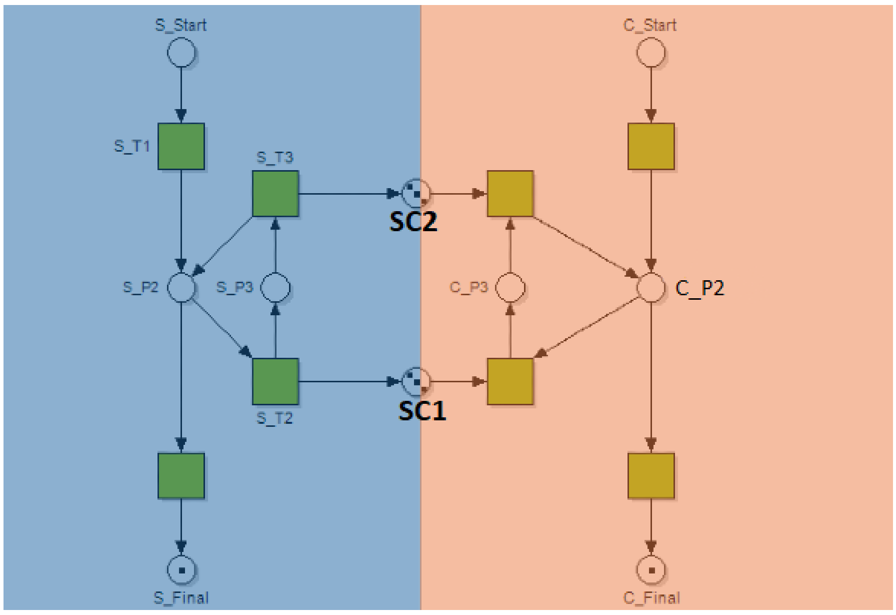


Figure 3.6: Mirrored client example of leg property violation.

The figure shows that the portnet is not weakly terminating: there are still remaining tokens even though the final places have been reached for both the client and server. To understand how this occurs we refer to the server side of the example. In the beginning, a token exists at S_Start which is consumed to generate one token in S_P2. From there it is possible for the transition S_T2 to be triggered as it contains a token in its only input place, S_P2. This will generate a token in S_P3 and $SC1$. This is where the issue starts, when S_T3 is triggered it consumes the token in S_P3 and produces a token in S_P2. This process may now repeat itself any number of times, causing the token count in $SC1$ to increase. Note how both the client and server can in C_P2 and S_P2 continue to the final places without ever having to consume the generated tokens. Therefore, this example does not weakly terminate: it contains leftover tokens upon reaching the final places.
The final property we introduce covers transitions. The property states that for a portnet to be valid, all of its transitions must connect to exactly one interface place and one input or output place.

3.4.1 Refinement Rules

The base of synthetic interface generation is formed around a construction method [2], which uses five different refinement rules that derive a Petri Net through continuous rule application. With exception of the fifth rule, each of these rules can be split into *base* rules and *modified* rules.

The base rules are the starting point for refinement, they operate upon a transition or place and generate a new structure within the portnet. The modified rules can be best described as altering the generated net to adhere to the validity properties of portnets, adding inputs/outputs and ensuring the leg and choice property are complied with.

This section studies the aforementioned rules, such that an appropriate way to apply them can be devised. These rules will be essential to creating valid portnets while adhering to user supplied complexity parameters.
There are five refinement rules that are to be considered and their explanation will be given in the following subsections.

R0: Default Place Refinement

The default rule, $R0$, forms the base of the refinement rules. As seen in Figure 3.7, the base rule converts a single place into a structure of $Place \rightarrow Transition \rightarrow Place$. The modified rules alter this construct by adding either a single input or output to the transition. It should be noted that the modified rules of $R0$ are the only rules that may generate a single input or output. All other refinement rules add an equal number of inputs and outputs.

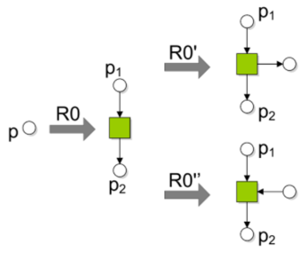


Figure 3.7: $R0$ adds a single place and transition, its modified rules generate one input and output

R1: Transition Refinement

The transition refinement rule, $R1$, provides an expansion of a transition, Figure 3.8 shows this. In expanding a transition, it adds one additional place and transition. An important characteristic that presents itself can be seen when the modified rules are applied. The transition expansion is considered as being part of a leg. As discussed earlier, the leg property states that within every leg, there must be at least two transitions with different directions of communication. The two modified rules satisfy this condition by adding one input and output to the two transitions. The only difference between the two modified rules is the order in which this is done.

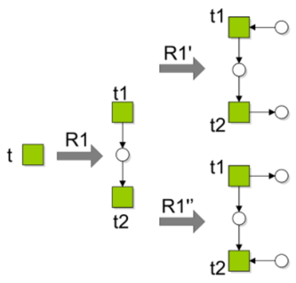


Figure 3.8: $R1$ adds a single place and two transitions, its modified rules generate one input and output.

R2: Non-deterministic Transition Refinement

The non-deterministic transition refinement rule, $R2$, is one of the rules that controls the non-determinism of the generated net. Like rule $R1$, it is a transition refinement rule. The difference between these rules lies in how they refine the transition. $R2$ duplicates a transition by adding another transition with the same start and end, as can be seen from Figure 3.9. The issue with the base rule of R2 is that it generates a structure which violates the leg property. To account for this, these legs need to be expanded, which can be seen as applying the base rule of $R1$ upon both generated transitions. When this is not done correctly an invalid portnet will be generated. This occurs when two different modified versions of $R1$ would be applied to add the inputs/outputs. This would violate the choice property, which requires all transitions directly following the place (having an incoming arc from it) to have communication in the same direction. The modified rules fix this issue by adding an additional place and transition in each leg. To preserve the choice property, the order of inputs and outputs in each leg is the same.

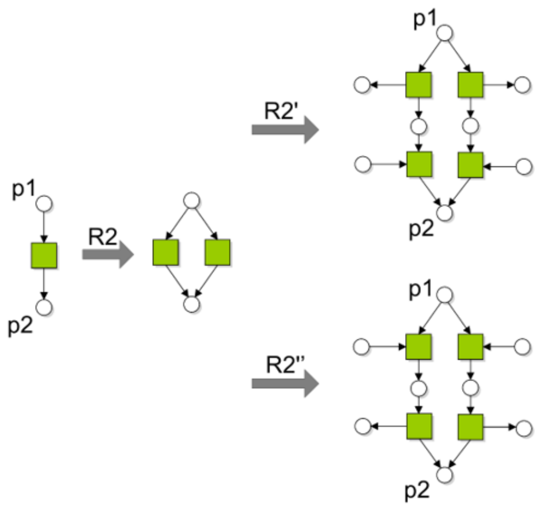


Figure 3.9: $R2$ adds one place and transition. The modified rules add two transitions and places. Furthermore, the modified rules add two inputs and outputs.

R3: Cyclic Place Refinement

Thus far, all rules have resulted in a structure where places and transitions are sequential. There are no loops and consequently there is no way to go back to any previous place within the net. Rule $R3$ (Figure 3.10) changes that by introducing a cyclic structure from a place refinement.

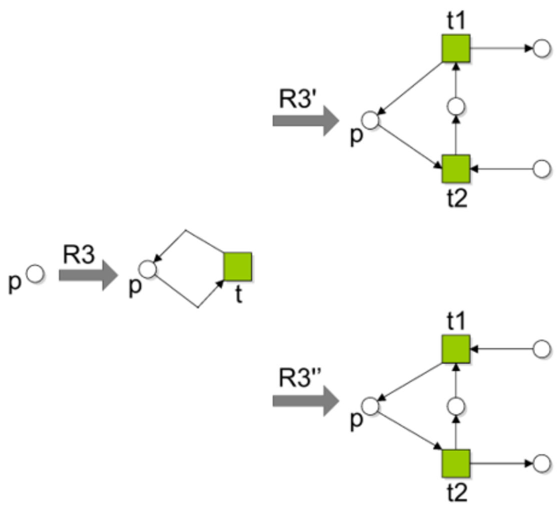


Figure 3.10: $R3$ adds one transition. The modified rules add an additional place and transition. Furthermore, one input and output are added.

The base rule of $R3$ introduces a structure that violates the leg property. The resulting $P1- > Transition- > P1$ structure does not have at least two transitions which have different directions of communication. Thus, to account for this property the modified rules add an extra transition and ensure both transitions have alternating directions of communication.
An important thing to note is that $R3$ similarly to $R2$ introduces non-deterministic behavior. This is dependent on the place which it is refined upon. Figure 3.11 shows an example of how $R3$ may be applied upon P2 to introduce non-determinism. When $R3$ is applied upon a non-deterministic place, the already existing outgoing arc becomes non-deterministic. Thus, the application of $R3$ would introduce 2 non-deterministic arcs to the net. However, if another application of $R3$ follows upon that same place ($P2$), this would only introduce one non-deterministic arc into the net.

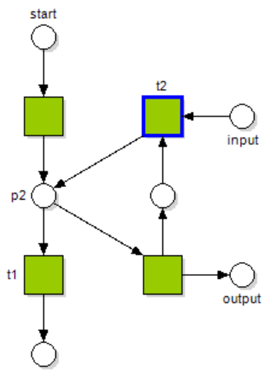


Figure 3.11: Introducing non-deterministic behavior by applying rule $R3$ upon place $P2$.

R4: Concurrent Place/Transition Refinement

Rule four provides a special case within construction. Unlike the other rules thus far, $R4$ is not extended by additional modified rules introducing inputs or outputs. As seen in Figure 3.12, this rule only adds add concurrency to an initial construct of $Transition \rightarrow Place \rightarrow Transition$.

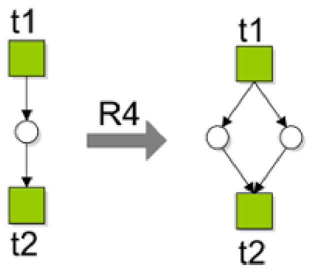


Figure 3.12: *R*3 adds one transition. The modified rules add an additional place and transition. Furthermore, one input and output are added.

