Verification of Factorio Belt Balancers using Petri Nets

Analyse von Factorio Belt-Belancer mit Hilfe von Petri Netzen
Bachelor thesis by Andre Leue
Date of submission: February 22, 2021

1. Review: Kirstin Peters
2. Review: Anna Schmitt
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Verification of Factorio Belt Balancers using Petri Nets
Analyse von Factorio Belt-Belancer mit Hilfe von Petri Netzen

Bachelor thesis by Andre Leue

1. Review: Kirstin Peters
2. Review: Anna Schmitt

Date of submission: February 22, 2021

Darmstadt

Bitte zitieren Sie dieses Dokument als:
URN: urn:nbn:de:tuda-tuprints-176215
URL: http://tuprints.ulb.tu-darmstadt.de/17621

Dieses Dokument wird bereitgestellt von tuprints,
E-Publishing-Service der TU Darmstadt
http://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de

Contents

1 Introduction **4**

2 Used notation **4**

3 Description of Belt Balancers components **5**

4 First model **7**
4.1 Non-deterministic model . . . 7
4.2 Using alternating input / output . . . 7
4.3 Handling of empty inputs / outputs . . . 9
4.4 Enforcing of input / output priorities . . . 11
4.5 Problems with the current model . . . 14

5 Second model **19**
5.1 Used modules . . . 19
5.2 Communication between modules . . . 22
5.3 Module internals . . . 23
5.3.1 Item generator . . . 23
5.3.2 Item consumer . . . 25
5.3.3 Item transfer . . . 25
5.3.4 Item loopback . . . 26
5.3.5 Item splitter . . . 26
5.3.6 Dummy modules . . . 35
5.4 Examples for module usage . . . 35

6 Properties of Belt Balancer **39**
6.1 Used sets and functions . . . 39
6.2 Balancer is never stuck . . . 40
6.3 Balanced output . . . 42
6.4 $a \to b$ Belt Balancer . . . 44
6.5 Throughput limited $a \to b$ Belt Balancer . . . 44
6.6 Throughput unlimited $a \to b$ Belt Balancer . . . 45
6.7 Universal $a \to b$ Belt Balancer . . . 45

7 Verification **46**
7.1 Preparation . . . 46
7.2 Modified deadlocks . . . 47
7.3 Place invariants . . . 47
7.4 Model checker . . . 49

8 Conclusion **52**

1 Introduction

Factorio [1] is a game about designing, building and maintaining factories. After crash landing on an alien planet the player has to launch a rocket back to space in order to win the game. To achieve this it is required to automate various processes, which comes with its own challenges. These range from space occupying trees, over logistical problems, to hostile, native inhabitants, which are not happy about the increasing pollution levels caused by the ever growing factory.

In the following chapters we will focus on one of the logistical problems. Factorio provides multiple transportation methods to get items from A to B, and transport belts are one of them. To work with belts efficiently Belt Balancers are used in certain locations, which act as load balancer and consist of splitters, underground belts and normal belts. These components are described in detail in Section 3. While the design and design process of Belt Balancers is certainly an interesting problem, we will focus on the verification and automation of this verification instead. We will analyse whether the outputs are actually balanced, how the Belt Balancer affects overall throughput and its resistance to external interferences. Since the manual verification of Belt Balancers is rather time consuming and monotone, it is desirable to automate this process. To do this we model the components of Belt Balancers with Petri Nets. Later on those Petri Net modules can be arranged to form a given Belt Balancer. Additionally we use linear temporal logic to define properties, which are commonly used to describe Belt Balancers in the Factorio community. For the automatic verification of this model the modelling language PROMELA is used, together with SPIN as interpreter. While the translation of Petri Nets to PROMELA works, the actual verification with SPIN does not. Reasonably sized Belt Balancers, which where translated with the algorithm we use, cannot be verified with SPIN due to internal limitation of the interpreter.

2 Used notation

A Petri Net is defined as a tuple (P, T, F) [2] [3], where P is the set containing all places, T contains all transitions and $F \subseteq ((P \times T) \cup (T \times P))$ is a flow relation. This relation contains all arcs of the Petri Net, therefore if $(a, b) \in F$ there exists an arc from a to b. To reduce the overall complexity of the Petri Nets we use inhibitor arcs to simplify inverse places. For this the Petri Net definition is expanded to (P, T, F, I). The sets P, T, F are defined as in the previous version. The new set $I \subseteq (P \times T)$ describes all inhibitor arcs, similar as F does for all normal arcs. We also use notation to access places influencing or influenced by a given transition $t \in T$. For this ${}^\bullet t$ contains all places $p \in P$, such that there exists an arc from t to p, and is defined as ${}^\bullet t := \{p \in P : (p, t) \in F\}$. Its counterpart is $t^\bullet$. This set contains all places which receive a token, if the transition $t \in T$ fires. It is defined as $t^\bullet := \{p \in P : (t, p) \in F\}$. Similar to ${}^\bullet t$ the set t_{inhib} contains all places, which inhibit the activation of $t \in T$. Therefore it is defined as $t_{\text{inhib}} := \{p \in P : (p, t) \in I\}$. If a Petri Net is drawn, each place in P is represented by a circle, each transition in T by a box and each arc in F by an arrow. To represent an inhibitor arc in I an arrow is used as well, but unlike the arrow for a normal arc it has a dot as head. Figure 1a contains examples for this.

Additionally two kinds of double headed arrows are used to reduce the total amount of arcs shown. An arrow with a pointy head at each end between (a, b) translates to $(a, b), (b, a) \in F$. The second kind has a dot at one end and a normal head at the other end. An arrow between a and b, where the dot is at a and the normal head at b, translates to $(a, b) \in F \wedge (b, a) \in I$. Figure 1b visualises this.

A marking of a Petri Net is defined as a function $M := P \to \mathbb{N}$ which maps a given place to the amount of tokens currently at this place. The initial marking is defined as M_0. With this it is now possible to define

(p,t)

the behaviour of Petri Nets. A transition $t \in T$ can fire in in a Marking M, if $\forall p \in {}^\bullet t : M(p) \geq 0$ and $\forall p \in t_{\text{inhib}} : M(p) = 0$. If an activated transition $t \in T$ fires, it transforms M to M', such that $(\forall p \in {}^\bullet t \setminus t^\bullet : M'(p) = M(p) - 1) \wedge (\forall p \in t^\bullet \setminus {}^\bullet t : M'(p) = M(p) + 1) \wedge (\forall p \in P \setminus (({}^\bullet t \cup t^\bullet) \setminus ({}^\bullet t \cap t^\bullet)) : M(p) = M'(p))$ holds.

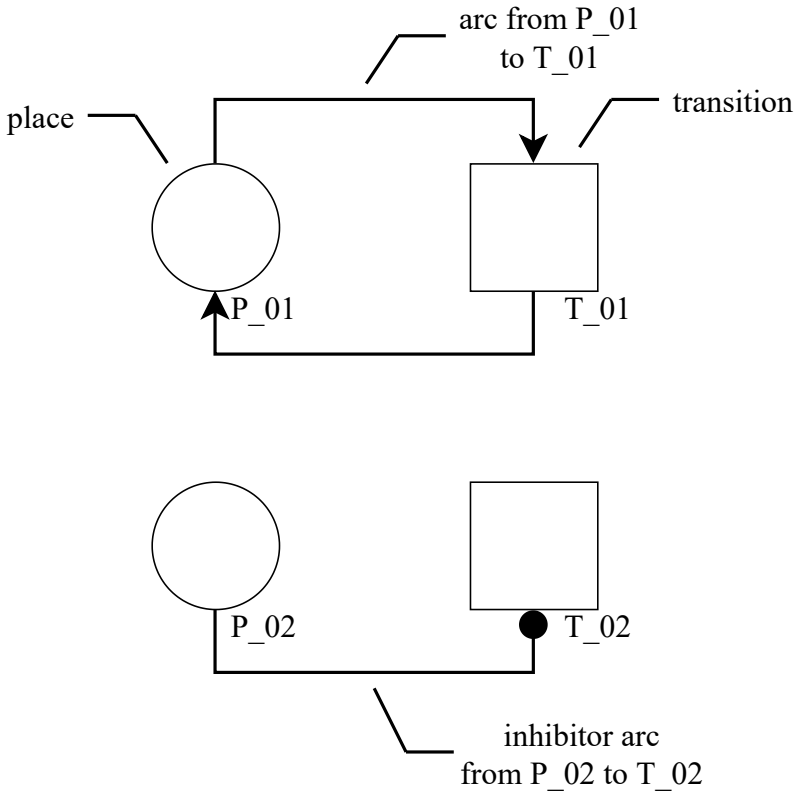
To make larger Petri Nets easier to understand we use different colours for places, transitions and arcs. Besides providing visual aid ,those colours have no further meaning, except grey. A grey place is part of the interface of its Petri Net module. Module interfaces are used to connect two modules with teach other, which is done by merging a place belonging to the interface from one module with an interface place from the other one.

To define the properties of Belt Balancers we use linear temporal logic (LTL) [4] [5]. The unary operator $\Box$ is defined as *always* and $\Diamond$ as *eventually*. Furthermore we use the binary operator $\mathcal{U}$, which represents until. In order for $a \mathcal{U} b$ to hold a has to hold at least until b holds. As soon as b holds, a does not have to hold any more. If b never holds a has to hold forever. Additionally we use $|p|$ for a given place $p \in P$ to receive the amount of tokens in p at the current state. This can be seen as an alternative version to $M(p)$.

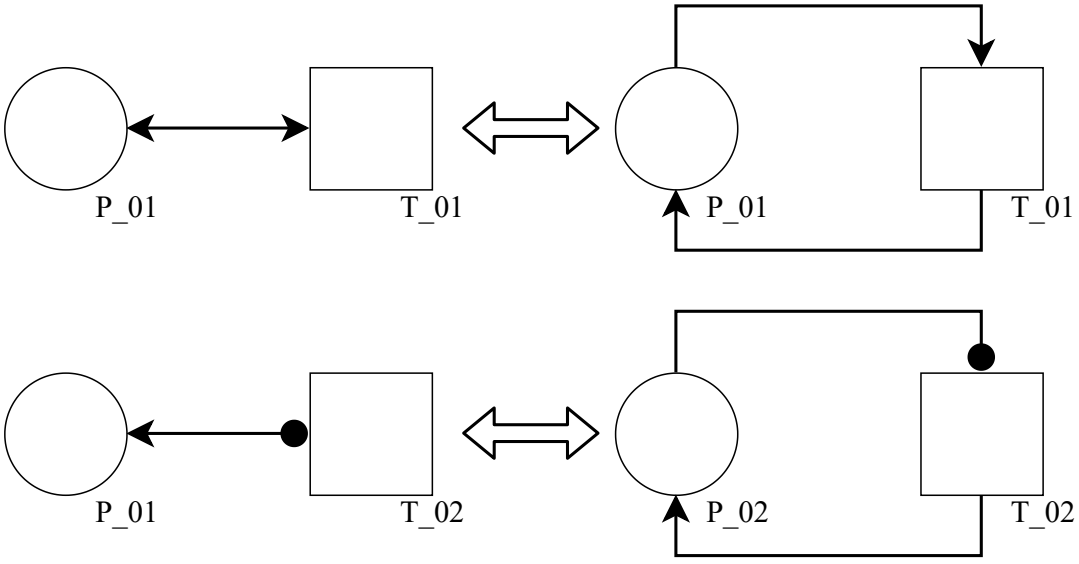
3 Description of Belt Balancers components

Belt Balancers consist of 3 main components, belts, underground belts and splitters. Figure 2a contains an example Belt Balancer. Belts transport items and resources from A to B (see Figure 2b). Underground belts behave like normal belts, but they transport the items underground. With them it is possible to transport resources below buildings or other belts (see Figure 2c). In a Belt Balancer belts and their underground variation are used to connect the inputs and outputs of splitters with each other and to connect the Belt Balancer itself with the outside world. Splitters on the other hand are more complex. They act as universal Belt Balancers with 2 inputs and 2 outputs. Their arrangement determines how the Belt Balancer behaves as a whole. If only one input and one output is connected and used, it behaves like a normal belt. If one input receives items and both outputs pull them (see Figure 2d), the splitter splits the belt and distributes the incoming items evenly between both outputs. This is done by moving incoming items to both outputs in an alternating fashion. Therefore if the top output received an item , the bottom one receives the next one, after which the top output will receive the next item and so on. In case one of the outputs is blocked the splitter behaves like a normal piece of belt again and will transfer all incoming items to the unblocked output, until both are functional again. If both inputs are used, but only one output is connected, the splitter will take an item from its inputs, again alternating between the top and bottom one. The received item is transferred to the connected output. In this configuration items are taken from both inputs evenly. If one of them should be empty the splitter behaves like a normal belt again. If both inputs and outputs are used, the splitter will behave like a combination of the previous two configuration. It will pull items from alternating inputs and move them to alternating outputs. Should one input be empty it behaves like the one input, two output configuration. In case an output is blocked it falls back to the behaviour of the two inputs, one output configuration.

It is possible to set an input or an output priority for a splitter. If for example the top input should be prioritized (see Figure 2e) the splitter will no longer take items from alternating inputs. Instead it will prefer items from the prioritized one. However, if both inputs receive items and both outputs are connected to a belt, a single input can no longer satisfy both output belts. In that case additional items will be taken from the non prioritized input to satisfy both outputs. The output priority works in the same way. If an output is prioritised it will receive all incoming items. If the prioritised output is satisfied, but more items are provided via both inputs, excess items will be transferred to the non prioritised output (see Figure 2f). If an output is



(a) Petri Net with examples for places, transitions and arcs



(b) Shows how double headed arcs are unfolded

Figure 1: Shows how Petri Net components are used

prioritised it may also receive a filter, which means only a specific kind of item may be transferred to that output. However we will not model those filters. Belt Balancers usually handle one kind of items only.

4 First model

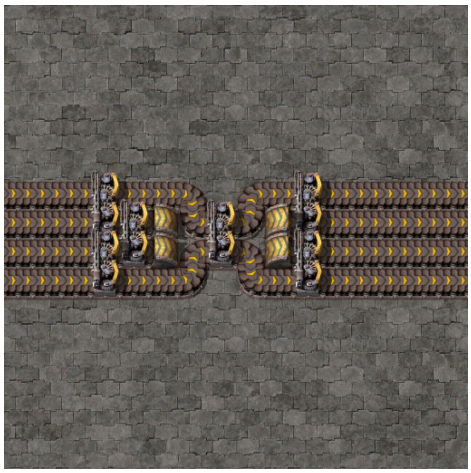
Factorio Belt Balancer consist of belts, underground belts and splitter. But since belts and underground belts only connect the inputs and outputs of two splitter they can be abstracted as edges between them. Therefore the first modelling idea is to create a splitter model with two input and output places, which represent the two inputs and outputs of its original. Later on the output and input places of multiple instances can be connected with transitions, much like splitter in Factorio are connected with each other via belts. With this each original splitter translates to one splitter module and each connection with belts results in a transition between two splitter modules. This makes translating an arbitrary Belt Balancer to its corresponding model rather fast and easy. As we will see this approach is to simplistic. Connecting two splitter with only a single transition leads to several interleaving issues which cause instability and non-deterministic behaviour, which is described in Section 4.5. Fixing those requires more complicated and in depth changes to this modelling approach, therefore this is only the first model of two.

4.1 Non-deterministic model

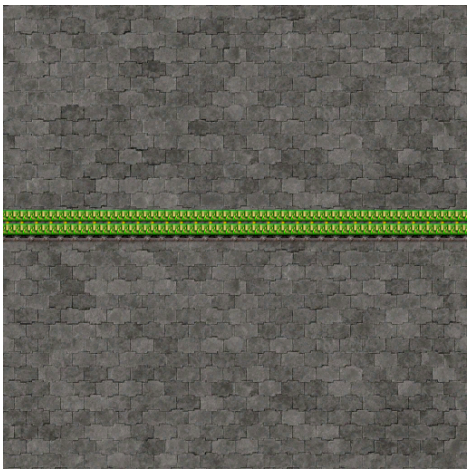
The basic Petri Net for a splitter consist of 2 input places, 2 output places and 4 transitions, which connect each input place with each output place. Additionally 2 cold transitions at the input and two cold transitions at the output are used to model the connection to other splitter modules. They will be removed or modified if the splitter module is used, depending on the connection. Figure 3 provides a visual representation of this basic net. The 4 central transitions model the 4 possible moves a splitter can make, transport an item from the top input to the top output, from top to bottom, from bottom to top and from the bottom input to the bottom output. Since the original splitter is only able to process one item from each input at a time inhibitor arcs are used to ensure one-safety at both inputs. Additionally one-safety is ensured for each output via inhibitor arcs since only one item may be at each output at a time. In contrast to the real splitter this Petri Net moves tokens from the input to the output non-deterministically. For example the current model could consume from IN_0 only which does not represent the original splitter. One could define fairness, but the following extensions provide more control over the firing conditions and provide a model closer to the original.

4.2 Using alternating input / output

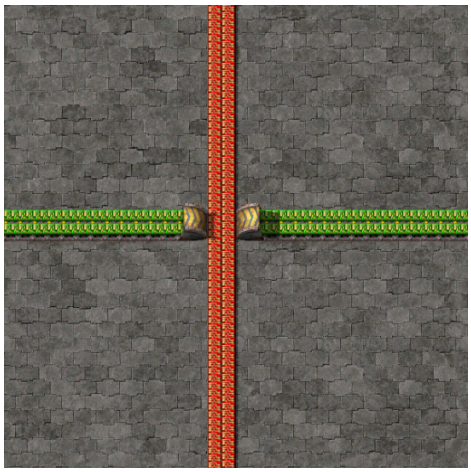
The first extension aims to implement the consumption of items from alternating inputs (red extension, Figure 4a) and the delivery of them to alternating outputs (green extension, Figure 4b). The red extension consists of two places. The first place RED_0 acts as an additional condition for T_00 and T_01 which means a token can only be consumed from the top input if RED_0 contains a token. T_10 and T_11 generate a token at RED_0 if they fire. Therefore an item may only be consumed from the top input after an item has been consumed from the bottom one. The second place RED_1 behaves the same, it enables the consumption from the bottom input and receives a token after an item has been consumed from the top one. Since only one of the red places receives a token during the initial marking the splitter may only consume from one input at a time, and it will consume from them in an alternating fashion.
To implement the same thing for the outputs the green extension (Figure 4b) is used. It behaves similar to the red one. It consists of two places, GREEN_0 enables the generation at the top output and receives a token if T_01 or T_11 has fired. GREEN_1 does the same, just the other way around. It enables the generation at



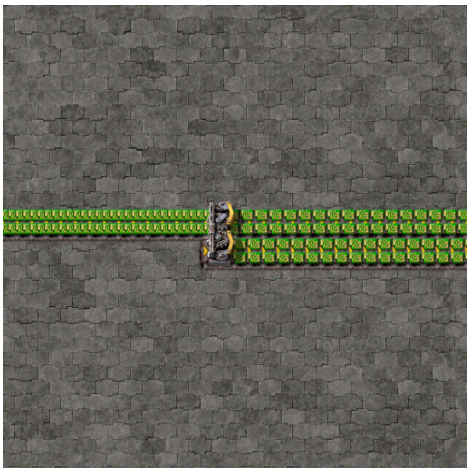
(a) Belt Balancer example with 4 inputs and 4 outputs



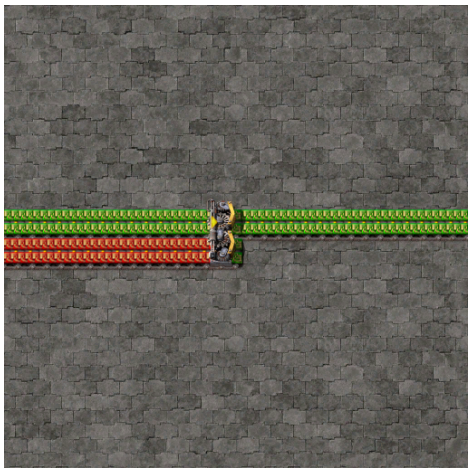
(b) A belt moving green circuits from left to right



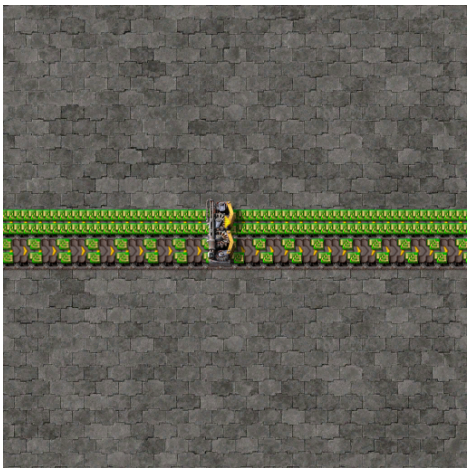
(c) An underground belt moving green circuits crosses a belt with red circuits



(d) A splitter splits a belt with green circuits into two belts with half the amount of items each



(e) A splitter with input priority at the top: Red circuits from the bottom are not moved to an output



(f) A Splitter with output priority at the top: Excess circuits are moved to the bottom output

Figure 2: Examples of how different components are used in Factorio

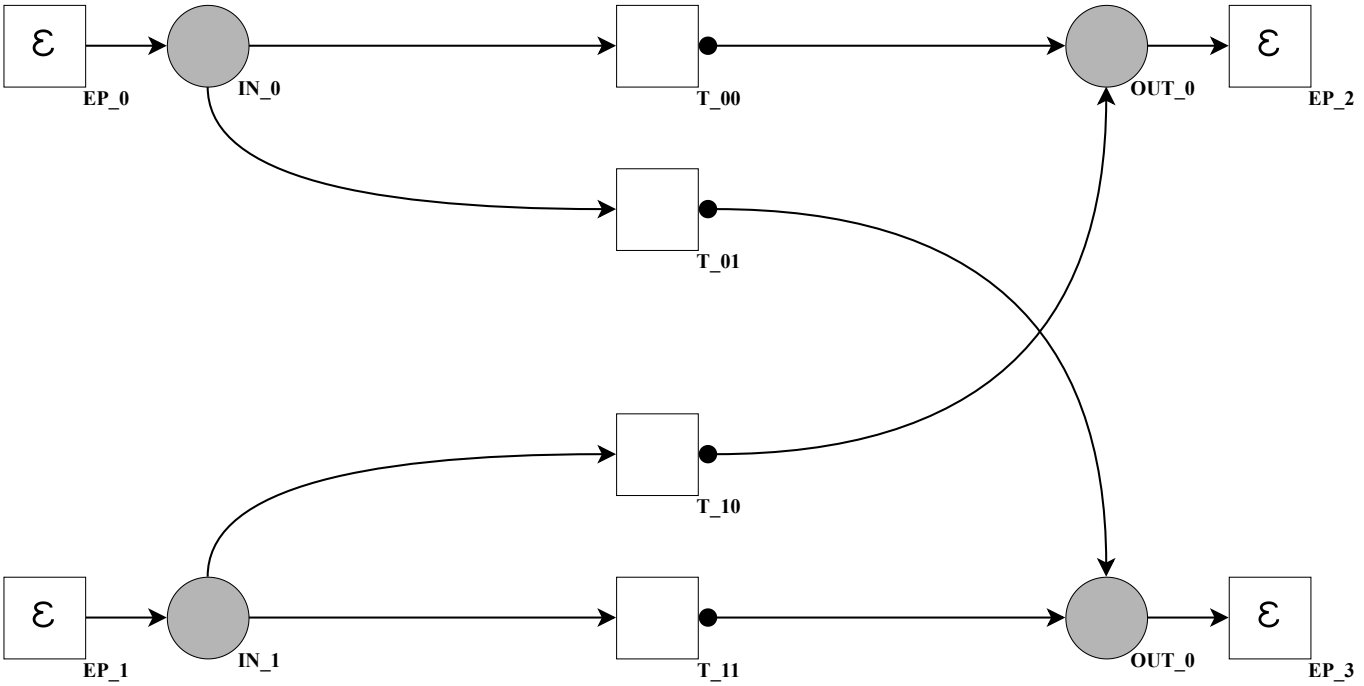


Figure 3: Skeleton of first splitter model as Petri Net

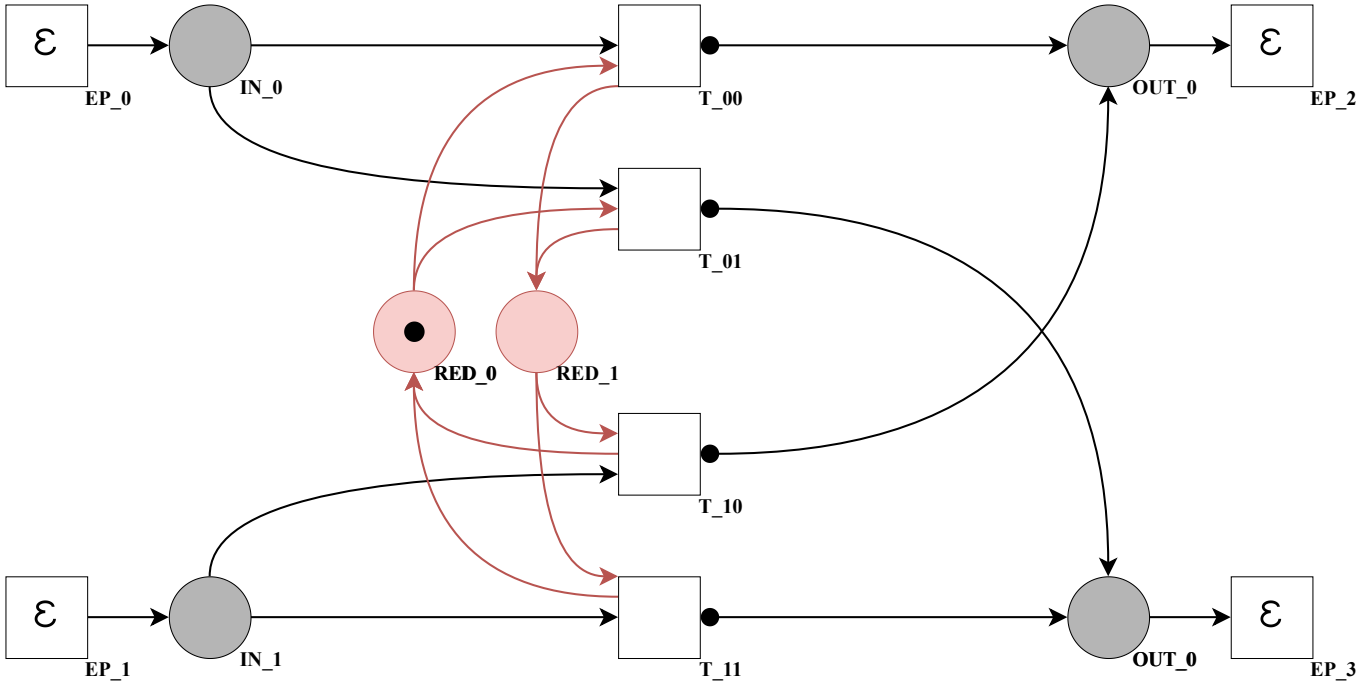
the bottom output and receives a token after T_00 or T_10 has fired.

Those two extensions provide the ability to consume tokens from alternating inputs and generate them at alternating outputs. But in the current state the Petri Net cannot handle an empty input or a blocked output indefinitely. Assume the initial marking in Figure 5a. If a token arrives at the input IN_0 (Figure 5b) the transition T_00 will transfer it to the top output OUT_0 (Figure 5c). This causes the tokens in the red and green extension to switch places which enables transition T_11 only. Now another token has been generated at the top input (Figure 5d). Since the red token switched places the new token cannot be transferred to the bottom output, but the original Factorio splitter would do so. Therefore the current Petri Net is insufficient and needs to be extended further.

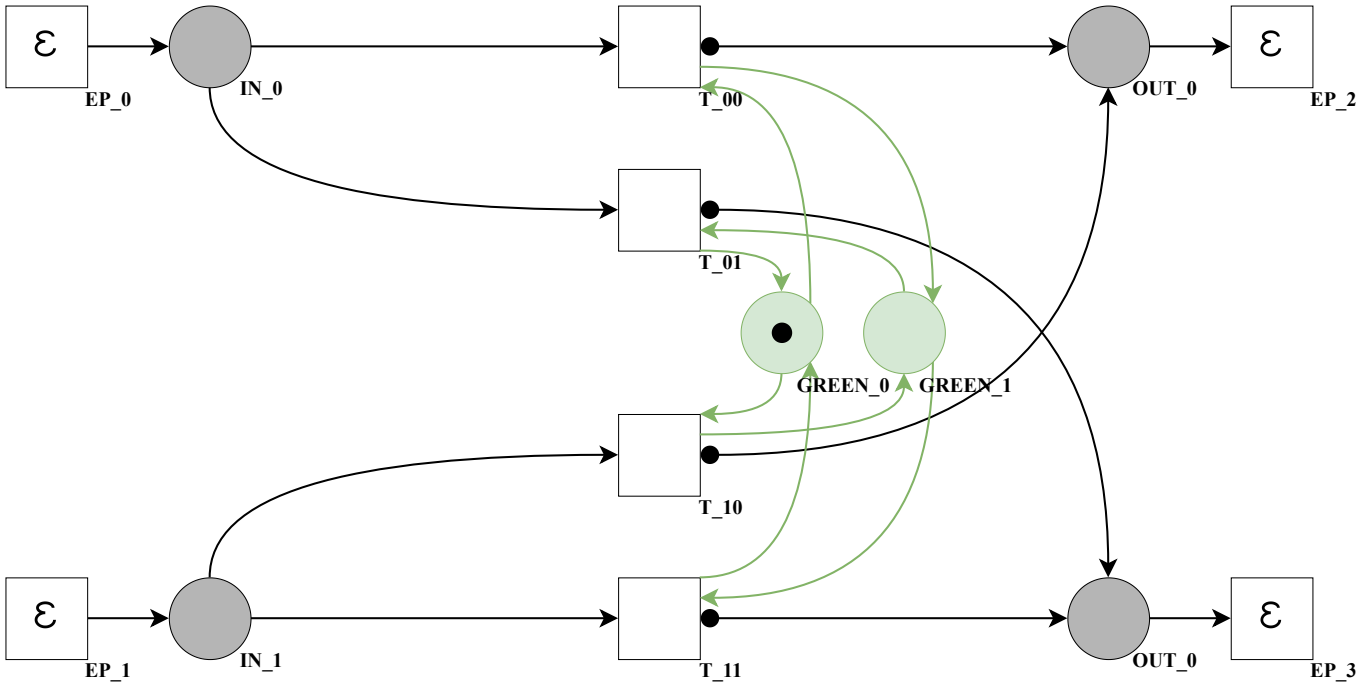
4.3 Handling of empty inputs / outputs

To handle empty inputs 2 transitions are added, one for each input. Those transitions are part of a new, orange extension (Figure 6a). ORG_0 and ORG_1 handle the case for an empty input by moving the token from the red extension if necessary. For example if the bottom input is empty but a token should be moved from there (RED_1 contains a token) ORG_1 moves the red token from RED_1 to RED_0 to enable the consumption from IN_0. To prevent unnecessary movement of the red token, ORG_1 also checks whether IN_0 contains a token or not. If there is no item to move, it would make no sense to enable the consumption from there. The same applies for ORG_0. It covers the case for an empty input at the top while the bottom one contains a token. The handling of a full output works the same and is implemented by the blue extension (Figure 6b). If OUT_1 contains a token, but the green extension only enables the transfer to this blocked output, BLUE_1 moves the green token if OUT_0 is empty. The same applies for BLUE_0.