In the context of the Dynamics project, this rule is not supported. This rule introduces concurrency, which means the resulting structure cannot be represented as a state machine. This construct is thus not a portnet and is also not supported in ComMA, as ComMA interfaces are state machines [8]. Considering the project pipeline as per Figure 3.1, this rule does not introduce any constructs that could be a result of the proposed solution. For this reason, the generator will not include this rule by default.

CHAPTER 4

Synthetic Interface Generation

This chapter discusses the synthetic generation of interfaces of varying complexity. This chapter is split into six sections. Section 4.1 will discuss the essential requirements the generator must satisfy. Section 4.2 introduces and defines the complexity parameters. The allowed ruleset is presented in Section 4.3. This allowed ruleset is used in Section 4.5 and 4.4 to discuss parameterized generation and devise the generation scheme for portnets adhering to user supplied complexity parameters. This chapter concludes with Section 4.6 discussing the limitations of the portnet generator.

4.1 Requirements

The requirements imposed on the generator are motivated by the work conducted in Dynamics [1]. This is done by performing accordance checking, and adapter generation if necessary. The proposed solution by Dynamics solves the issues of having to update all components implementing an interface, post-modification of this interface.

The goal of this thesis is to develop a generator capable of creating interfaces that emulate the properties of those supplied to Dynamics. These generated interfaces can then be used for testing the solution and for conducting a scalability analysis of the methodology proposed by the Dynamics project.
There are various requirements that the generator must satisfy. Before making the requirements concrete, it should be made clear where in the Dynamics pipeline (Figure 3.1) the generator operates: this will define what the output of the generator is. Figures 4.2 and 4.1 show the two possible approaches to generation.

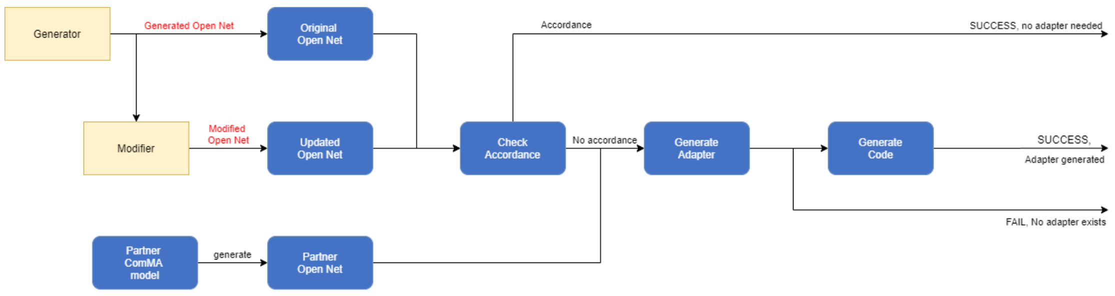


Figure 4.1: Option 1: generating Open (Petri) Nets.

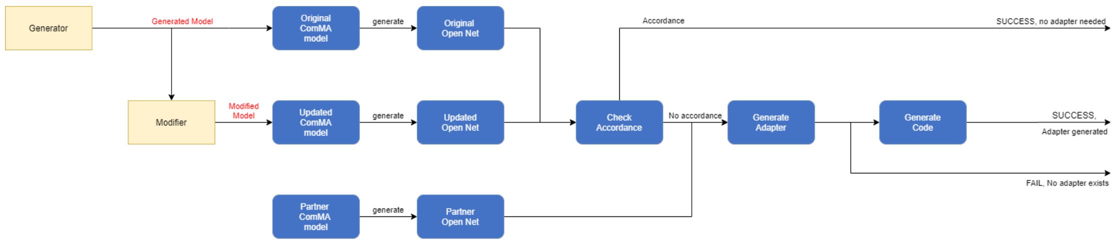


Figure 4.2: Option 2: generating ComMA models.

The first approach is to generate portnets. This is the input accepted by FIONA to enable accordance checking and adapter generation.
The second approach is to generate ComMA interfaces. The Dynamics project provides a converter, which would then translate this into a portnet. Subsequently, the portnet would be provided to FIONA as input.
The biggest shortcoming to the first approach is that it assumes the converter is fixed. If this were not the case, and the converter is later modified, resulting in an elimination or addition of possibilities, this thesis's generator would be outdated, and it could no longer accurately be used for scalability analysis.
As our goal is to synthetically generate interfaces, which can later be utilized for scalability analysis, the generator should solely consider interfaces that can be a result of Steps 1 and 2 in the proposed five-step methodology. If the first approach was chosen, we would need to evaluate how the converter makes a portnet out of a ComMA model. This so that the generator does not produce any interfaces that could not be created by the converter.
This gives the motivation behind the choice for the second approach: by generating ComMA models, one makes the generator more resistant to changes within the Dynamics project.
With the knowledge of what is to be generated, the requirements can now be formalized.

1. The first requirement is that a notion of complexity must be defined. The generator needs to be able to generate interfaces of various complexity. Henceforth, it is essential that what is considered as complex can also be passed to the generator as parameters. These parameters can be used during scalability analysis to get an understanding of which constructs within the generated portnet add the most complexity to either accordance checking and/or adapter generation.

2. The second requirement is that the generator must produce valid portnets. A converter should be included alongside the generator to convert the generated portnets into ComMA interface specification.

3. As scalability analysis requires two ComMA interfaces, an old and new interface, the generator should provide a format which allows further extensions of the generator to perform modifications upon the constructed interfaces.

4.2 Complexity Parameters

Synthetic interface generation concerns itself with generating interfaces of various complexity. Thus far, the notion of complexity has not been defined. This section will introduce three non-zero parameters the generator considers as adding complexity: inputs, outputs and the prevalence of non-determinism. Furthermore, this section will concern itself only with explaining and defining these parameters. Parameter constraints and how generation takes them into account is left for Section 4.5.

4.2.1 Number of Inputs and Outputs

Out of the three parameters, inputs and outputs are the most trivial. In the case of ComMA interfaces, inputs and outputs translate to signals and notifications, respectively. Defined in the requirements, the decision was made to generate portnets which are then converted to ComMA interfaces. As mentioned in Section 3.4, one of the requirements of portnets is that every transition is connected to one input or output. Thus, the number of inputs and outputs directly relate to the number of transitions in the net. Therefore, as transitions dictate the number of places present, indirectly the number of inputs and outputs specified are a way of controlling the size of the generated portnet. Parameters are specified as integers to the generator in the range $inputs, outputs > 0$.

4.2.2 Non-Determinism

The third controllable parameter during generation is that of non-determinism. Compared to inputs and outputs, non-determinism is of more interest, as the definition is open to various interpretations.

Definition

The definition of the prevalence of non-determinism is given as the percentage of arcs in the generated net which originate from a state with multiple outgoing arcs. Formally:

$$a =< a_{origin}, a_{destination} > \qquad \text{2-tuple representation of arc} \qquad (4.1)$$
$$A_p = \{a : a_{origin} == p\} \qquad \text{Arcs (a) originating from place P} \qquad (4.2)$$
$$N = \{p : |A_p| > 1\} \qquad \text{Non-deterministic places} \qquad (4.3)$$
$$X = \{a : a_{origin} \in N\} \qquad \text{Arcs with origin in non-deterministic place} \qquad (4.4)$$
$$prevalence = \frac{|X|}{\text{total arcs}} \qquad \text{Prevalence of non-determinism} \qquad (4.5)$$

What the equations show is that an arc is a 2-tuple where the elements are the originating and destination place, respectively. The set of arcs originating from a place, p, is denoted by A_p. A place within a portnet is said to belong to the set of non-deterministic places, N, if the cardinality of A_p is greater than one, meaning multiple outgoing arcs. Furthermore, all arcs that are non-deterministic have their origin as element of the set N.
The definition of the prevalence of non-determinism follows as the fraction of non-deterministic arcs outgoing from places in the entire portnet.

4.3 Allowed Ruleset

Thus far, this chapter gave the requirements that interface generation must adhere to. Furthermore, the complexity parameters have been introduced and defined. This section will build on the introduced topics to devise a generation scheme utilizing the portnet refinement rules from Section 3.4. The paper [2] introducing these rules introduces the concept of *weak termination*: "From each reachable state, the system has the option to eventually terminate."
In this section, these rules are used to generate the allowed ruleset: the set of rules which may be subsequently followed by another rule to generate valid portnets.
The allowed ruleset guarantees that using the construction rules in the way described retains the properties of portnets, as given in Section 3.4. Furthermore, facilitates the creation of a refinement scheme using the described complexity parameters introduced in Section 4.2.

4.3.1 Definition

The paper that describes the refinement rules does not directly provide information for adopting these rules in a generative way. Furthermore, the paper only concerns itself with the validity of the portnet itself. The notion of complexity does not appear within the paper. Therefore, as our generator needs consider the user-supplied parameters, we introduce a refinement scheme which extends these rules for usage in the context of this thesis. A state machine representation is given in Figure 4.3, this is used in the following subsections to elaborate on the rules concerning application of the construction rules. Final states are modified rules denoted by two circles; they indicate the end of a *refinement iteration*.

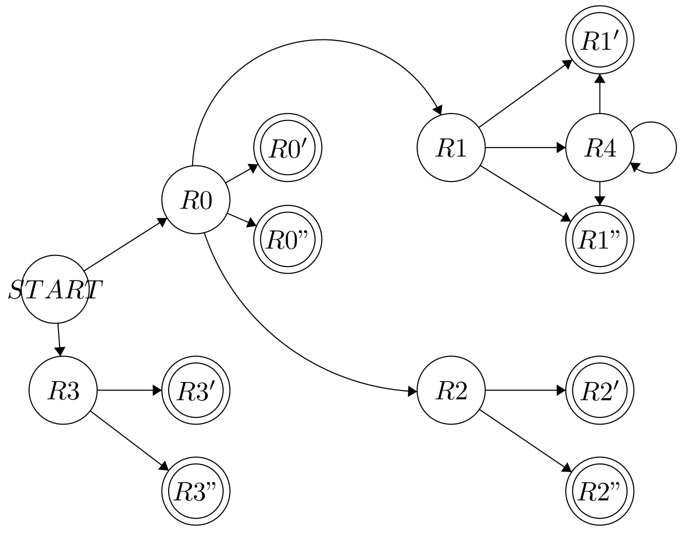


Figure 4.3: Ruleset as a state-machine showing the allowed order of rule applications.

Refinement Iteration

The application of the refinement rules is done within *refinement iterations*. A refinement iteration is a sequence of refinement rules applied on an existing place, while preserving portnet properties. To further elaborate, each refinement iteration starts with a single place. on this place, one may according to the allowed ruleset apply a single rule (R3 or R0). From this point forward, the allowed ruleset is in a new state depending on the rule applied. A refinement iteration proceeds by randomly selecting rules based on the rule that was previously applied. This continues until a final state is reached. This random selection is done while adhering to the selected parameters, and taking into account that the properties of a valid portnet must be retained after a full refinement iteration.
Once a final state is reached, the refinement iteration concludes, and the resulting structure is guaranteed to be a valid portnet. This portnet then provides the basis of subsequent refinement iterations. This continues until the stop conditions, the supplied parameter values, are reached.

4.3.2 Rules for Refinement Applications

Within refinements, the generator distinguishes between two *refinement patterns*: meaning a rule is either a place or transition refinement. A refinement iteration is composed of refinement rules that replace/expand the selected part of the portnet. Only base rules are considered as being part of a refinement pattern, meaning that only they refine a transition or place. Modification rules solely serve to ensure the result of a refinement iteration is a valid portnet. Therefore, when evaluating the refinement rules, the only rules that perform place refinement are the base rules $R0$ and $R3$. The three other base rules perform transition refinement. Thus, excluding iterations that include applications of $R4$ (excluded in this work), all refinement iterations start with exactly one place refinement optionally followed by a transition refinement. The refinement iteration is then concluded by applying a modified rule.

Start

The start state marks the beginning of a refinement iteration. Each refinement iteration starts by randomly selecting one of the places within the portnet. From here on, the refinement iteration starts at random, either $R0$ or $R3$ is selected for further refinement of the portnet. As per the properties of portnets, each portnet must have one initial place and one final place. **Due to this restriction, $R3$ is excluded from being applied on these places. Thus, the first refinement iteration, during which the portnet only consists of one single place, may not start with R3.** Furthermore, $R3$ is also excluded from being applied on the final place. The final place may change during refinement iterations; hence the generator keeps track of this and appropriately labels the final place while refinement occurs.

Rule R0

Within the allowed ruleset, if selected, the $R0$ state marks the first rule within a refinement iteration. This rule, as introduced earlier, refines a single place and the result is a $Place \rightarrow Transition \rightarrow Place$. However, this is not a valid construct. The transition must meet one condition to guarantee that the portnet is valid: it must be connected to an input or output. The only way to satisfy this condition is by either applying modified rules on the transition, or refining the transition further, ensuring that this property is satisfied later. The places that have been introduced during the refinement need not be considered. These places can further be refined in subsequent iterations, as opposed to transitions which must be closed (having a single input/output connected) to comply with the properties of portnets. What follows is that from $R0$, the only further rules that may be applied are modified rules or the transition refinement rules, $R1$ and $R2$.

Rule R3

The cyclic place refinement rule, $R3$, has two restrictions. As mentioned earlier, it may not be applied on the places denoted by start or final. This means that it may only be applied on intermediate places. This introduces an interesting construct given in Figure 4.4.

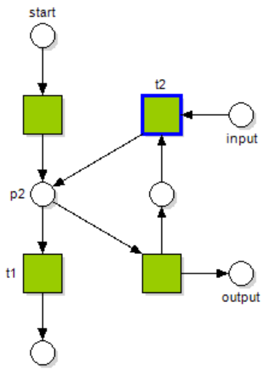


Figure 4.4: Possible choice property violation.

Observing this figure, one can note that $R3$ generates non-deterministic behavior. Furthermore, in the example given, if $t1$ was to have an input connected to it: the choice property would have been violated as not all transitions following P2 have the same direction of communication. Fortunately, $R3$ introduces an equal number of inputs and outputs regardless of what modified rule is applied. Thus, if an application of the modified rule of $R3$ would result in a choice property violation, this is replaced by the application of the other modified rule.
An interesting property of the modified rule of $R3$ is that it is an application of the base rule $R1$ and a modified version of $R1$, on the transition resulting from the application of $R3$.

Rule R1 and R4

$R1$ is always preceded by an application of the default rule, $R0$. The base rule of $R1$, which creates a $Transition \rightarrow Place \rightarrow Transition$ construct must always be followed by an application of a modified rule of $R1$ to reach a final state. This is due to the leg property, in the case of $R1$ there are two transitions, these must have different directions of communication thus an input and

output or vice-versa. A further extension to $R1$ is the ability to apply $R4$ on a construct that is generated from the base $R1$. $R4$ adds concurrency by adding an additional place between two transitions. This can theoretically be done an unbounded number of times; the only condition is that eventually a modified rule of $R1$ is applied.
If the decision is made to enable applications of $R4$, deriving a ComMA specification from the generated portnet is no longer possible. ComMA specifies state machines and thus does not support concurrency. This means that generation of a Petri Net is possible, but the conversion to a ComMA interface is not.

Rule R2

The non-deterministic transition refinement rule introduces non-deterministic portnet structures during generation. As a result of this, the parameter "prevalence of non-determinism" is directly controlled by $R2$. The base rule which adds an additional transition between two places directly violates the leg and possibly the choice property. The legs do not have two transitions and thus no communication in both directions. Furthermore, depending on how the inputs/outputs are added to these transitions, it could also violate the choice property. To prevent this from happening, one of the modified versions of $R2$ must be applied for the refinement iteration to finish.

4.4 Generation Scheme

The previous sections have introduced how the refinement rules are utilized and discussed constraints to the supplied parameters. This section will introduce the generational scheme of portnets though the usage of user-supplied complexity parameters.

Generation is split into two steps: selecting the rules and applying the rules upon a randomly selected place. The allowed ruleset in Section 4.3.1 introduced the concept of a refinement iteration. All rules within a refinement iteration are applied upon a selected place. Therefore, the generation first selects the rules and after that selects a place upon which this is applied.

To start generation, first the number of inputs and outputs is equalized. From the properties of the refinement rules it follows that only the modified rules of $R1$ can account for a difference in number of inputs or outputs. Therefore, the generator first equalizes the number of inputs and outputs.

1. Generation stars with the three complexity parameters supplied: inputs, outputs and prevalence of non-determinism.
2. If the number of inputs does not equal the number of outputs, continue to step three. Otherwise, abort these steps and continue generation.
3. Take the minimum of inputs and outputs, p_{min}.
4. Subtract p_{min} from the maximum of inputs and outputs generating p_{delta}.
5. p_{delta} denotes the number of modified rule 0 applications that must occur.
   - This will be the input or output version of Rule $R0$.
6. Start generation with p_{min} as the number of both inputs and outputs. Furthermore, supply the generator with a pre-existing number of modified inputs/outputs $R0$ applications.

This equalization process yields three things: An equal number of inputs and outputs, a list of pre-filled rule applications, and the unmodified prevalence of non-determinism. These parameters are supplied to the generator after which generation starts.

4.4.1 Applying Parameters to Generation

During the construction of a portnet the generator attempts to satisfy the three user-specified parameters. Out of these, non-determinism is the most complex to achieve, as was shown in Section 3.4. All rules introduce a number of inputs and/or outputs. However, only $R2$ and $R3$ introduce non-deterministic behavior. For that reason, an issue may occur if selection of rules is not done appropriately: the number of inputs and outputs may be reached before the non-determinism parameter is reached. Considering that possibility, the set of available rules are split into the set of deterministic and non-deterministic rules. Given the user-supplied prevalence non-determinism as $P_{expected}$ and the current prevalence of generation as $P_{current}$: if $P_{expected} < P_{current}$ all deterministic rules are excluded from the allowed ruleset. Conversely, if if $P_{expected} > P_{current}$ all non-deterministic rules are excluded.

4.4.2 Generational Description

The information provided in this section so far has introduced how the parameters are applied, and how the generator receives the right number of parameters for generation, by if necessary, pre-filling a list of $R1$ applications. We have now introduced all necessary concepts for generation.

Algorithm 1 shows the pseudocode associated with generation. As given earlier there are two parts to generation: the selection of rules and the application of rules.

Generation of rules starts by evaluating the parameter conditions. While inputs and outputs initially start out as equal, this can change during generation if a modified version of $R0$ is selected. Therefore, both these parameters are stored as individual variables. The first step towards rule selection is determining if the rules to be selected are from the non-deterministic or deterministic set of rules. From there, the generator randomly selects from this set a final rule. This is done by evaluating the allowed ruleset introduced in Section 4.3.1, which is a state machine. Once a final rule is found, this sequence of rules is stored. A new refinement iteration may now begin.
It should be noted that the allowed ruleset is restricted dynamically. Any path which may exceed the number of inputs or outputs is removed from selection.
Once the rules are selected, they must be applied. Each portnet starts with a single place. During the application, a sequence of rules (refinement iteration) is selected randomly from the list of rules. Afterwards, a random place is selected. The sequence of rules is then applied onto that place. These steps continue until no rules remain.