If all extensions, i.e. the red, green, orange and blue one, are applied to the basic Petri Net they model the

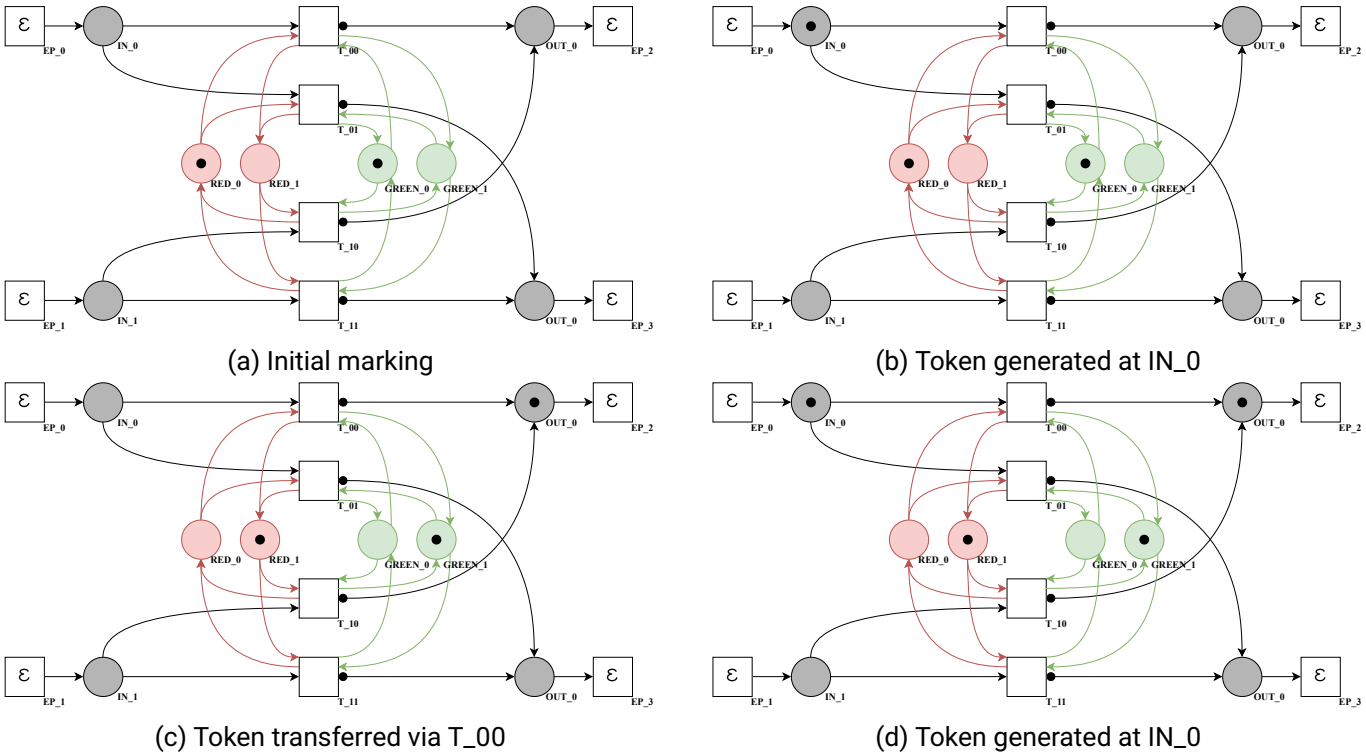


(a) Red extension controlling used input



(b) Green extension controlling used output

Figure 4: Additional red and green extension



(a) Initial marking

(b) Token generated at IN_0

(c) Token transferred via T_00

(d) Token generated at IN_0

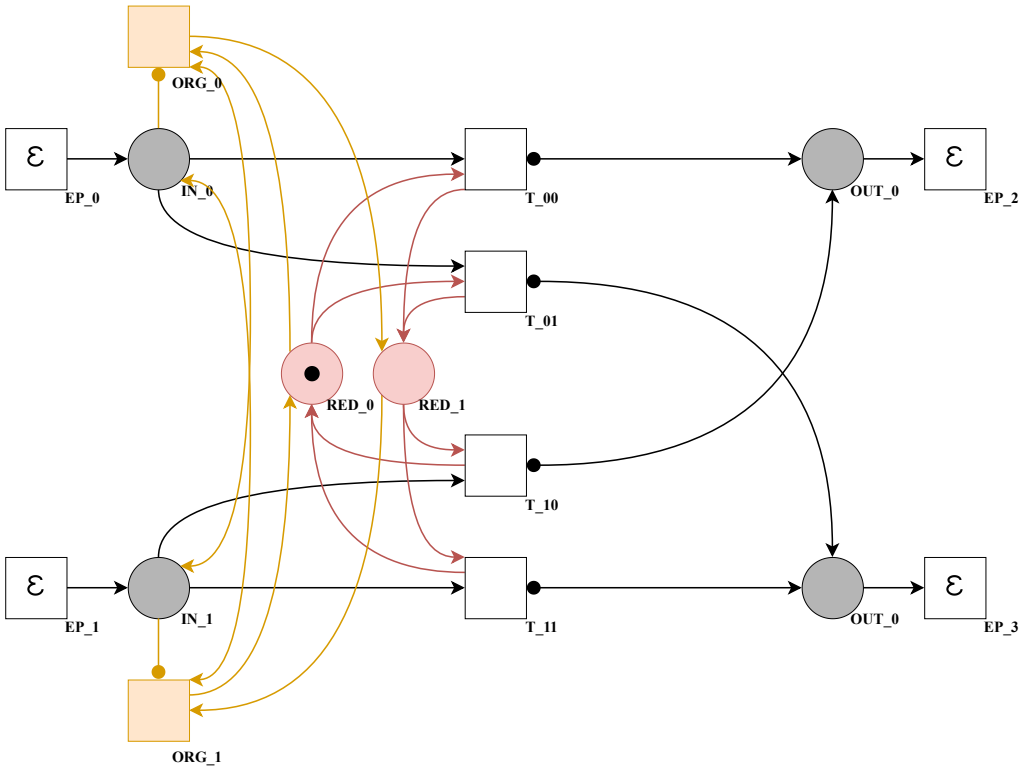
Figure 5: Example for a stuck Petri Net

behaviour of a Factorio splitter accurately. To give an example we assume the same situation as in 4.2. After the initial marking in Figure 7a IN_0 receives a token (Figure 7b) which is transfered to OUT_0 by T_00 (Figure 7c). Another token as been generated at IN_0 (Figure 7d) but, with the orange extension, ORG_1 can handle this case by moving the token from RED_1 to RED_0 (Figure 7e). Now T_01 can transfer the token to OUT_1 (Figure 7f).
But one problem remains, the current Petri Net does not support input and output priorities. To add this feature more extensions are needed.

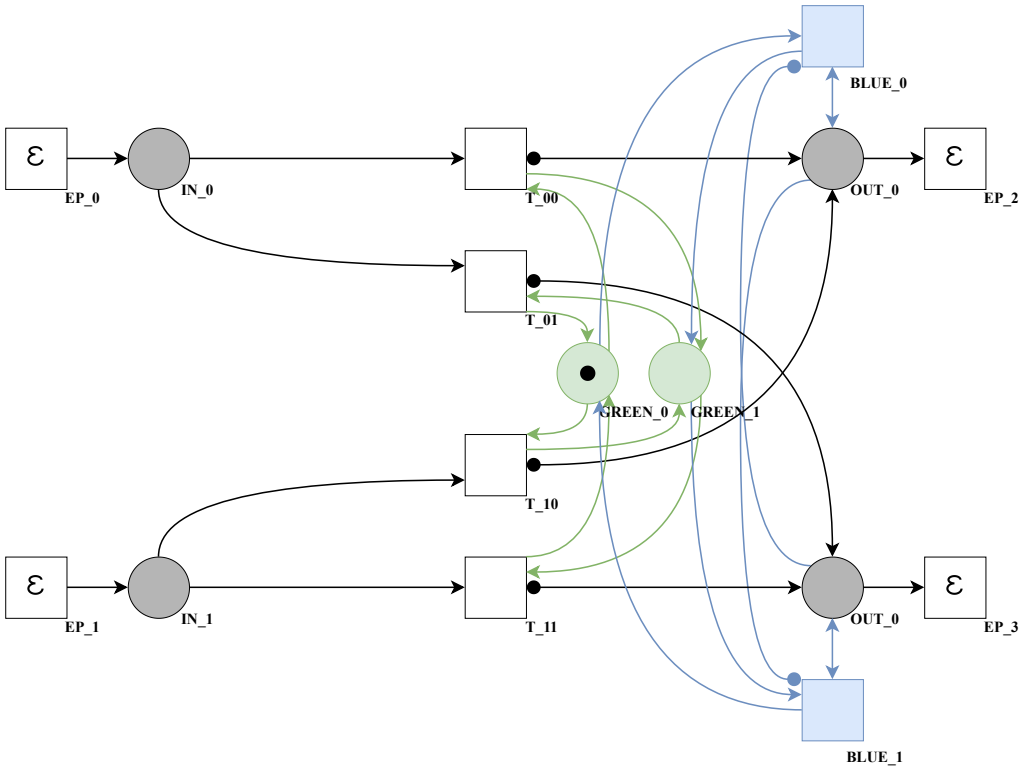
4.4 Enforcing of input / output priorities

For implementing input priorities the turquoise extension is introduced. First, we add an additional place to each input (Figure 8) which represents the priority of that place. If P_IN_0 contains a token the top input has priority, the same applies for P_IN_1 with IN_1. If no input should be prioritised both priority places have to contain a token. To actually enforce the priority we need an additional place to control the central transitions. To do this P_IN_0_EN and P_IN_1_EN are added. They enable the central transitions if they contain a token. The transitions T_P_IN_0 and T_P_IN_1 generate this enable token if the respective priority place contains one. Because of this the behaviour of the Petri Net is equivalent to the version without priority if no priority is set (both priority places contain a token). If e.g. IN_0 is prioritised T_00 and T_01 always receive an enable token via T_P_IN_0. But right now a priority is not enforced. If IN_0 is prioritised and it contains a token, the token is moved to an output. If another token arrives at IN_0 it is not moved immediately since RED_0 does not contain a token.

To move the token from the red extension to the right place two additional transitions are added (Figure 9),

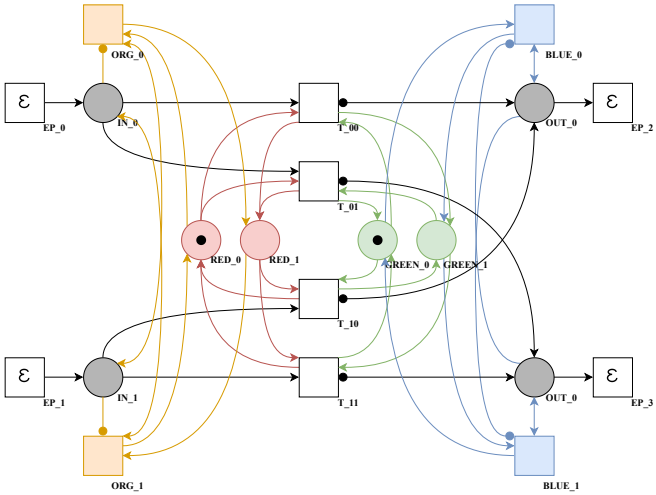


(a) Orange extension handling empty inputs

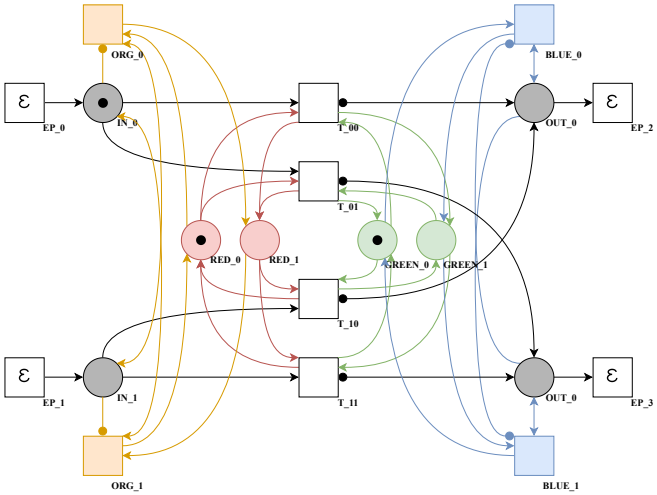


(b) Blue extension handling full outputs

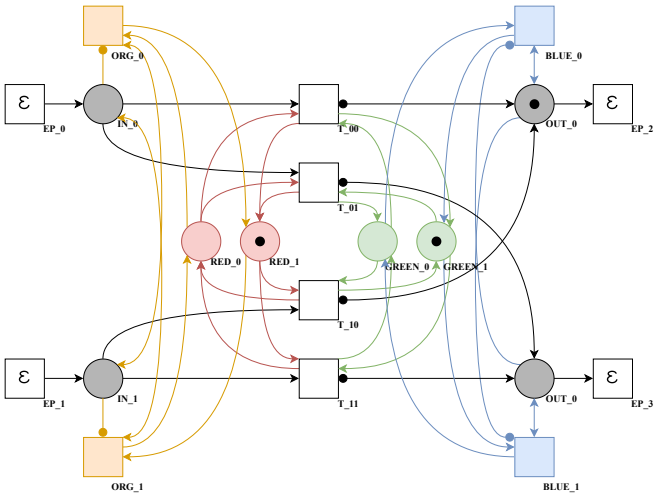
Figure 6: Additional orange and blue extension to handle input / output anomalies