Algorithm 1 Generation of portnets

```
function GENERATOR(inputs,outputs,prevalence,rules=[])
    ▷ Note: inputs = outputs as defined in earlier generation scheme

    ▷ Note: Rules may be a list of already defined R1 applications.

    $CurrentPrevalence \leftarrow 0$

generate rule applications:
    while inputs, outputs != 0 do
        ▷ Refine until all parameters satisfied
        $CurrentPrevalence = computePrevalence(rules)$

        if $CurrentPrevalence < Prevalence$ and $inputs, outputs > 1$ then
            ▷ Apply $R2$ or $R3$ if the prevalence under the supplied value.

            rules += Pick a random modified version of rule two or three
            inputs, outputs -= ruleInputsOutputs(rules[-1])
        else
            ▷ If current prevalence is higher than supplied exclude R2 and R3.
            currentRule = Start
            tempRules = []

            while currentRule not Final do
                ▷ Refinement iteration is a sequence of rule applications
                currentRule = random(ruleset(currentRule))
                temprules += currentRule
            inputs, outputs -= ruleInputsOutputs(currentRule)
            rules += temprules

Application of rules:
    portnet = CreatePortnet()                    ▷ portnet starts with a single place
    while Rules not empty do
        rule = pop at a random index rule from rules
        place = random(portnet.places)
        portnet applyRule(portnet,place,rule)
```

4.5 Evaluating Non-Determinism

In the previous Section 4.3, the allowed ruleset was introduced. With these insights it is clear how the rules are applied at random. Concerning the parameters, inputs and outputs are trivial to achieve. They are positive integers that can always be satisfied using the refinement rules. Prevalence of non-determinism proves harder to achieve.

What has not been discussed so far is what range the prevalence of non-determinism can take. The definition is given in Section 4.2 as the fraction of arcs that are non-deterministic. A value of 1.00 would correspond with all arcs of the portnet being non-deterministic, this is currently not possible.

To evaluate the maximum achievable prevalence of non-determinism, we isolate the rules $R2$ and $R3$, these are the only rules capable of adding non-determinism during generation. Furthermore, as only $R1$ can independently add inputs or output, this implies that to achieve maximum amount of non-determinism, the number of inputs and outputs must be equal. This also implies that for an unequal number of inputs and outputs, the maximum achievable non-determinism is dependent on the minimum of inputs and outputs.

Another requirement to achieving the maximum amount of non-determinism is given due to the constraint imposed upon $R3$, it may not be applied upon the initial place or final place. Only the former can affect non-determinism as the final place does not have any outgoing arcs. Therefore, the initial place must be refined using rule $R2$. This introduces two non-deterministic arcs and two deterministic arcs resulting in the prevalence of non-determinism being 0.5. To continue, evaluating $R3$ it becomes apparent that it may turn a previously deterministic arc into a non-deterministic arc. Hence, applying this upon both places generated by $P2$ results four non-deterministic arcs and two deterministic arcs. The example of this is given in Figure 4.5, for clarity connected inputs and outputs are omitted.

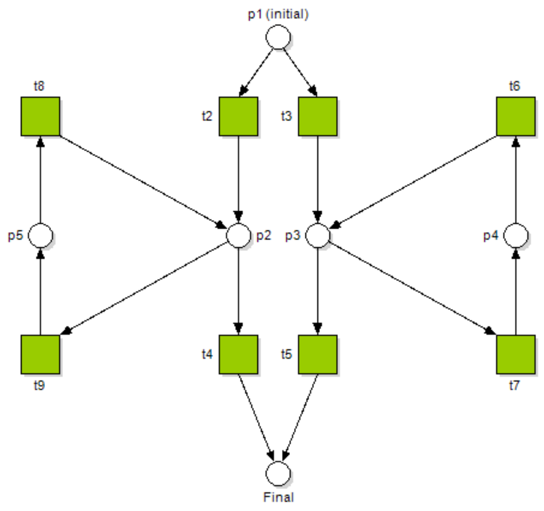


Figure 4.5: Portnet showing the maximum achievable non-determinism as 0.75

The example given consists of two deterministic arcs, and six non-deterministic arcs resulting in a prevalence of $\frac{3}{4}$. As should now be clear, in this example if $R3$ is once again applied upon the deterministic places ($P5$ and $P4$), this value would again increase, as the number of non-deterministic arcs would increase while the number of deterministic arcs would remain at two. The number of times this can be done is dependent on the supplied number of inputs and outputs. Therefore, if we assume $inputs = outputs = x$;

$$\lim_{x \to \infty} prevalence = 1.00$$

Thus for the best-case scenario, for which $R2$ is applied upon the initial place and $R3$ is only applied upon deterministic places, the prevalence converges to 1.00.

The introduced theoretical limit is probabilistic extremely unlikely. The odds of selecting $R3$ or $R2$ are equal. Furthermore, in the portnet a random place is selected to apply this rule on. Assuming $R3$ is however selected each time, in the example from Figure 4.5 this would need to be continuously applied on deterministic places, in the example these are places $P4$ or $P5$. As the size of the net increases, the probability of this happening decreases.

In the perfect case, where the generator makes specific choices to ensure the maximum possible non-determinism, it is possible to compute this maximum for a given number of inputs and outputs. The following formula only holds when $inputs, outputs > 4$, this is the minimum number required to have $R2$ applied upon the initial place and two applications of $R3$ upon the resulting places of $R2$, as seen in Figure 4.5.

$$rule_1 = |inputs - outputs| \tag{4.6}$$

$$rule_2 = 1 \tag{4.7}$$

$$rule_3 = min(inputs, outputs) - (rule_2 \cdot 2) \tag{4.8}$$

$$deterministic = (rule_1) + 2 \tag{4.9}$$

$$non\_deterministic = 2(rule_3 + rule_2) \tag{4.10}$$

$$max_{prevalence} = \frac{non\_deterministic}{deterministic + non\_deterministic} \tag{4.11}$$

The equations show how the theoretical maximum amount of non-determinism can be computed for the scenario where only non-deterministic rules are applied. Equation (4.6) computes the difference between inputs and outputs, this dictates the number of $R1$ applications that must follow. Equation (4.8) shows that the number of $R3$ applications is equal to the minimum of inputs and outputs. Furthermore, as the perfect case has one application of rule two (resulting in two inputs and outputs) subtract this. For the perfect case, the number of deterministic places is equal to the number of $R1$ applications and the two non-refined deterministic places. As both $R2$ and $R3$ add two non-deterministic arcs, from Eq. (4.10) the number of non-deterministic places follows. Finally, the maximum can be computed using the number of deterministic and non-deterministic places, as per the definition of the prevalence of non-determinism.

4.6 Limitations

This chapter has introduced an approach for generating portnets of various complexity according to refinement rules. We conclude this chapter by listing the limitations of the introduced approach. There are two crucial limitations to the methodology devised. The first limitation is the constrained generational scope of this methodology. The second limitation is a result of the randomized generation: the product of generation may not resemble what realistic interfaces look like.

4.6.1 Generational Scope

The paper [2] introduced in Section 3.4, which introduces the refinement rules that are utilized for generation, does not go into depth on the scope of portnets that may be generated. Consider Figure 4.6, an example of a valid portnet (labels and inputs/outputs removed for clarity). The construct shows that it is possible to go from place $P1$ to $P2$ through the intermediate place $P3$. If inputs and outputs are added that adhere to the choice property, it will satisfy all conditions of a portnet. With the current refinement rules, it is not possible to create such a construct. No rule makes it possible to connect between already existing places or transitions, only new places or transitions can be added. There is no rule in the format of the current refinement rules that could be added to resolve this limitation, while preserving portnet properties.

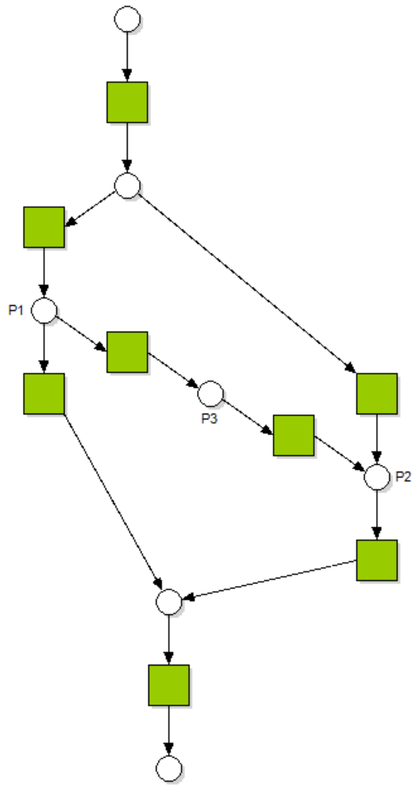


Figure 4.6: Example of a portnet that cannot be generated by the current methodology.

4.6.2 Randomized Complexity Generation

This generator uses a randomized constructional approach for generation. When not considering the constraints of user-supplied parameters and retaining the validity properties of portnets, all decisions are made at random. The issue that presents itself is that all rules are equally likely to be selected. The use case of the Dynamics project [1] concerns that of Thales. This generator does not adjust rule probabilities to account for more common interfaces as these interfaces have not been made available.

CHAPTER 5

Converting Internal Representation

In Chapter 4, the methodology for generating synthetic interfaces was introduced. Using the described methodology, the generator will have created an internal representation of a portnet. In this chapter, the conversion of the internal representation to a ComMA interface will be given.

5.1 Internal Representation

As portnets are state machines, there are various methods available for representing these. An example is the Petri Net Markup Language (PNML) [3]. Although these formats provide a standardized way for representing the results of generation, they make it difficult to convert this to a ComMA interface. Furthermore, if this generator is to later be extended to perform modifications upon interfaces this makes it even more problematic. Therefore, the generator opts for the mathematical representation of a portnet utilizing four sets. The internal portnet representation is a quadruple (P, T_{in}, T_{out}, A) consisting of: The set of places (P), the set of input enabled transitions (T_{in}), The set of output enabled transitions (T_{out}), and the set of arcs (A). The generator also provides a graphical visualization of the portnet. This is done through generating a PNML file which can be opened in Yasper[1], a process modelling software.

5.2 Conversion of Internal Representation

Given a portnet, to convert this to a valid ComMA interface, the specific ComMA notations need to be mapped to the four sets used for the representation. There are two files that must be generated for a ComMA interface: a signature file and an interface file. The signature file defines the signals and notifications, in this context these are the inputs and outputs respectively. In the following examples Figure 5.1 is used as a reference portnet.