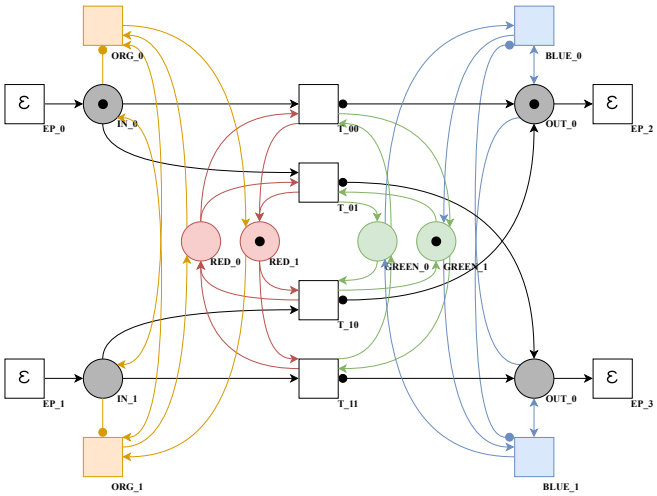
(a) Initial marking



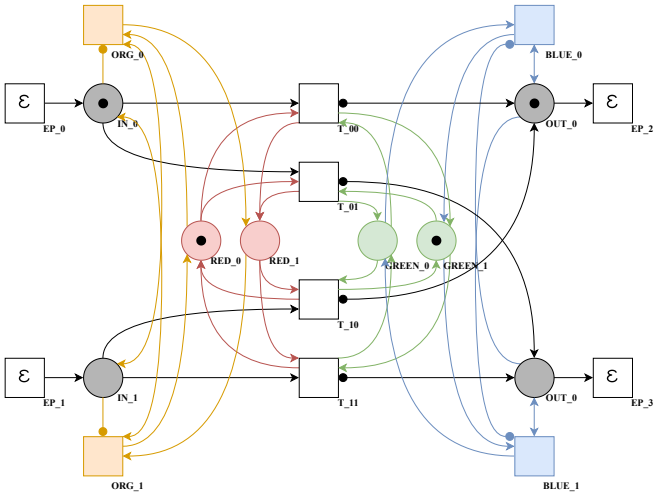
(b) Token generated at IN_0



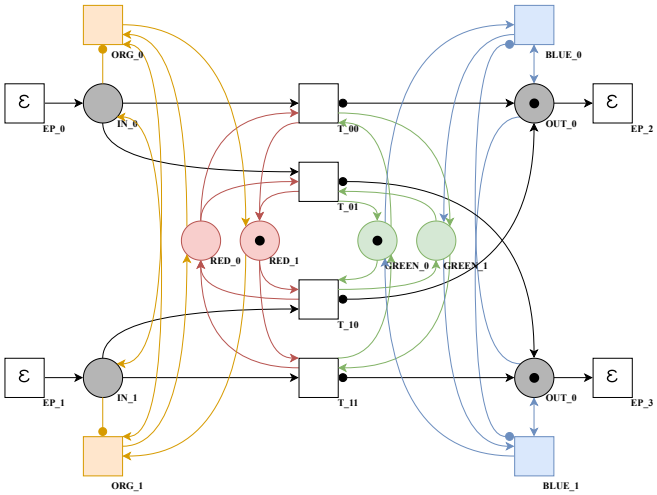
(c) Token transferred to OUT_0 by T_00



(d) Another token generated at IN_0



(e) ORG_0 moves token from RED_1 to RED_0



(f) Token transferred from IN_0 to OUT_1 by T_01

Figure 7: Example run of current Petri Net

P_IN_0_ENF and P_IN_1_ENF. They enforce the priority by moving the red token if the respective input contains an item to move, if the input is prioritised and if the red token needs to be moved. In case of P_IN_1_ENF the transition fires if IN_1 and P_IN_1 contain a token, if P_IN_0 is empty and if the red token is at RED_0 and therefore enables the transitions for the top input. By firing it moves the token from RED_0 to RED_1 which enables transitions for the bottom input. Now only one problem remains, if e.g. the top input has priority none of the transitions for the bottom input may ever fire.

The handling of empty, prioritised inputs is done by 2 new transitions (Figure 10). P_IN_0_EMP fires if IN_0 is empty and has priority, if IN_1 contains a token and has not priority and if P_IN_1_EN is empty. This creates a token at P_IN_1_EN and enables T_10 and T_11. The same applies for its bottom counterpart P_IN_0_EMP. This allows the unprioritised input to fire once, if the prioritised input is empty. An example run with prioritised input is shown in Figure 11.

Output priorities are handled by the lime extension (Figure 13) which is pretty much the same as the turquoise one. There are some minor differences since the edge cases occur if the output is full instead of empty. For example P_OUT_0_ENF does not checks whether OUT_0 contains a token, but fires if this output is empty.

4.5 Problems with the current model

The current Petri Net with all its extension has a couple of problems. First, the way it handles incoming items is still not accurate if compared to its original from Factorio. In Factorio both items may receive items simultaneously and to do the same our model has to perform two steps. In step one the top input IN_0 receives a token and in step two the bottom input IN_1 receives its token. After both tokens have been generated the Petri Net may start to distribute them according to its logic. But due to the non-deterministic nature of Petri Nets those items do not have to arrive in two consecutive steps. If the token arrives at IN_1 the Net could interleave and move this token to an output before the second one has been generated. This may seem like a cosmetic issue in terms of item distribution at first, but especially with the addition of priorities it also results in unstable behaviour. To give an example: Assume the initial marking of a splitter with prioritised top input (Figure 14a). To improve readability the output related extensions are not shown in further graphs. As the first step the transition T_P_IN_0 creates a token at P_IN_0_EN (Figure 14b) to enable both central transitions related to the top input. Now an item arrives at the top input, which means a token is generated at IN_0 (Figure 14c) which enables the central transitions to move it to an output (Figure 14d). With this transfer the token from the red extension has been moved as well. Now the bottom input IN_0 generates receives a token (Figure 14e) and to be able to move this token P_IN_0_EMP creates an enable token at P_IN_1_EN (Figure 14f). In an ideal scenario the token at IN_1 would be moved to an output next, but lets generate a token at IN_0 first (Figure 15a). The next step should be to enforce the top priority by moving the red token, generating the enable token and moving it. But since P_IN_1_EN still contains an enable token the bottom one could be moved or it could wait and move the next time a token arrives at the top input. This makes the Petri Net as a whole non-deterministic and therefore needs to be fixed. This could probably be done with yet another extension however stacking extension over extension rapidly decreases the readability of the Petri Net. Additionally it increases the complexity of the net which may result in more interleaving issues or other bugs. Because of this the current modelling idea will be revisited to find a more elegant way to solve those issues.

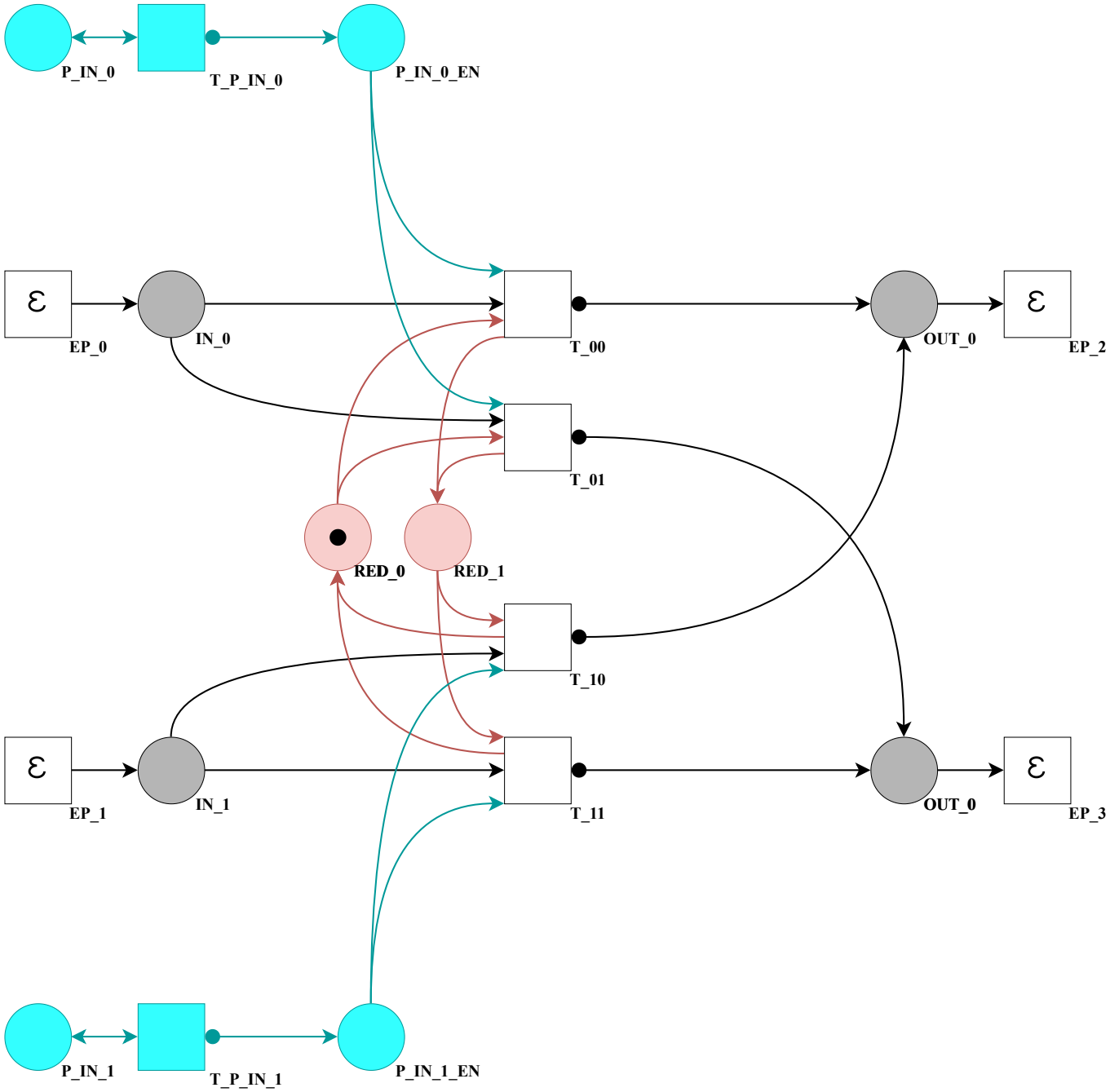


Figure 8: Adding priority and places to control central transitions

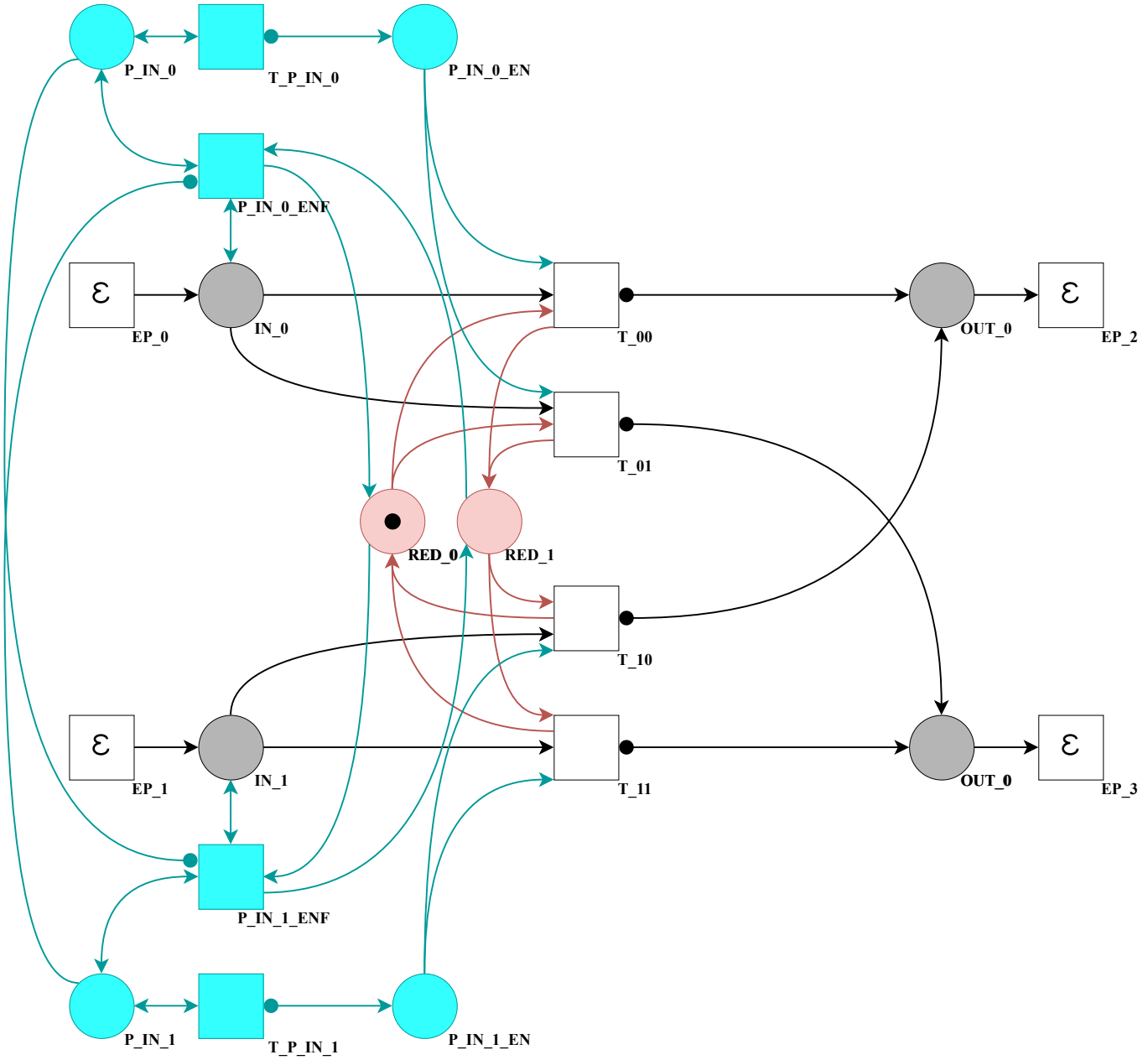


Figure 9: Handling mispositioned token from the red extension

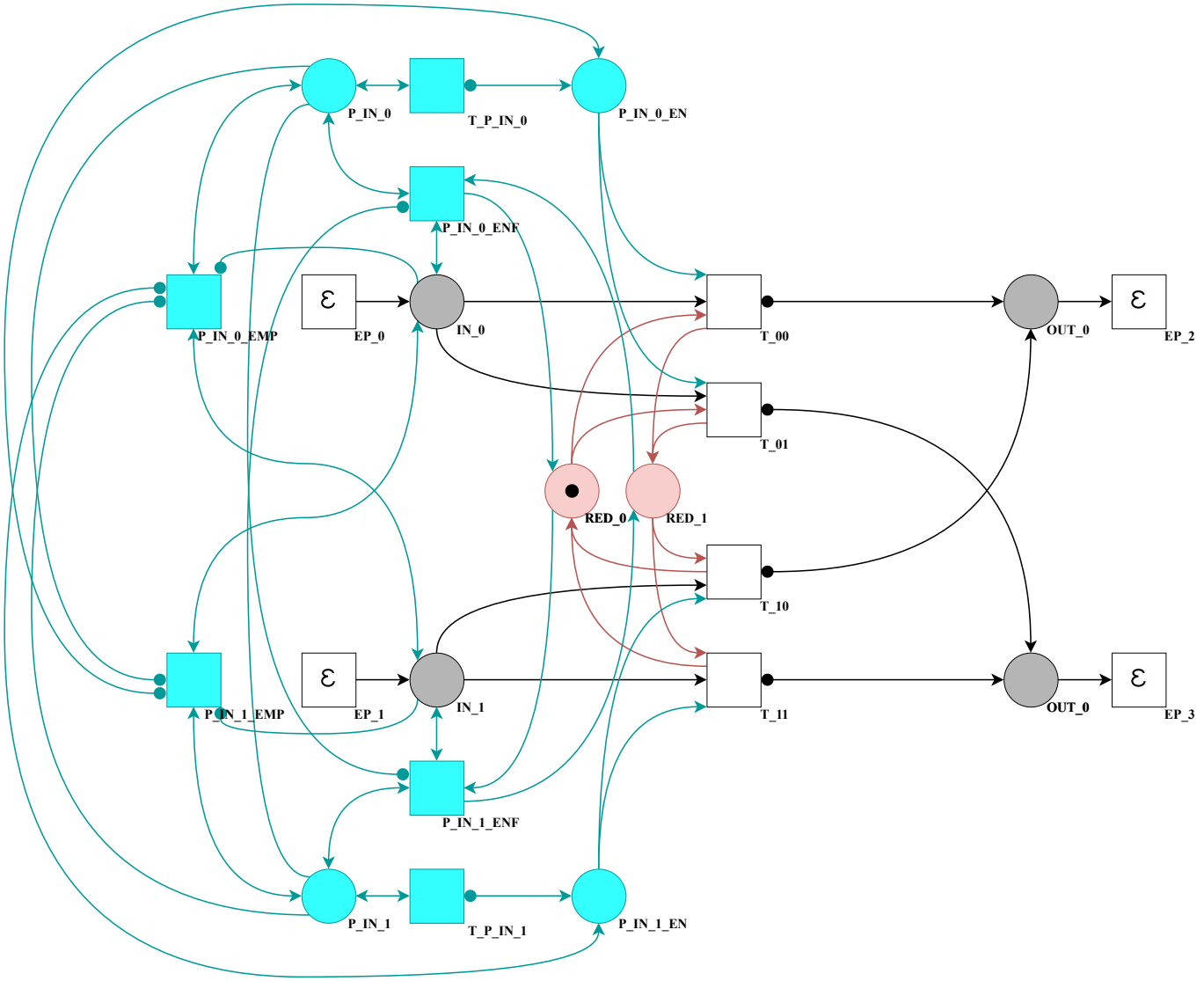
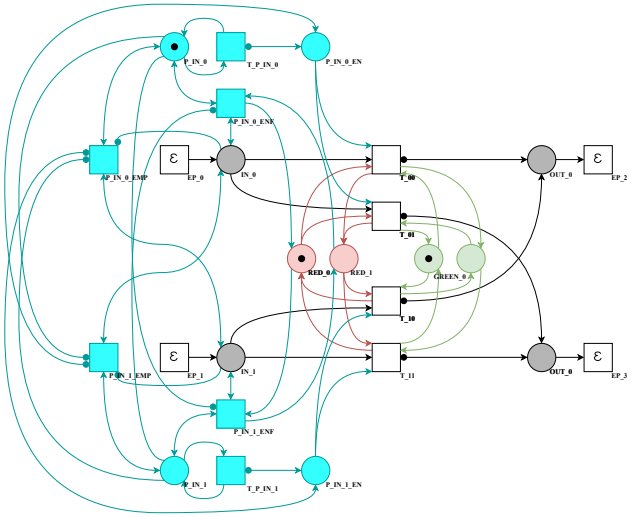
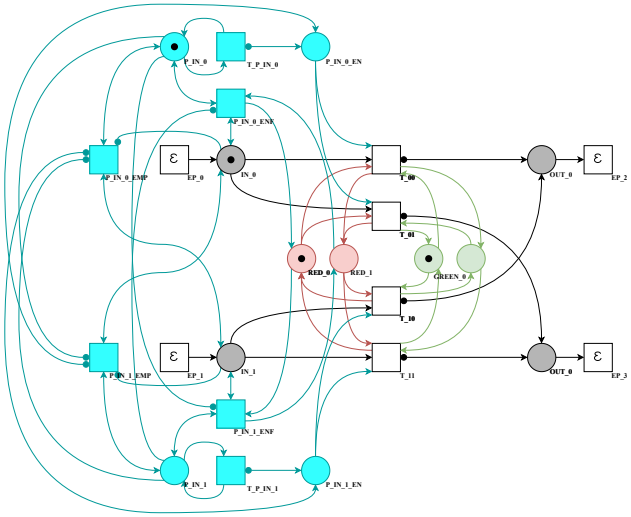


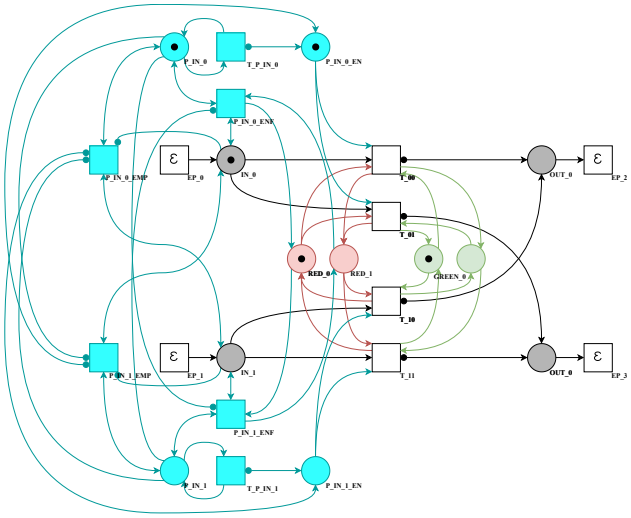
Figure 10: Handling empty, prioritised inputs



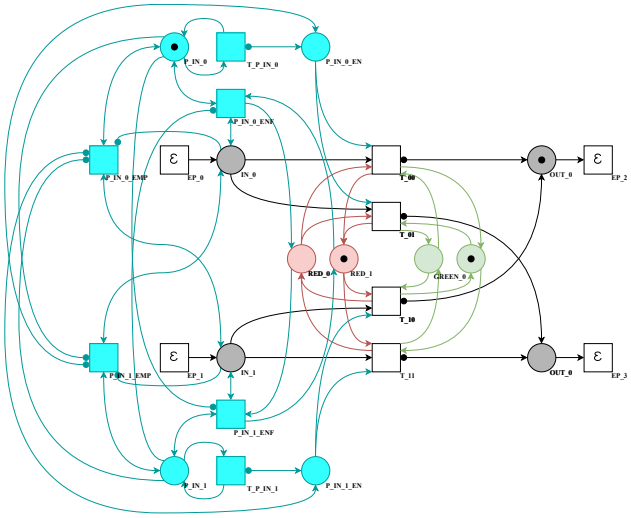
(a) Initial marking



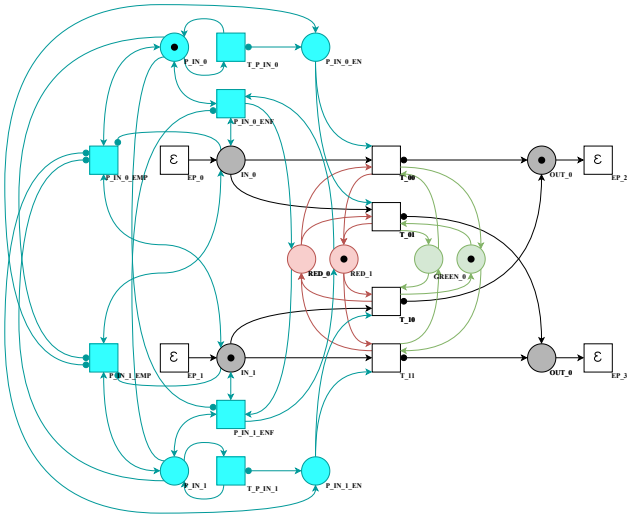
(b) Token generated at IN_0



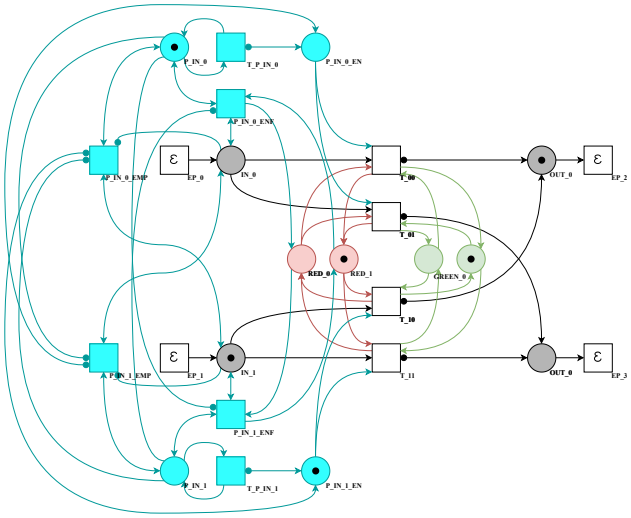
(c) Token generated at P_IN_0_EN



(d) Token transferred from IN_0 to OUT_0

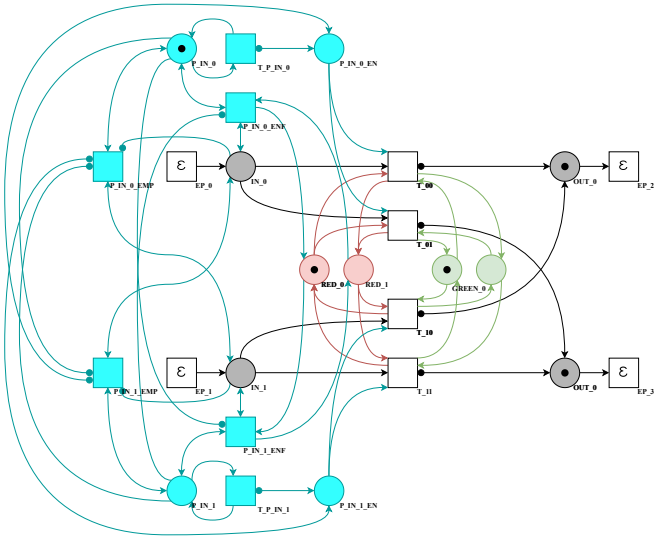


(e) Token generated at IN_1



(f) Token generated at P_IN_1_EN by P_IN_0_EMP

Figure 11: Example run of current Petri Net. Continued at Figure 12



(a) Token moved from IN_1 to OUT_1

Figure 12: Continuation of Figure 11

5 Second model

This approach aims to add more control over the components of a splitter. Instead of allowing each splitter module to perform a step whenever an item arrives at its input it should start processing the tokens only if it is certain that all splitter before it are finished. To receive this information from its predecessor some sort of communication needs to be implemented between splitter modules. To achieve this more modules are added.

5.1 Used modules

All used modules are divided into two categories, productive modules and transfer modules. Productive modules represent some kind of move or action of the Belt Balancer. This category consists of 3 modules. First the item generator. It represents an input of the Balancer and to do so it generates tokens which will be distributed by the rest of the Petri Net. To model a Belt Balancer output the item consumer is used. This module consumes the tokens distributed by the net work of splitter modules. The splitter module itself belongs to this category too. Each splitter module represents a single splitter in the Belt Balancer.
The second category describes the transfer modules. Their task is to manage the communication between productive modules. There are 5 possibilities to connect two productive modules. It is possible to connect an item generator with an item consumer or splitter, a splitter can be connected to another splitter or an item consumer and all item consumers are connected to all item generators. The item transfer module handles all of those cases except the connection between all item consumers to all item generators. This last case is covered by the item loopback module. It connects all item consumer with all item generators and therefore closes the loop. Additionally to those two more functional modules the input dummy and output dummy belong to this category as well. They model an unused input or output of a splitter without generating or consuming any tokens. An example containing all possible connections and modules is shown in Figure 29.

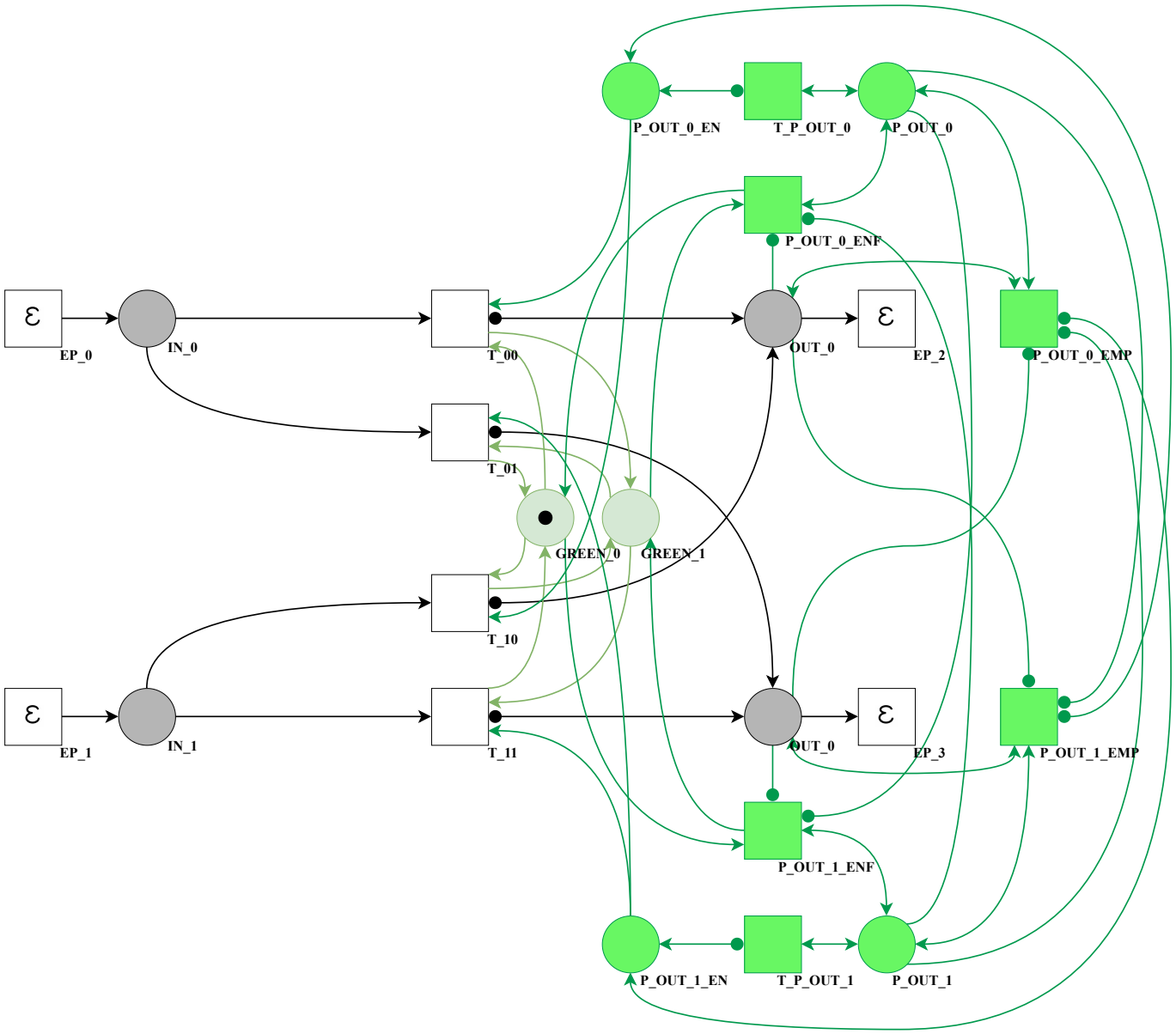
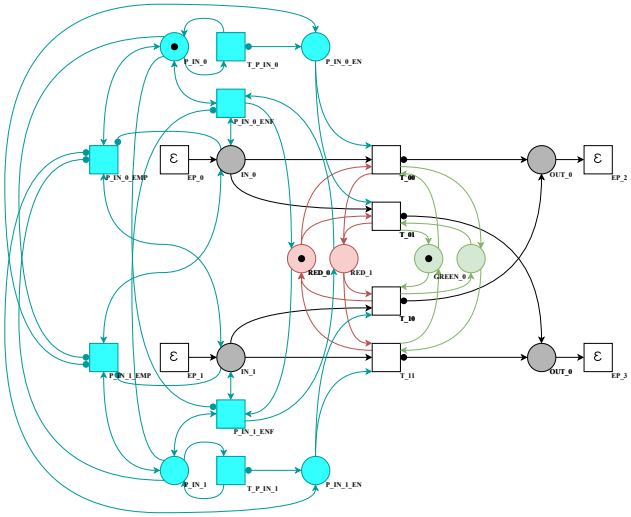
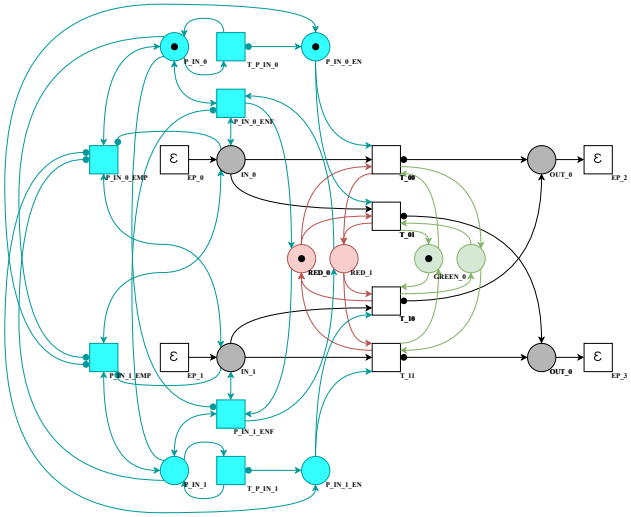