[1]`http://www.yasper.org/`

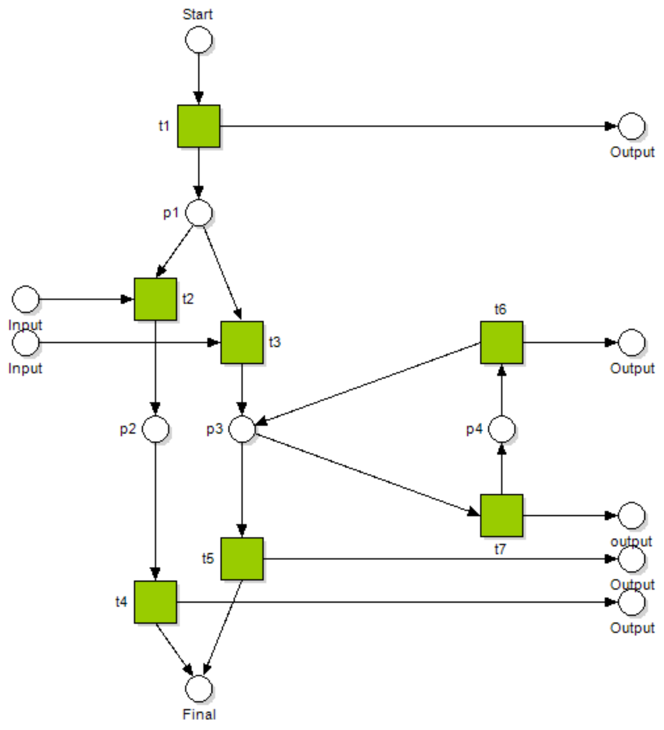


Figure 5.1: Example portnet with two inputs, five outputs and non-determinism of 0.57.

The signature file is a listing of the signals (inputs) and notifications (outputs) within the portnet. Within our notation the label for an input connected to a transition "*t2*" is *t2IN*. Conversely, if this were an output this would be labeled "*t2OUT*". For this example, the signature file "***example.signature***" is generated and displayed in Listing 5.1. We do not use any types, therefore the first import can be an empty file. The signature file is uncomplicated in our case. We list all the inputs under signals and all the outputs under notifications.

Listing 5.1: ComMA signature file for example 1.

```
import "Example.types"
signature Example

signals
/* Inputs */
t2IN
t3IN
notifications
/*
 * Outputs
 */
 t1OUT
 t4OUT
 t5OUT
 t6OUT
 t7OUT
```

Generating the interface happens using a 7-step approach. Recall from Section 3.2 that the ComMA representation is a state machine. Thus, transitions would remain as is, while places would be states. To understand the difference between input and output enabled transitions the examples in Figures 5.2 and 5.3 are shown for places $P2$ and $P3$, using the earlier provided portnet example. The syntax for the first input enabled transition shows it to be triggered by the signal $t2IN$. Furthermore, if the state were to contain more outgoing arcs towards input enabled transitions, this syntax would simply need to be duplicated within the state block. To continue, the output enabled transitions from place $P2$ show that instead of having a trigger, an action is performed namely sending the notification.

Listing 5.2: Syntax for input enabled transitions.

```
state p1 {

    transition trigger: t2IN
        next state: p2


        transition trigger: t3IN
            next state: p3

}
```

Listing 5.3: Syntax for output enabled transitions.

```
state p3 {

    transition
        do: t5OUT
        next state: final

    transition
        do: t7OUT
        next state: p4
}
```

Having discussed the terminology and syntax, the 7-step approach for the conversion of the generated portnet to a ComMA interface is given.

1. First determine the initial state: $\exists p \in \ \mathrm{P} \ : (\forall a \in A : a_{destination} \neq p)$
2. Determine the final state: $\exists p \in \ \mathrm{P} \ : (\forall a \in A : a_{origin} \neq p)$
3. For all places, p: generate the corresponding ComMA code without adding any transitions
4. Add the initial state marking
5. For all states, p: add empty transitions by evaluating the set of arcs with that state as origin
6. For each transitions, t: if they belong to the set of input enabled transitions add a signal, otherwise add a notification. The next state is given by the arc a, where $a_{origin} = t$ and thus $a_{destination} =$ next state.
7. All states that are supposed to go towards the final state will instead go back to the initial (start) state.

Using this approach the example from Figure 5.1 can be converted. The result is shown as the full interface file

Listing 5.4: ComMA interface file for example 1.

```
import "Example.signature"

interface Example version "1.0"
machine Example {
    initial state Start {
        transition
            do: t1
            next state: p1
    }

    state p1 {
        transition trigger: t2
            next state: p2

        transition trigger: t3
            next state: p3
    }

    state p2 {
        transition
            do: t4
            next state: Start
    }

    state p3 {
        transition
            do: t5
            next state: Start

        transition
            do: t7
            next state: p4
    }

    state p4 {
        transition
            do: t6
            next state: p3
    }


}
```

CHAPTER 6

Experiments

In the previous chapters the the generation of synthetic interfaces was introduced using a parameterized generation approach. In this chapter various experiments will be conducted to showcase generation using various parameters.
The experiments will be conducted by supplying the generator with various parameters and discussing the resulting output.

6.1 Maximum Prevalence of Non-Determinism

In Section 4.5, constraints to the parameters were introduced. Interestingly, the maximum amount of non-determinism for a given number of inputs and outputs was stated to near 1.00 in the ideal case. This ideal case consisted of continuous refinements using rule $R3$ upon deterministic places, following the application of $R2$ on the initial place.

The setup of this experiment will attempt to evaluate the maximum achievable prevalence of non-determinism. The chosen parameter values will be:

$$\begin{aligned}&\mathit{prevalence} = 1.0\\&\mathit{inputs} = \mathit{outputs} = x = \{x : x \in [0, 200], x \bmod 4 = 0\}\end{aligned}$$

The observed non-determinism will be an average over twenty measurements using these parameter values.

The results of the experiment are visible in Figure 6.1 (See Appendix A.2).
There are two interesting findings from this result. As expected, the number of times the theoretical limit is reached is one out of all measurements. Remember that to obtain the maximum amount of non-determinism, all legs of $R2$ must be refined using an application of $R3$. The probability of this happening in a perfect sequence is small.

The second finding is that as the number of inputs and outputs increases, the number of outliers decreases. Initially it is visible that the computed average fluctuates, even with twenty measurements conducted. This behavior disappears as the number of inputs and outputs increases. At that point it becomes increasingly less likely for $R3$ refinements to continuously pick the deterministic places within the portnet. Therefore, higher number of inputs and outputs translate to a probabilistically higher chance of greater non-determinism.

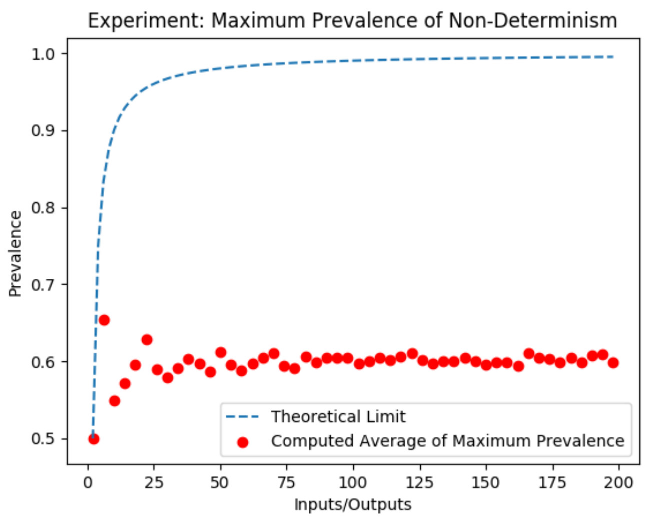


Figure 6.1: Maximum prevalence of non-determinism as a function of the inputs and outputs.

To continue evaluating the maximum prevalence of non-determinism and to show how unlikely it is to ever achieve it, the previous experiment will be conducted again. This time no averages will be taken, instead for each data point 200 measurements will be conducted, and the maximum observed will be displayed. The results are visible in Figure 6.2.

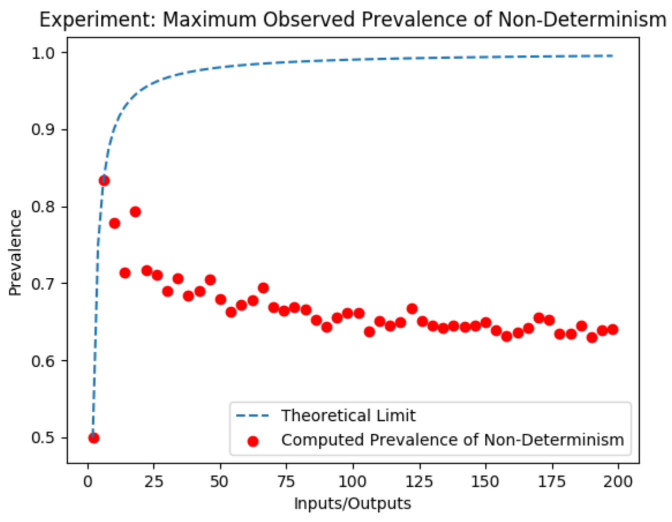


Figure 6.2: Maximum observed prevalence of non-determinism.

The data points (Appendix refappendix:experiment2) show that even when conducting 10^4 measurements in total, the theoretical maximum is only reached twice. Furthermore, as indicated earlier we notice that the maximum observed follows a downwards line. If we observe the properties of rules $R2$ and $R3$ (Table 6.1); it is apparent that $R2$ and $R3$ have an overall prevalence of $\frac{1}{2}$ and $\frac{2}{3}$ respectively. The key property of $R3$ is that it does not modify the count of deterministic arcs within the portnet; if the place it is refined upon is deterministic. Given a large enough net, this will likely not happen. Therefore, as both non-deterministic rules are equally likely the average of their overall prevalence can be taken which equals $\frac{7}{12} \approx 0.58$. If we observe the data from the first experiment in Figure 6.1, it is apparent that this value of 0.58 is eventually converged towards.