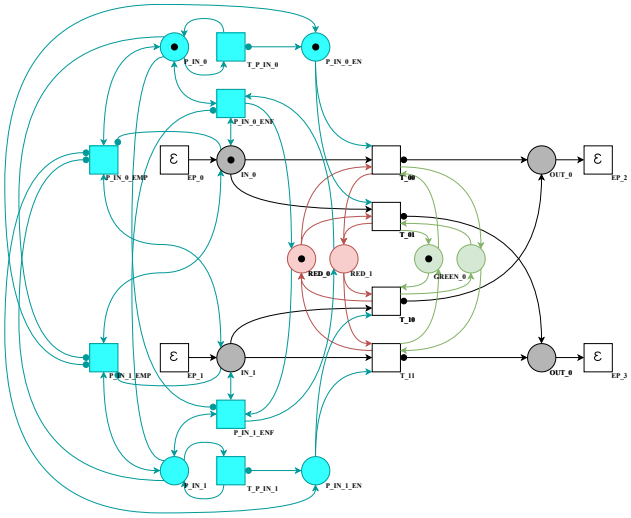
Figure 13: Lime extension to implement output priorities



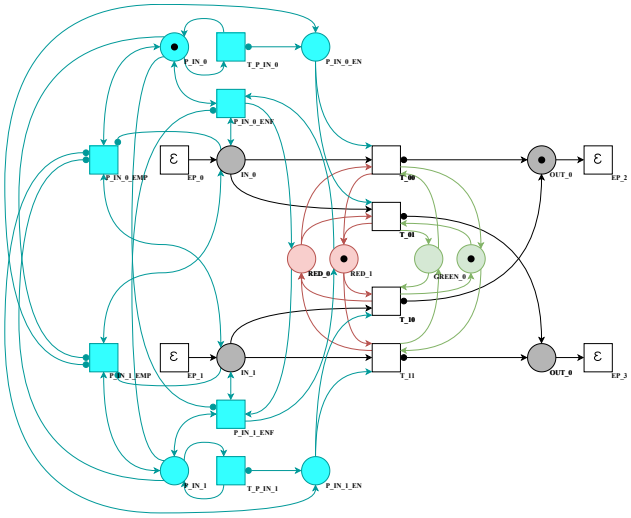
(a) Initial marking



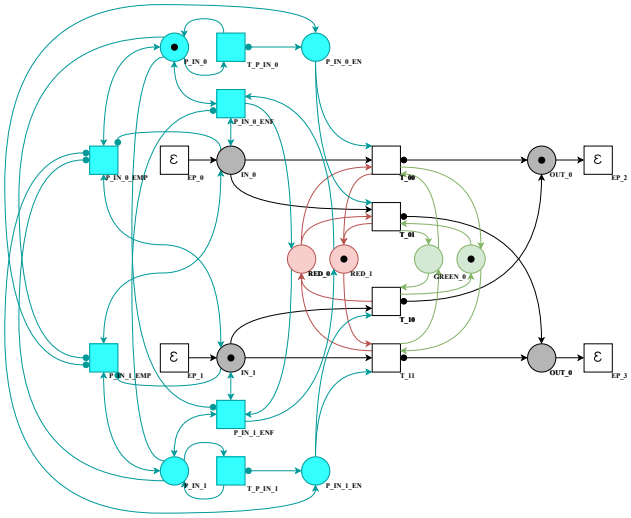
(b) Token generated at P_IN_0_EN



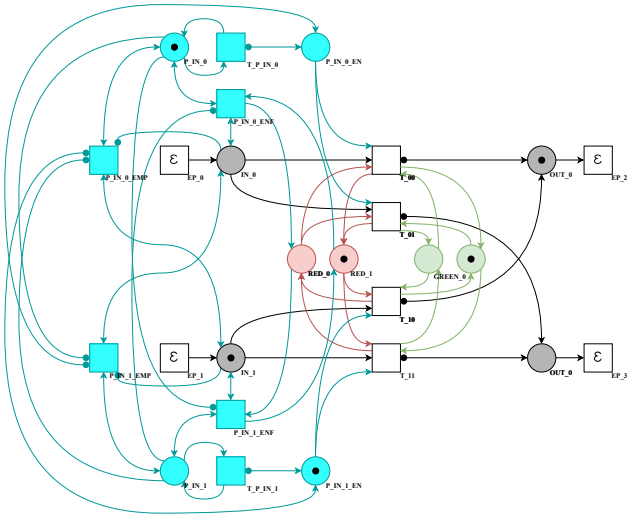
(c) Token generated at IN_0



(d) Token transferred from IN_0 to OUT_0

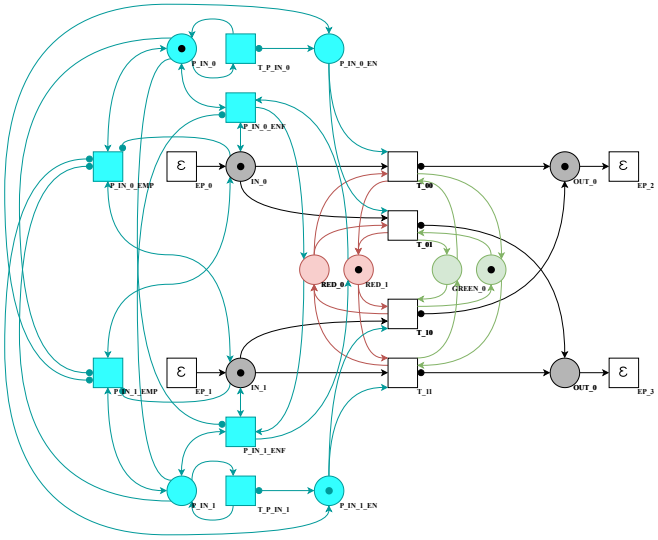


(e) Token generated at IN_1



(f) Token generated at P_IN_1_EN by P_IN_0_EMP

Figure 14: Example run of current Petri Net



(a) Token generated at IN_1

Figure 15: Continuation of Figure 14

5.2 Communication between modules

The communication between two modules is inspired by the handshake used in the AXI interface [6]. This interface is usually used to provide parallel, synchronous and high performance communication between multiple master and slave modules on an on-chip level. But as already mentioned this model will not implement a complete AXI interface, but instead use a modified version of the ready/valid handshake to transfer information between two modules only if both sides are ready. Figure 16 provides a visualisation of this modified version which is explained by the following Section in detail.

Each input and output port consists of 3 places and represents an interface for other modules. Those places are VALUE, VALUE_RDY (value ready) and VALUE_VAL (value valid). Ports are always used to connect the output port of one module with the input port of another one by merging the places belonging to those ports. Therefore the VALUE place of the input port is merged with the VALUE place of the output port and the same applies to both VALUE_RDY places and VALUE_VAL places.

There are two possibilities for he initial marking, depending on whether a transfer module output is connected with a working module input or a working module output with a transfer module input. In the first case the handshake starts with state (a). The token at VALUE_RDY signals the output module that the input module is ready to receive new information via VALUE. But at this point the output module may not create any tokens in VALUE_RDY or VALUE_VAL, it needs to consume a token from VALUE_RDY first. Additionally the input module may not revoke the ready token (the token from VALUE_RDY). Once it has been created it is final and the module has to wait until it received a token from VALUE_VAL.
In state (b) the output module consumed the ready token and is now working internally to create the data which will be generated at VALUE. In our case the module determines whether it generates a token at VALUE or not.

After it is finished, it creates the data at VALUE and generates a token at VALUE_VAL to ensure the validity of the data. This allows the transfer of an ”empty token”, meaning the information that no token has been

generated. Additionally the output module may neither revoke the valid token (token in VALUE_VAL) nor modify VALUE as soon as it creates the token in VALUE_VAL. Furthermore it is possible that VALUE already contains a token, because the input module did not consume the data from a previous transfer. In that case the output module may not generate another token at this place. Instead it has to create just the token in VALUE_VAL.

With a token in VALUE_VAL, the input module may now consume the valid token from VALUE_VAL and the information from VALUE, but the consumption of the content of VALUE is optional. If for example a splitter module is not able to distribute the token it may just stay in VALUE.
Eventually the input module creates a token at VALUE_RDY again to start the cycle again.
As noted above there is a second possibility. If the input module belongs to the category of working modules and the output one to transfer modules the handshake starts at (c) with no tokens in VALUE, but the constrains and steps are the same.
It is important to note that none of the modules have to add or remove a token from VALUE, but they have to perform their consume and generate operations on VALUE_RDY and VALUE_VALID as described above always eventually.

5.3 Module internals

In the following Section the functionality of each module is explained. Additionally their usage is described more in depth in comparison to Section 5.1.

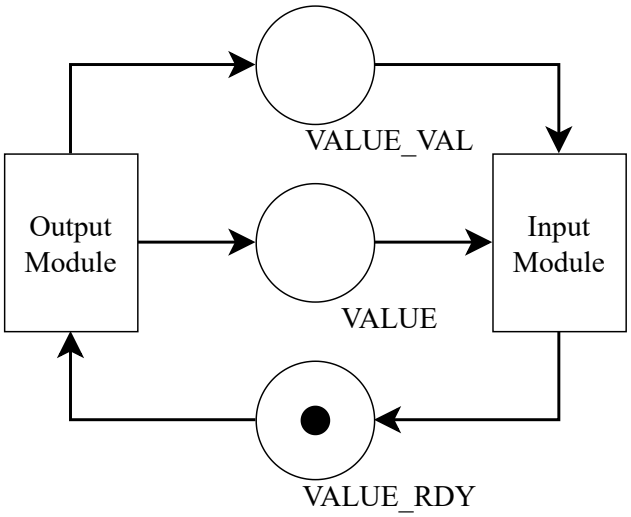
5.3.1 Item generator

Each item generator module models a single input of the Belt Balancer. It consists of one input and one output port, but the value_place from the input port is not used. The place only exists to keep the interface consistent between all module types. Figure 17 shows the Petri Net of this module.

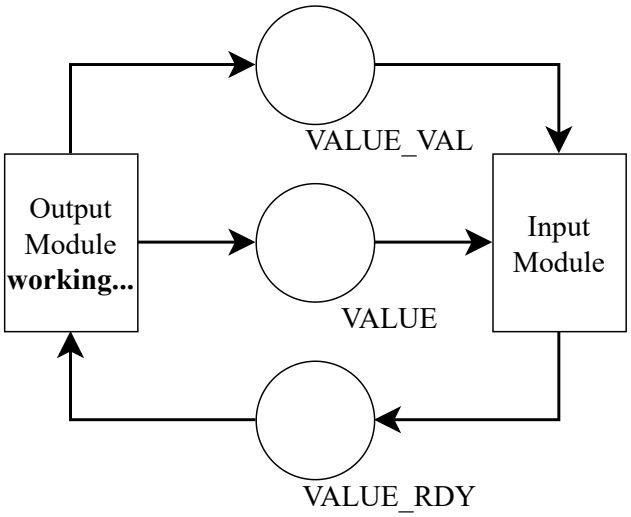
The core structure of this module consists of the transitions T_GENERATED, T_NOT_GENERATED and the places belonging to the input and output port. The idea is to let the module decide non-deterministically whether it generates a token at the output or not. This is achieved by firing one of those two transitions once. If T_GENERATED fires it creates a token at the output, T_NOT_GENERATED does not. To ensure that only one of them fires, both are enabled by the place LOCAL_EN. Only one token will be generated at this place and therefore only one of the transitions can fire. T_IN generates this token together with the token at GLOBAL_EN. To do so the transition requires a token at IN_VAL and OUT_RDY, it waits until the input value is valid and the module at the output is ready to receive a new value. Only if both tokens are present it consumes them and generates the tokens for GLOBAL_EN and LOCAL_EN.

If either T_GENERATED or T_NOT_GENERATED consumed the token from LOCAL_EN the transition T_OUT may fire. This transition creates tokens at IN_RDY, to signal the predecessor the module is ready to receive a new value, and OUT_VAL, to mark the output value as valid. Since LOCAL_EN is empty most of the time, T_OUT would generate those tokens permanently. To prevent this, it also requires the token from GLOBAL_EN. With this it fires only if the module is currently working. After T_OUT fired the module waits until T_IN fires and therefore enables it again.

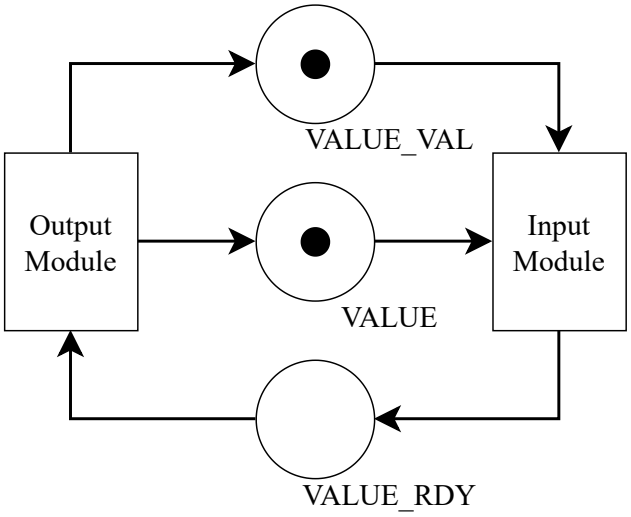
In an abstract way, the module has a working and an idle state. If GLOBAL_EN contains a token the module is working. If GLOBAL_EN is empty, the module is idle and waiting for an input. T_IN and T_OUT serve as



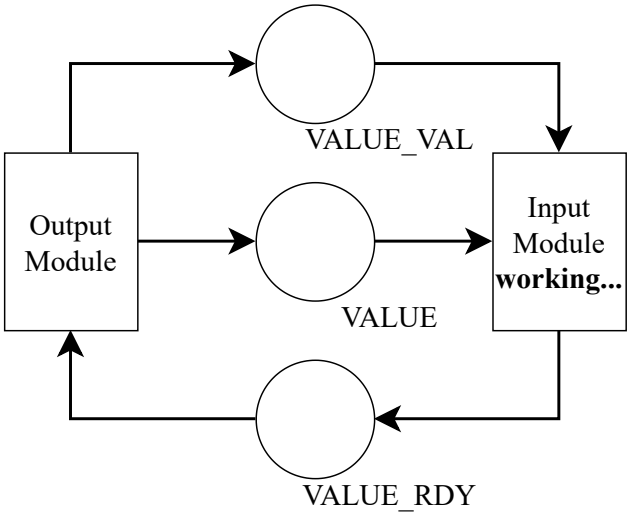
(a) Initial marking



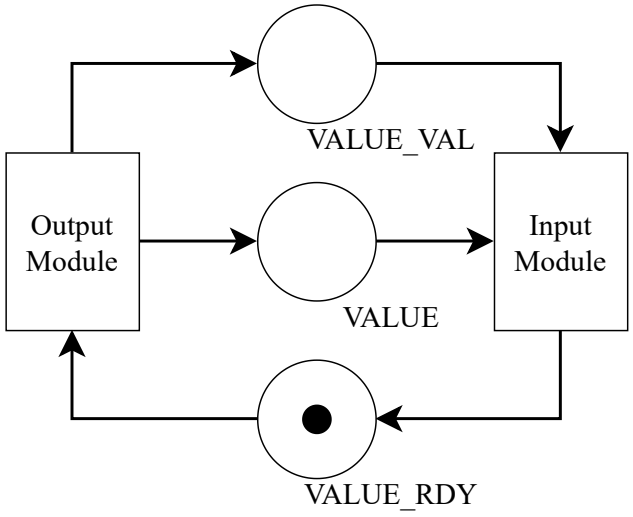
(b) Output module consumes token from VALUE_RDY



(c) Output module generates token in VALUE_VAL and potential data in VALUE



(d) Input module consumes potential data and token from VALUE_VAL



(e) Input module generates token in VALUE_RDY

Figure 16: Handshake between two modules

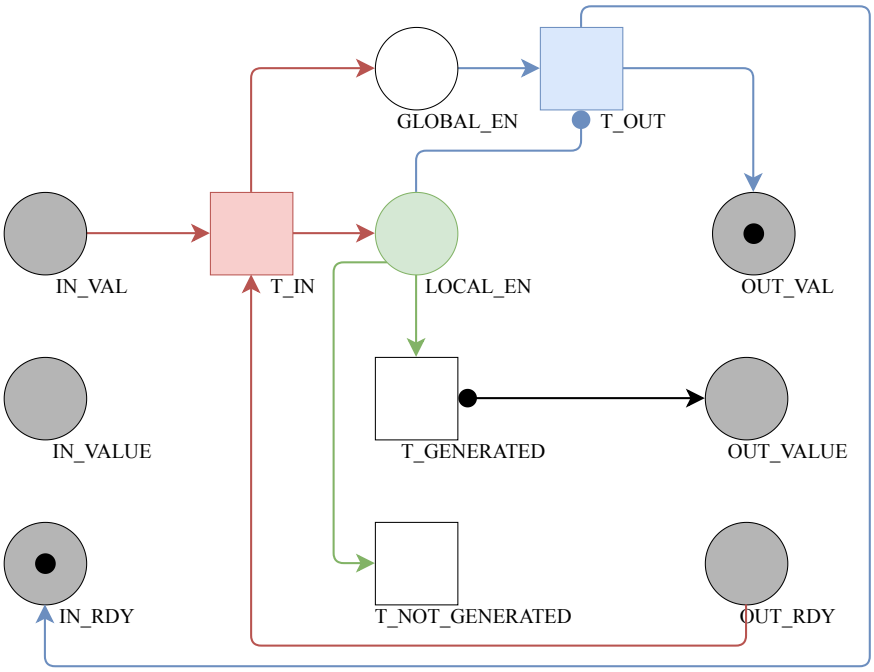


Figure 17: Item generator module

transitions between the two states. The same idea is used in the item consumer and transfer module as well.

5.3.2 Item consumer

An item consumer module (Figure 18) has the same structure as an item generator. It has one input and one output port, a T_IN transition to switch the module from the idle to the working state, a T_OUT transition to switch back and two central transitions which perform the actual task of the module. T_CONSUMED and T_NOT_CONSUMED are those two central transitions. T_CONSUMED fires if it is enabled by the LOCAL_EN place and if a token exists at IN_VALUE. T_NOT_CONSUMED can fire as soon as it is enabled by LOCAL_EN. Like the transitions in the item generator they fire non-deterministically. Therefore the module decision whether a token from the input is consumed is non-deterministic as well.

5.3.3 Item transfer

The item transfer module (Figure 19) follows the same logic. But instead of having two central transitions it has three, since there are more cases to cover. If the input contains a token and the output does not, it should be moved to the output. This is covered by T_TRANSFERED. But if the input is empty or the output is full the module should not do anything, which is covered by T_IN_EMPTY and T_OUT_FULL respectively. Again, only one of those three transitions can fire per cycle, since they all depend on LOCAL_EN. But unlike the item generator and consumer module the transfer module as a whole behaves deterministically. This is true even if it is not known whether T_IN_EMPTY or T_OUT_FULL will fire if the input is empty and the output contains a token. But since both transitions have the same effect (a token from LOCAL_EN is consumed) it is not relevant which one actually fires.

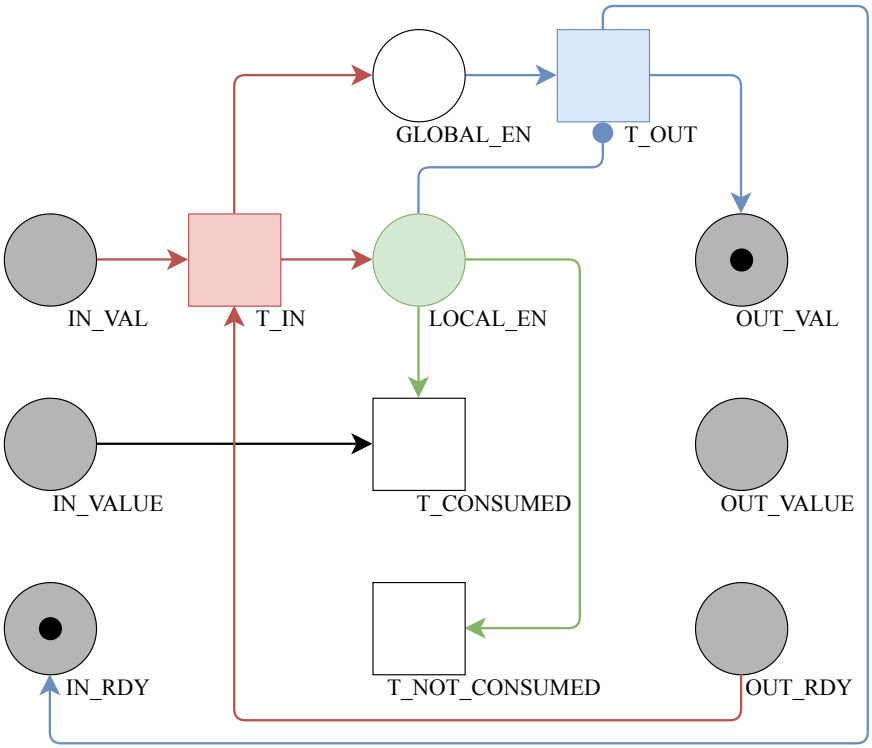


Figure 18: Item consumer module

5.3.4 Item loopback

This module is the only one whose Petri Net needs to be generated dynamically. The task of the item loopback module is to connect the output of all item consumer modules with the input of all item generator modules. Due to this, the amount of input and output ports depends on the amount of generator and consumer modules. To be more specific, the amount of input ports is equal to the amount of item consumer modules and the amount of output ports is equal to the amount of item generator modules. In comparison to the previous modules the item loopback is rather simple. Despite having an arbitrary amount of input and output ports it only consists of a single transition. This transition T_LOOP consumes the tokens from all IN_VAL places of all input ports and OUT_RDY tokens from all output ports and creates a token at each IN_RDY and OUT_VAL place. To simplify this, the loopback module enables all item generator modules if all item consumer modules have completed a working phase. With this functionality, the loopback module acts as a global barrier in the otherwise asynchronous system. An abstract Petri Net for this module is shown at Figure 20. In this graph n is equal to the amount of item consumers and m is equal to the amount of item generators.

5.3.5 Item splitter

Despite the problems of the first version of the splitter model from Section 4b the design is solid until the extension for priorities is added. Therefore the structure until that point (Figure 6) is reused. Now it only needs to be adapted to the current structure by adding valid and ready places for the inputs and outputs and by adding the transitions to switch between working and idle state. This results in Figure 21. It is important to note that places with the same represent the same element in P. This means, places with the same name may be merged together. This separation is mainly used to increase readability, since transitions like T_IN

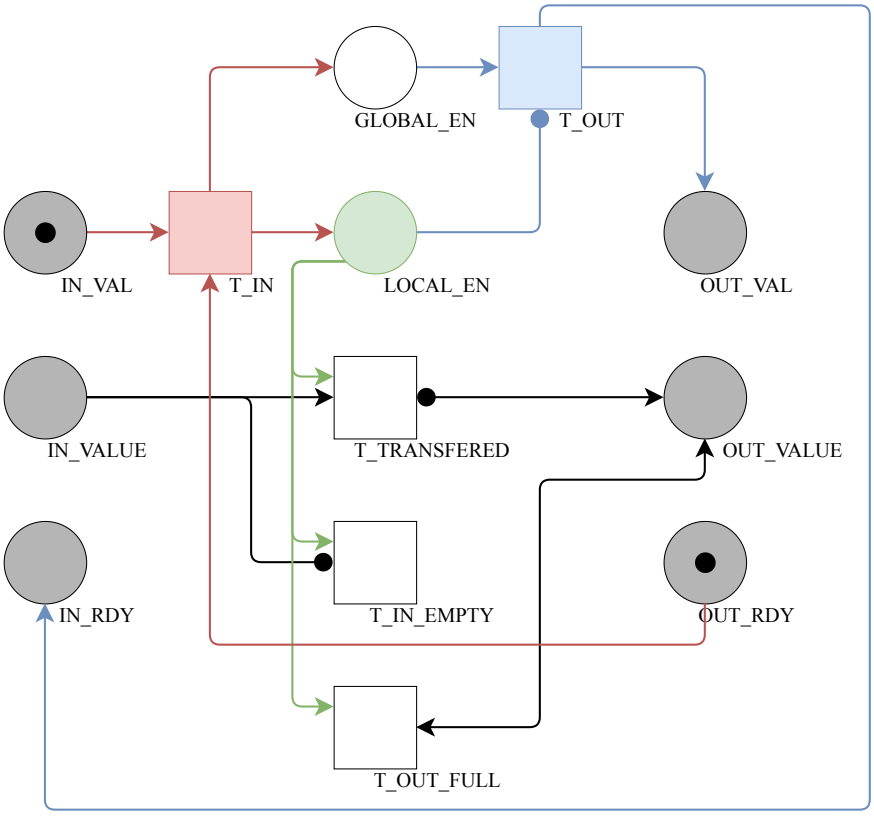


Figure 19: Item transfer module

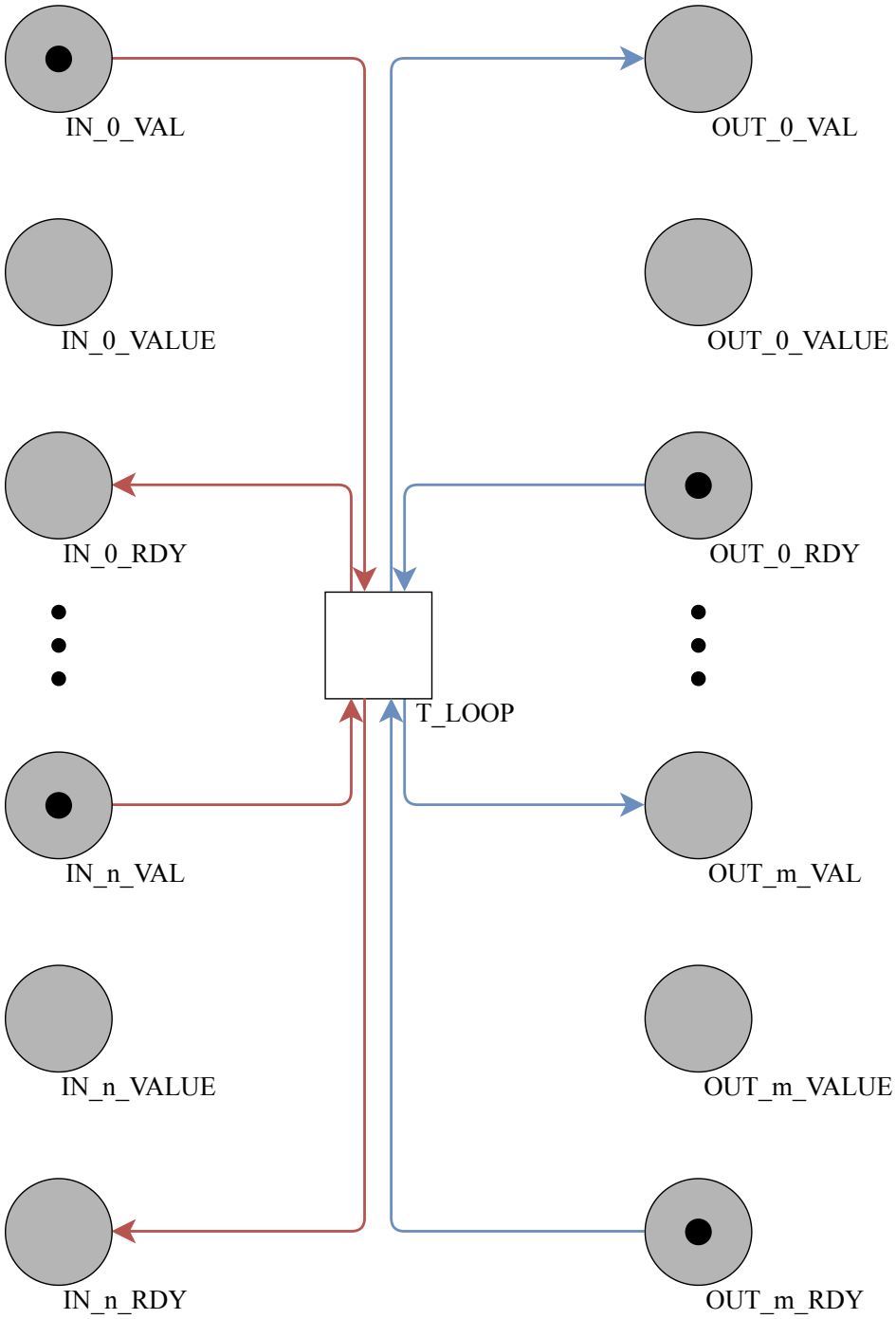


Figure 20: Item loopback module

connect multiple places distributed across the entire Petri Net which creates long, intersecting arcs. Now this Petri Nets misses an equivalent to the LOCAL_EN place. In the item generator, consumer and transfer module it enables only one of the central transitions to fire, but firing just one of the central transitions (T_00, T_01, T_10, T_11) is not enough for a splitter. For example if both inputs contain a token and both outputs are empty both tokens should be moved to an output before switching back to the idle state, and this requires two of the central transitions to fire. Therefore we add an additional place per input (IN_0_HAND, IN_1_HAND) and output (OUT_0_HAND, OUT_1_HAND) port. Those places contain a token if the corresponding port has not been handled jet. There are three circumstances under which an input port is seen as handled, therefore the token from IN_n_HAND has been consumed. First, if a token from IN_n_VALUE has been moved. In this case the corresponding central transition consumed the token from the 'handled' place. Second, if the input is empty. In this case no token can be moved from there. Therefore the transition IN_0_EMPTY consumes the token from IN_0_HAND if the top input is empty and IN_1_EMPTY does the same for the bottom one. The third case involves the outputs. If both outputs are handled no token from the input can be moved to any output any more. Because of this both inputs can be seen as handled as well, which enables OUT_HANDLED_0 and OUT_HANDLED_1 which in turn consume possible tokens from the corresponding IN_n_HAND places. It is not possible to merge both transitions. If only one output is empty, but both inputs contain a token only one of those input tokens can be moved. This causes the corresponding handled token, those are tokens in IN_0_HAND and IN_1_HAND, to be consumed as well. Now only one of the input places has its handled token, but both outputs are full. If OUT_HANDLED_0 and OUT_HANDLED_1 would be a single transition, it could not fire, since this transition requires a token at IN_0_HAND and IN_1_HAND. Therefore two independent transitions are required.