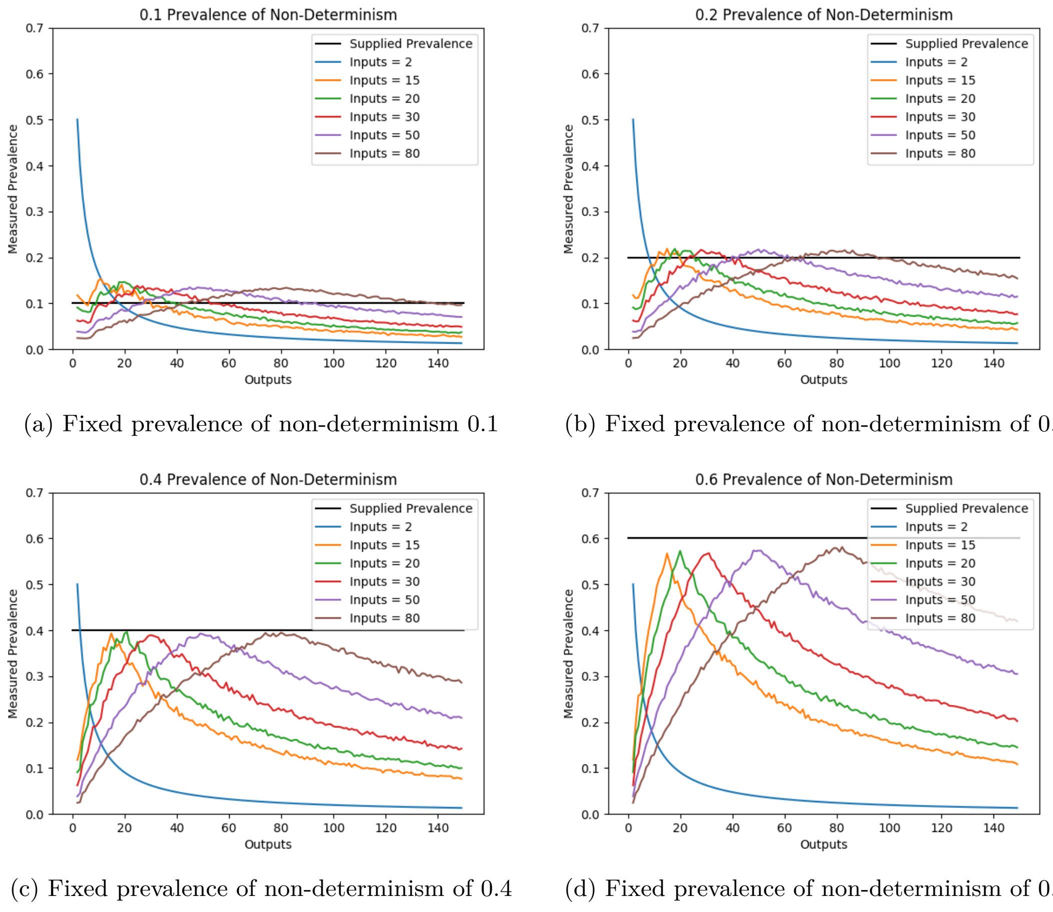
Table 6.1: Rule properties showing (non)deterministic arcs added.

Rule	Deterministic Arcs	Non-Deterministic Arcs
2	2	2
3	1 (0 If added to a deterministic place)	2

6.2 Relation Input/Outputs to Prevalence

The experiments conducted in the previous Section 6.1 revealed the maximum theoretical limit of non-determinism. In that experiment, the number of inputs and outputs was set to be equal. Recall that the maximum attainable non-determinism is highest when inputs and outputs are equal. We will now consider varying number. In the following experiments inputs and prevalence of non-determinism will be set to a fixed amount. The experiment will then show observed non-determinism as a function of the outputs.
It should be noted that inputs and outputs are interchangeable in this experiment. If the outputs were fixed instead of inputs the results would remain the same as this does not change how non-determinism is computed.

Figure 6.3 shows the results of the conducted experiments. Four subfigures are shown indicating the fixed amount of prevalence given to the generator as parameter from the set $\{0.2, 0.1, 0.4, 0.6\}$. The six curves within the plot denote the fixed number of inputs used for the experiment from the set $\{2, 15, 20, 30, 50, 80\}$. Therefore, each plot is the amount of observed prevalence as a function of the outputs: the other parameters are fixed.



(a) Fixed prevalence of non-determinism 0.1

(b) Fixed prevalence of non-determinism of 0.2

(c) Fixed prevalence of non-determinism of 0.4

(d) Fixed prevalence of non-determinism of 0.6

Figure 6.3: Prevalence in relation to outputs.

Several interesting remarks can be made about the results. First, as to be expected when fixing the number of inputs to two, it is unavoidable that the amount of non-determinism exceeds that. The generator tries to match the parameter and picks rule $R2$. This fills all the required inputs and outputs and therefore the generator stops.

Another observation that can be made is that as opposed to the first experiment, the results in Figure 6.3d do not exceed the 0.6 boundary of prevalence of non-determinism. This is a result of how the generator tries to satisfy parameter values. During iteration, the prevalence of the portnet that is still under generation changes. As introduced in Section 4.5, during construction of the portnets the generator alternates between deterministic and non-deterministic rulesets. Therefore, if the parameter value of 0.6 is reached the generator will switch to a deterministic rule application. This might result in the value dropping below this threshold and for it no longer being possible to satisfy the parameter value. Fortunately, the error is negligible.

CHAPTER 7

Conclusion

The methodology presented in the Dynamics project solves the issue of detecting and correcting incompatibilities that arise when service interfaces are updated. This methodology has yet to be tested for the scalability of its time complexity, in both detecting incompatibilities and correcting them through adapter generation. This thesis builds towards that by introducing the generation of synthetic interfaces with user-supplied complexity parameters. The complexity of interfaces is defined as being dependent on the number of inputs, outputs and the prevalence of non-deterministic arcs. In the case of non-determinism, an intuitive definition was given such that it allows for comparisons to be made between different interfaces using solely the parameters provided. The methodology for the generation of interfaces was devised using the provided refinement rules [2]. The basis of this methodology is generating an internal representation of a portnet, using the refinement rules and complexity parameters. This resulting portnet representation is then converted to a ComMA interface specification.

The usage of the refinement rules can be considered a limitation of the introduced generator. Refinement performs gradual addition/expansion of the portnet. This limitation is a result of choosing a refinement approach: it is not possible to connect between already existing places. This methodology can thus generate portnet interfaces but does not cover the full set of portnets. By transitivity, as portnets are a subset of the full scope of ComMA interfaces, this generator can thus not generate all possible ComMA interfaces. However, it is unlikely that this significantly affects the purpose of this work. The generated interfaces that can currently be created provide a starting point for scalability evaluations.

The experiments conducted within this thesis concern themselves with evaluating the constraints of the proposed methodology. The first set of experiments attempts to evaluate the theoretical limit of non-determinism. In Section 4.5 the theoretical limit was established to converge to one. The conducted experiments showed how unlikely it is for this value to be reached. In practice, we have found this limit to lie around the range of $\frac{2}{3}$. This is approximately the average non-determinism $R2$ and $R3$ introduce.
Furthermore, the second set of experiments show that the maximum amount of non-determinism is given by the relation between inputs and outputs. Specifically, the maximum amount of non-determinism for a given number of inputs is achievable when this number equals the number of outputs. The experiment however shows that there is a certain error that is to be expected regardless of whether the number of inputs and outputs equal. This is not just a result of rounding error but also due to the way the generator tries to fit these parameters.

7.1 Future Work

The current approach provides a methodology for generating interfaces of varying complexity. The work in this thesis opens further possibilities for research which are of interest to the Dynamics.

7.1.1 Scalability Analysis

The goal of this thesis is to provide a starting point for scalability analysis. Before scalability analysis can be conducted, there needs to be an updated version of the interface. The introduced generator only constructs the original interface. What remains is to perform modifications upon this generated interface. A paper regarding generated adapters [7] introduces possible transformation rules. These rules can be used as a baseline for further research in the topic of performing modifications.

Once the concept of modified interfaces is introduced, it can be utilized to perform a scalability analysis of the methodology proposed in the Dynamics [1] project. Of interest will be the correlation between the complexity of the portnet versus the performed modifications upon the interface. Furthermore, this research could then be extended to delve into the actual Dynamics methodology to determine the bottlenecks and present mitigations.

7.1.2 Modifying Ruleset

The current experimental generator provides an approach to generating portnets which are converted into their corresponding ComMA interface specification. This approach based upon the refinement rules, while functional, does not accurately represent possible interfaces. One of the primary limitations of generator concerns the randomized generation. Generation is uniquely controlled by the parameters that are supplied and rule selection is done completely at random. The issue lies in the randomness of the approach. Although it is helpful for scalability evaluations to have completely random interfaces, as otherwise unnoticed complexity bottlenecks could present itself, it would be desirable if further research can be done into the more commonly utilized interfaces that will be supplied to the Dynamics project. Desirably, this research would focus on two areas.

The first point is the ability to reverse engineer interface specifications to their complexity parameters and their refinement rules. This can be used to verify whether the generator can generate such constructs. If this is possible, determine which rule applications are more common so that accurate probabilities can be evaluated.

The second point would focus on interfaces that cannot currently be generated. The current methodology does not cover the generation of all possible portnets. To attempt to cover more complete set of interfaces, it would be desirable if future research attempts to extend the current ruleset or replace them by rules that cover a larger set of compatible portnets.

7.1.3 Complexity

This thesis provides complexity as three parameters: inputs, outputs and prevalence of non-determinism. There are more constructs that could be proven to introduce complexity. For example, the length of non-deterministic legs could be studied to show if they affect complexity. Furthermore, another interesting parameter is the introduction of looping structures in portnets. To see how complexity may affect the proposed methodology in Dynamics, further research could incorporate knowledge obtained by looking at the technicalities within the project. Studying how the incompatibilities are detected, and how adapters are generated can prove to be of interest in determining what should be considered complex.

Finally, currently there is no direct notion of complexity: it is an aggregate of three parameters. If further complexity parameters are added, it would prove interesting to conduct an analysis of which parameters have the largest effect on scalability. From this it would be possible to assign weights to each parameter and perhaps come up with a single parameter that is an aggregate of all these parameters. Having a clear definition of complexity that is quantifiable would facilitate making comparisons between the complexity of two interfaces.

7.2 Ethics

The aim of this thesis is to aid in the Dynamics project. This thesis on its own does not have any direct ethical implications. The research we present does not directly influence, or interact with individuals or entities. As the focus of the research is to aid the Dynamics [1] project, we can consider if the ethical concerns regarding Dynamics may indirectly be linked back to our research. Dynamics aims to automate a costly (both time and money wise) manual task, the updating of service interfaces. It increases the longevity of systems and thus makes these more productive. On the other hand, this can be seen as automating a process which previously required human labor, a topic which has been discussed often for their ethical implications [9]. However, the individuals that would usually be involved within the work of updating service interface are those that possess specific domain knowledge. Furthermore, the methodology proposed in Dynamics does not have the ability to cover the set of all possible interfaces. Therefore, the normal ethical implications of automation do not fully apply here.

What may be of interest is that the Dynamics project is partially funded by Thales. Thales operates within the defense industry and thus while one cannot directly assume what the research may be used for, there is the chance of it being used in a way that some people consider morally wrong.

In conclusion, the ethical implications of the research done within this thesis are greatly dependent on how further work is used.

Bibliography

[1] B. Akesson, J. Sleuters, S. Weiss, and R. Begeer. Towards continuous evolution through automatic detection and correction of service incompatibilities. *ModComp*, 2019.

[2] D. Bera, K. M. Van Hee, M. Van Osch, J. M. E. van der Werf, et al. A component framework where port compatibility implies weak termination. In *PNSE*, pages 152–166, 2011.

[3] J. Billington, S. Christensen, K. Van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The petri net markup language: concepts, technology, and tools. In *International Conference on Application and Theory of Petri Nets*, pages 483–505. Springer, 2003.

[4] H. Bossenbroek and R. van Hees. The INAETICS architecture - Introducing INAETICS White paper, Tech. Rep., 2015.[Online].

[5] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM (JACM)*, 30(2):323–342, 1983.

[6] R. P. Dick, D. L. Rhodes, and W. Wolf. Tgff: task graphs for free. In *Proceedings of the Sixth International Workshop on Hardware/Software Codesign.(CODES/CASHE'98)*, pages 97–101. IEEE, 1998.

[7] C. Gierds, A. J. Mooij, and K. Wolf. Reducing adapter synthesis to controller synthesis. *IEEE Transactions on Services Computing*, 5(1):72–85, 2010.

[8] I. Kurtev, M. Schuts, J. Hooman, and D.-J. Swagerman. Integrating interface modeling and analysis in an industrial setting. In *MODELSWARD*, pages 345–352, 2017.

[9] M. Lawrence, C. Roberts, and L. King. Managing automation. *Employment, inequality and ethics in the digital age*, 2017.

[10] N. Lohmann, P. Massuthe, C. Stahl, and D. Weinberg. Analyzing interacting ws-bpel processes using flexible model generation. *Data & Knowledge Engineering*, 64(1):38–54, 2008.

[11] P. Massuthe, W. Reisig, and K. Schmidt. *An operating guideline approach to the SOA*. Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät ..., 2005.

[12] P. Massuthe and D. Weinberg. Fiona a tool to analyze interacting open nets. 2008.

[13] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[14] S. Stuijk, M. Geilen, and T. Basten. Sdf^ 3: Sdf for free. In *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, pages 276–278. IEEE, 2006.

[15] W. M. Van der Aalst. The application of petri nets to workflow management. *Journal of circuits, systems, and computers*, 8(01):21–66, 1998.

[16] W. M. van der Aalst, A. J. Mooij, C. Stahl, and K. Wolf. Service interaction: Patterns, formalization, and analysis. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 42–88. Springer, 2009.

[17] K. van Hee, J. Hidders, G.-J. Houben, J. Paredaens, and P. Thiran. On the relationship between workflow models and document types. *Information Systems*, 34(1):178–208, 2009.

APPENDIX A

Experiment Datapoints

A.1 6.1 Experiment 1

Table A.1: Data points showcasing the number of inputs and outputs corresponding to the maximum observed non-determinism and theoretical limit.

Inputs and Outputs	Obtained Non-Determinism	Theoretical Limit
2	0.5000	0.5
6	0.6548	0.75
10	0.5497	0.8333333333333334
14	0.5725	0.875
18	0.5961	0.9
22	0.6286	0.9166666666666666
26	0.5904	0.9285714285714286
30	0.5790	0.9375
34	0.5906	0.9444444444444444
38	0.6029	0.95
42	0.5970	0.9545454545454546
46	0.5867	0.9583333333333334
50	0.6126	0.9615384615384616
54	0.5965	0.9642857142857143
58	0.5881	0.9666666666666667
62	0.5966	0.96875
66	0.6049	0.9705882352941176
70	0.6109	0.9722222222222222
74	0.5943	0.9736842105263158
78	0.5914	0.975
82	0.6068	0.9761904761904762
86	0.5985	0.9772727272727273
90	0.6051	0.9782608695652174
94	0.6042	0.9791666666666666
98	0.6052	0.98
102	0.5969	0.9807692307692307
106	0.6010	0.9814814814814815
110	0.6050	0.9821428571428571
114	0.6024	0.9827586206896551
118	0.6066	0.9833333333333333
122	0.6110	0.9838709677419355
126	0.6020	0.984375
130	0.5967	0.9848484848484849
134	0.6005	0.9852941176470589
138	0.6008	0.9857142857142858
142	0.6045	0.9861111111111112
146	0.6002	0.9864864864864865
150	0.5963	0.9868421052631579
154	0.5984	0.9871794871794872
158	0.5984	0.9875
162	0.5938	0.9878048780487805
166	0.6109	0.9880952380952381
170	0.6051	0.9883720930232558
174	0.6040	0.9886363636363636
178	0.5984	0.9888888888888889
182	0.6045	0.9891304347826086
186	0.5987	0.9893617021276596
190	0.6082	0.9895833333333334
194	0.6092	0.9897959183673469
198	0.5989	0.99

A.2 6.1 Experiment 2

Table A.2: Data points showcasing the number of inputs and outputs corresponding to the maximum observed non-determinism and theoretical limit.

Inputs and Outputs	**Maximum Observed Prevalence**	**Theoretical Limit**
2	0.5000	0.5
6	0.8333	0.75
10	0.7778	0.8333333333333334
14	0.7143	0.875
18	0.7941	0.9
22	0.7174	0.9166666666666666
26	0.7115	0.9285714285714286
30	0.6897	0.9375
34	0.7059	0.9444444444444444
38	0.6842	0.95
42	0.6905	0.9545454545454546
46	0.7045	0.9583333333333334
50	0.6800	0.9615384615384616
54	0.6635	0.9642857142857143
58	0.6724	0.9666666666666667
62	0.6774	0.96875
66	0.6940	0.9705882352941176
70	0.6690	0.9722222222222222
74	0.6644	0.9736842105263158
78	0.6688	0.975
82	0.6667	0.9761904761904762
86	0.6534	0.9772727272727273
90	0.6444	0.9782608695652174
94	0.6559	0.9791666666666666
98	0.6616	0.98
102	0.6618	0.9807692307692307
106	0.6381	0.9814814814814815
110	0.6514	0.9821428571428571
114	0.6447	0.9827586206896551
118	0.6496	0.9833333333333333
122	0.6680	0.9838709677419355
126	0.6508	0.984375
130	0.6457	0.9848484848484849
134	0.6418	0.9852941176470589
138	0.6449	0.9857142857142858
142	0.6444	0.9861111111111112
146	0.6458	0.9864864864864865
150	0.6500	0.9868421052631579
154	0.6396	0.9871794871794872
158	0.6321	0.9875
162	0.6366	0.9878048780487805
166	0.6416	0.9880952380952381
170	0.6550	0.9883720930232558
174	0.6532	0.9886363636363636
178	0.6348	0.9888888888888889
182	0.6354	0.9891304347826086
186	0.6452	0.9893617021276596
190	0.6296	0.9895833333333334
194	0.6385	0.9897959183673469
198	0.6414	0.99

APPENDIX B

Example Interfaces and Conversions

B.1 Example 1

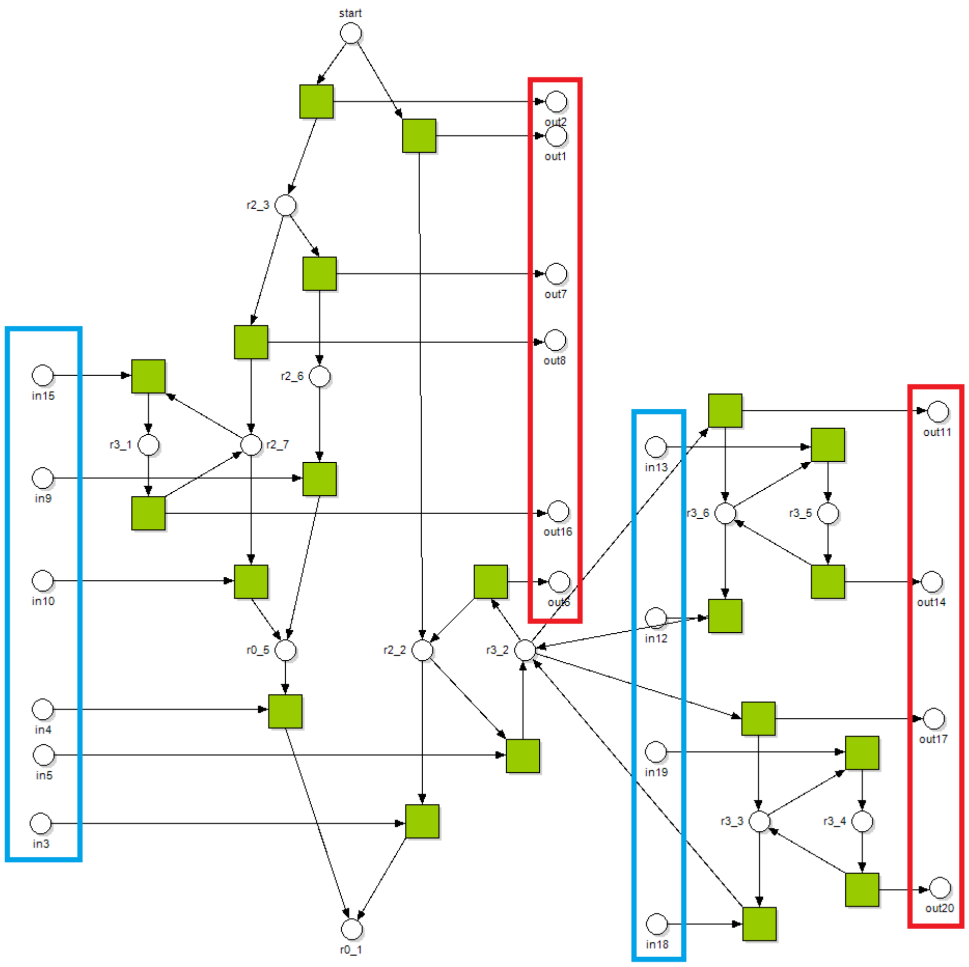


Figure B.1: Generated interface using the parameters inputs = outputs = 10, prevalence = 0.8 | Resulting prevalence = 0.75

Listing B.1: ComMA interface specification for example 1.

```
machine Example {
    initial state start {
        transition
            do: out2
            next state: r2_3
        transition
            do: out1
            next state: r2_2
    }
    state r2_3 {
        transition
            do: out7
            next state: r2_6
        transition
            do: out8
            next state: r2_7
    }
    state r2_6 {
        transition trigger: in9
            next state: ro_5
    }
    state r2_7 {
        transition trigger: in15
            next state: r3_1
        transition trigger: in10
            next state: ro_5
    }
    state r3_1 {
        transition
            do: out16
            next state: r2_7
    }
    state r0_5 {
        transition trigger: in4
        next state: start
    }
    state r2_2 {
        transition trigger: in5
            next state: r3_2
        transition trigger: in3
            next state: start
    }
    state r3_2 {
        transition
            do: out6
            next state: r2_2
        transition
            do: out11
            next state: r3_6
        transition
            do: out17
            next state: r3_3
    }
    state r3_6 {
        transition trigger: in13
            next state: r3_5
        transition trigger: in12
            next state: r3_2
    }
    state r3_5 {
        transition
            do: out14
            next state: r3_6
    }
    state r3_4 {
        transition
            do: out20
            next state: r3_3
    }
    state r3_3 {
        transition trigger: in18
            next state: r3_2
        transition trigger: in19
            next state: r3_4
    }
}
```

B.2 Example 2

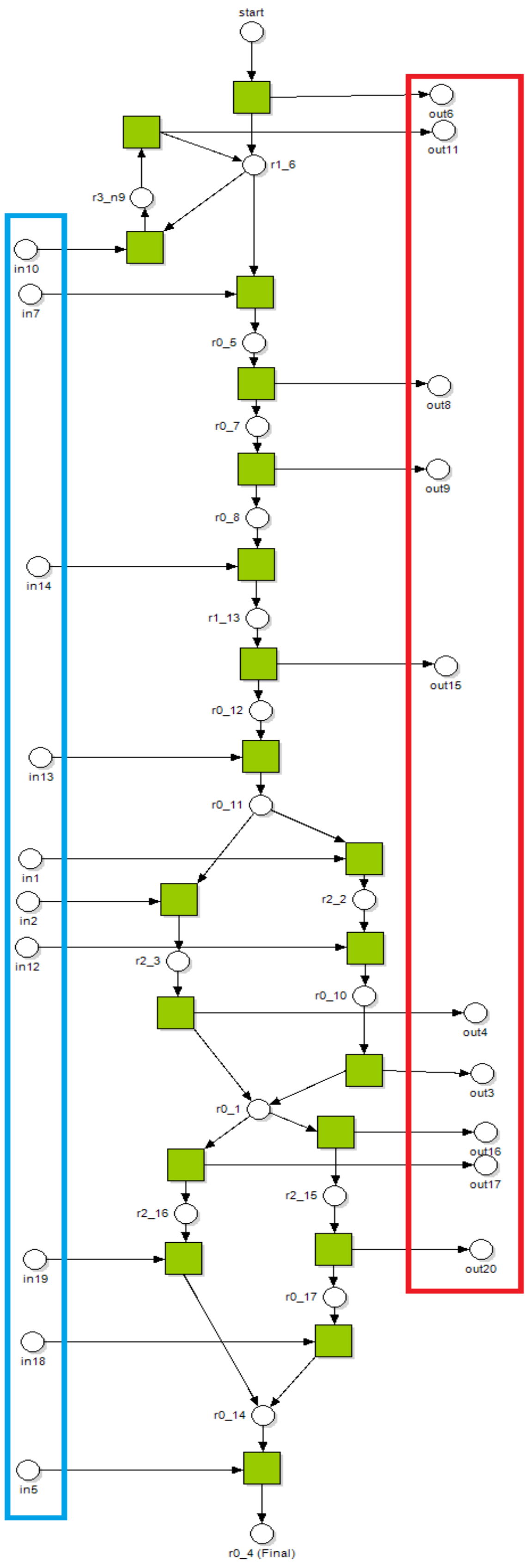


Figure B.2: Generated interface using the parameters inputs = outputs = 10 and prevalence = 0.3 | Resulting prevalence = 0.3

Listing B.2: ComMA interface specification for example 2.

```
machine Example {
    initial state start {
        transition
            do: out6
            next state: r1_6
    }
    state r1_6 {
        transition trigger: in7
            next state: r0_5
        transition trigger: in10
            next state: r3_n9
    }
    state r0_5 {
        transition
            do: out8
            next state: r0_7
    }
    state r0_7 {
        transition
            do: out9
            next state: r0_8
    }
    state r0_8 {
        transition trigger: in14
            next state: r1_13
    }
    state r1_13 {
        transition
            do: out15
            next state: r0_12
    }
    state r0_12 {
        transition trigger: in13
            next state: r0_11
    }
    state r0_11 {
        transition trigger: in1
            next state: r2_2
        transition trigger: in2
            next state: r2_3
    }
    state r2_2 {
        transition trigger: in12
            next state: r0_10
    }
    state r2_3 {
        transition
            do: out4
            next state: r0_1
    }
    state r0_10 {
        transition
            do: out3
            next state: r0_1
    }
    state r0_1 {
        transition
            do: out16
            next state: r2_15
        transition
            do: out17
            next state: r2_16
    }
    state r2_16 {
        transition trigger: in19
            next state: r0_14
    }
    state r2_15 {
        transition
            do: out20
            next state: r0_17
    }
    state r0_17 {
        transition trigger: in18
            next state: r0_14
    }
    state r0_14 {
        transition trigger: in18
            next state: start
    }
}
```

B.3 Example 3

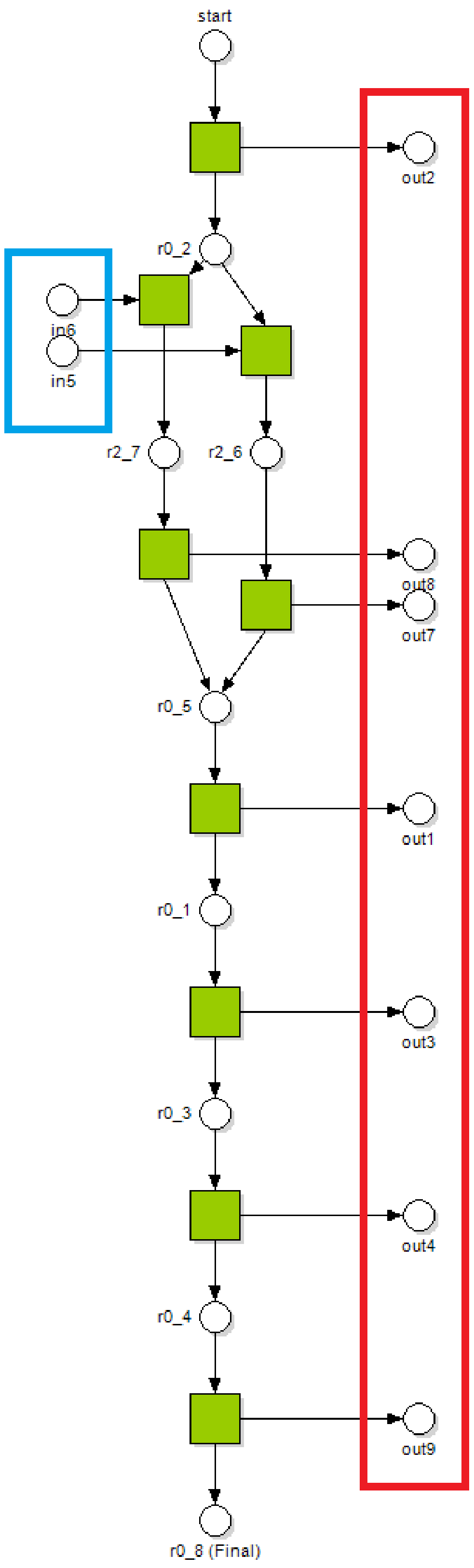


Figure B.3: Generated interface using the parameters inputs = 2, outputs = 7 and prevalence = 0.8 | Resulting prevalence 0.22

Listing B.3: ComMA interface specification for example 3.

```
machine Example {
    initial state start {
        transition
            do: out2
            next state: r0_2
    }
    state r0_2 {
        transition trigger: in6
            next state: r2_7
        transition trigger: in5
            next state: r2_6
    }
    state r2_7 {
        transition
            do: out8
            next state: r0_5
    }
    state r2_6 {
        transition
            do: out7
            next state: r0_5
    }
    state r0_5 {
        transition
            do: out1
            next state: r0_1
    }
    state r0_1 {
        transition
            do: out3
            next state: r0_3
    }
    state r0_3 {
        transition
            do: out4
            next state: r0_4
    }
    state r0_4 {
        transition
            do: out9
            next state: start
    }

}
```