The 'handled' handling for the output ports works the same. An output port is handled if a token has been moved to the port, if the port is full or if both input ports have been handled. Now those handled places, therefore OUT_0_HAND and OUT_1_HAND, are used to control when T_OUT can fire. If T_IN generates the token in GLOBAL_EN, to switch to the working state, it also creates a token at all those handled places. If all of those handled places are empty and if GLOBAL_EN contains a token, T_OUT may fire to switch the splitter back to its idle state. With those modifications, the model is up to date again and we can start to add priorities. Figure 22 provides a visual representation of the current state of the Petri Net.

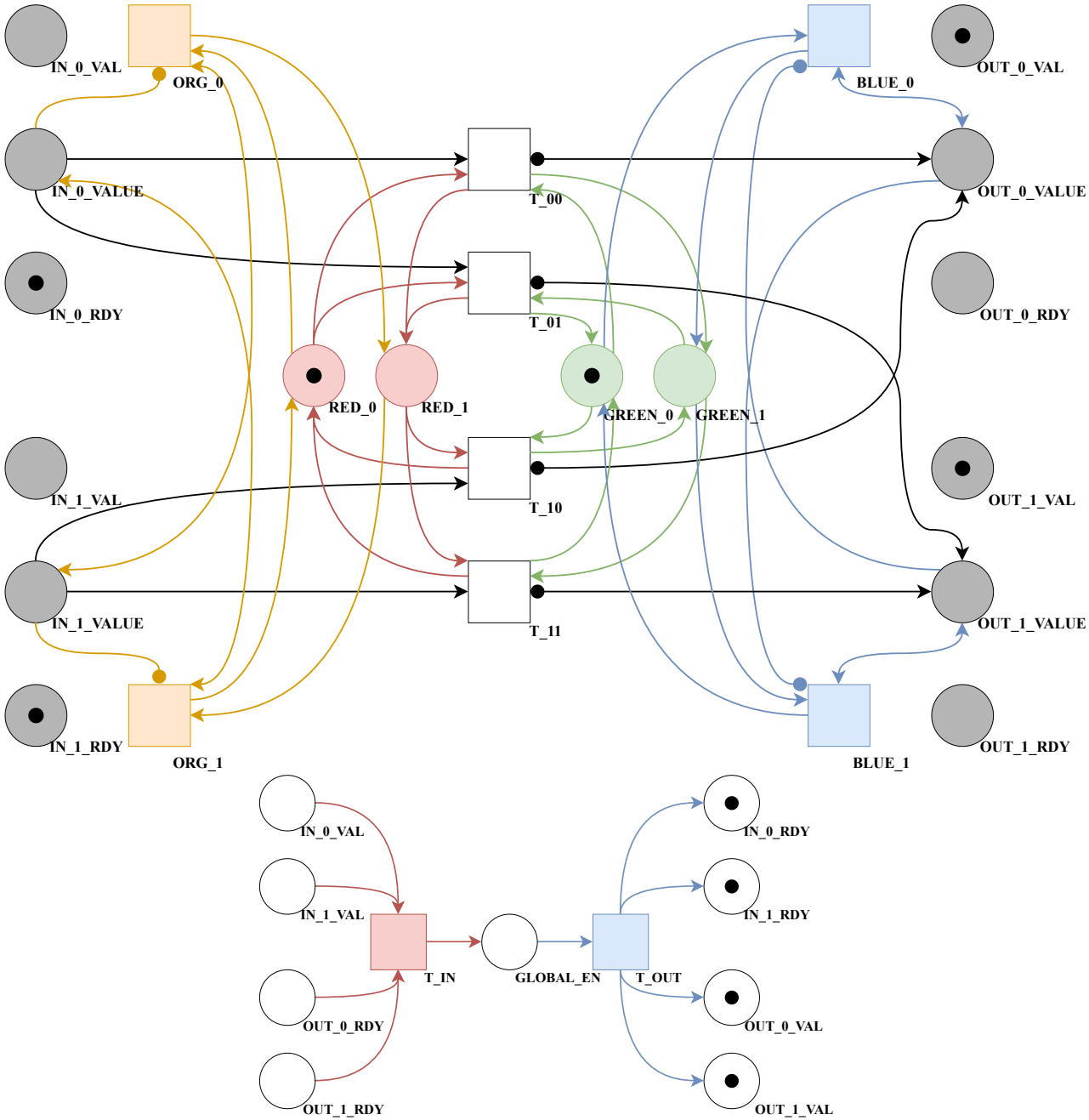


Figure 21: First adjustments of the previous model

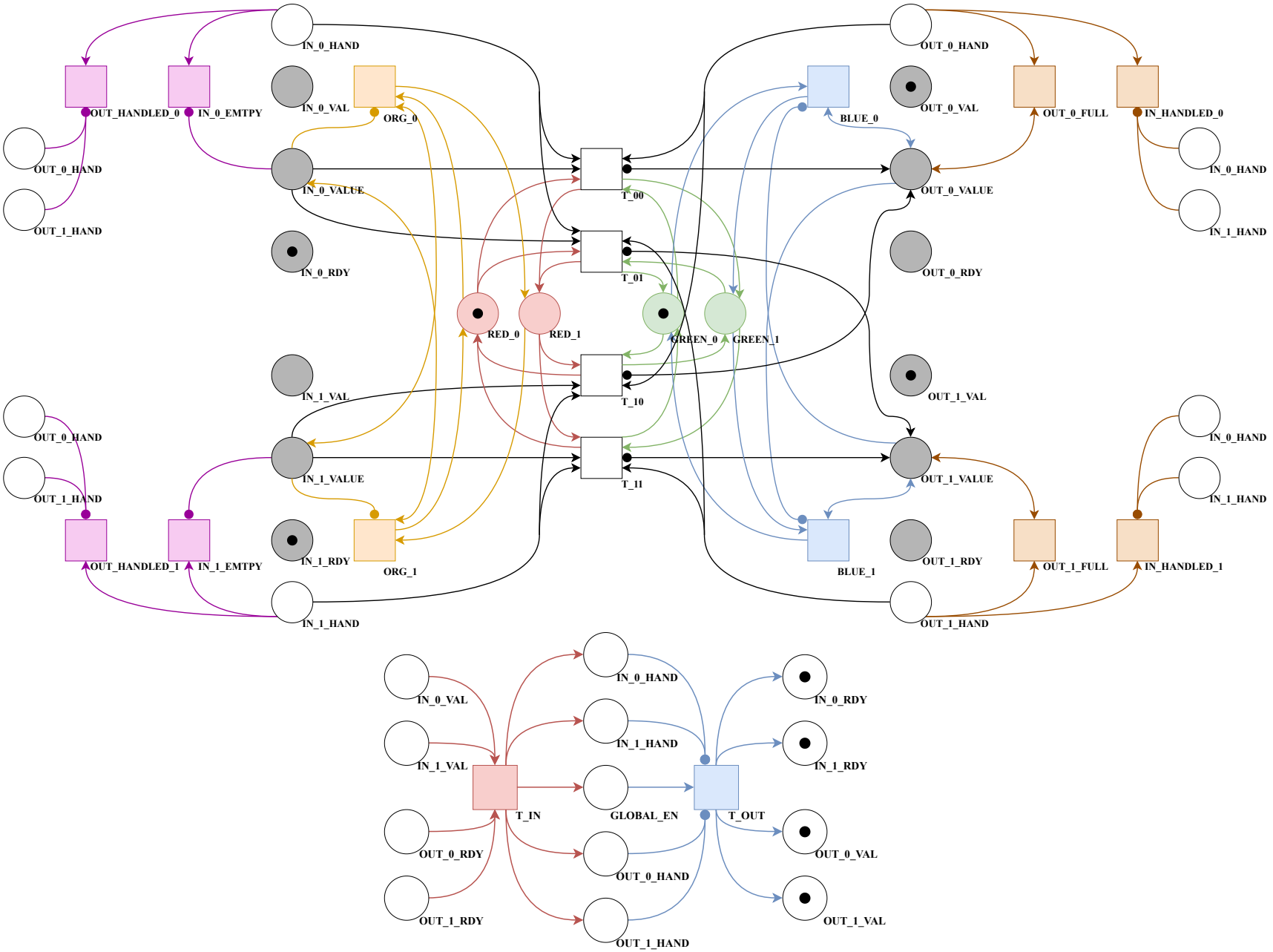


Figure 22: Final adjustments of the previous model

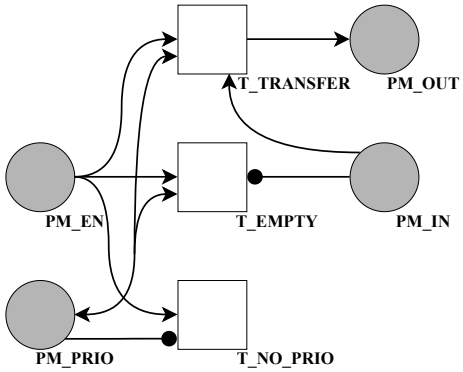


Figure 23: Preparation submodule for item splitter

One way to model priorities is to ensure, that the tokens in the red and green places are at the right position. Currently, if RED_0 and both inputs contain a token, the token from IN_0 will be consumed first. With this in mind, the idea is to ensure that the token is at the right place to consume from the prioritised input first. Of course the same applies for the output and the green extension. To prevent any interleaving issues, a new state needs to be added. This state could be added before or after the working state. If we insert this state after the working one, it would move the tokens from the red and green extensions after handling the input tokens. But for this to work, the initial marking of the module has to be changed, based on existing input and output priorities. For example, if IN_1 has priority, the token from the red extension has to be at RED_1 instead of RED_0. This is not necessarily a problem, but it can be prevented. Instead of resetting the module after it finished its working phase it can be prepared before it starts working. This means a preparation phase is added before the module actually distributes tokens from its inputs. With this, the initial marking does not need to be changed depending on priorities, since the module will move the red and green tokens itself if needed. To sum this up, the splitter module starts in its idle state, switches to the preparing state if T_IN fires and after that it starts working.

To prepare the splitter module a preparation module is introduced in Figure 23. It consists of an enable place PM_EN, which enables the module much like the LOCAL_EN of the previous ones. PM_PRIO contains a token if the respective port is prioritised. PM_IN is the red or green place from which a token should be taken and moved to PM_OUT if the port has priority. The three central transitions handle those actions. If the token needs to be moved, T_TRANSFER will move it from PM_IN to PM_OUT. If no movement is needed, T_EMPTY just consumes the enable token. In the case that a priority is set but PM_EN contains a token, none of those two transitions will fire. Instead T_NO_PRIO will consume the token from PM_EN. Each input and output port of the splitter module will receive such a preparation module. A visualisation of all changes is provided by Figure 24. For IN_0 the instance of such a module is called PREP_IN_0. PM_PRIO of PREP_IN_0 is equivalent to P_IN_0, the same holds for PM_EN and IN_0_PREP_EN. Since this module handles the preparation, in case the top input is prioritised it needs to move the token from the red extension to RED_0. Therefore the corresponding PM_IN place merges with RED_1 and PM_OUT with RED_0. The same applies for PREP_IN_1, PREP_OUT_0 and PREP_OUT_1. They are the same module type, which has been connected to different places.

To prevent the orange (ORG_0, ORG_1) and blue (BLUE_0, BLUE_1) extension from messing up the positioning of the red and green token during the preparation phase new arcs are added. ORG_0 is only allowed to move the red token if IN_1_HAND contains a token. Therefore it can only act during the working phase. Now it is only allowed to enable T_10 and T_11 by moving the red token if the bottom input has not been handled yet, which makes sense.

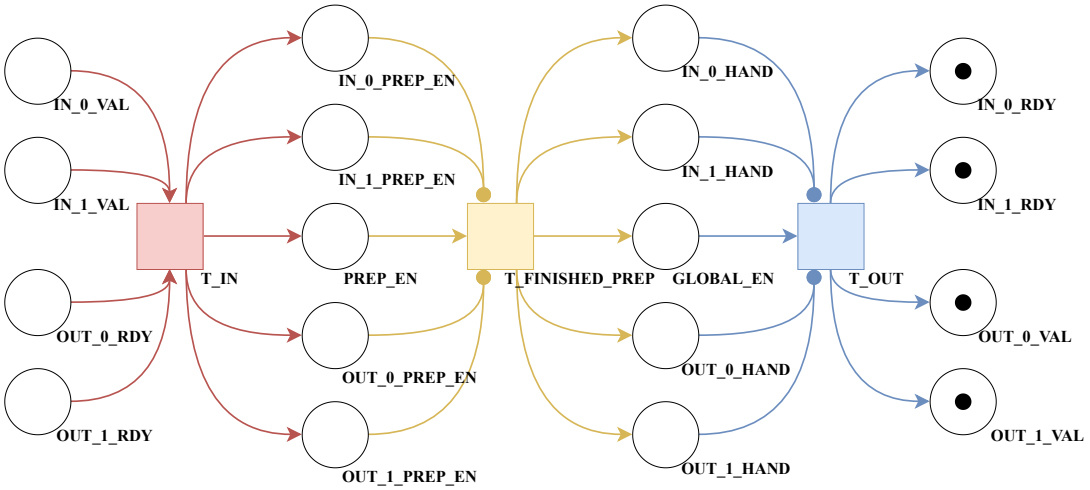


Figure 24: First part of the second model with priorities. Continued at Figure 25

To add the new splitter module state a new transition between T_IN and T_OUT is added. This transition T_FINISHED_PREP creates tokens at the old output of T_IN. T_IN in turn now creates a token in the new place PREP_EN, which acts as the GLOBAL_EN of the preparation phase. Additionally T_IN generates tokens at all four PREP_EN places. If all 4 places are empty T_FINISHED_PREP ends the preparation phase and starts the working phase by firing.

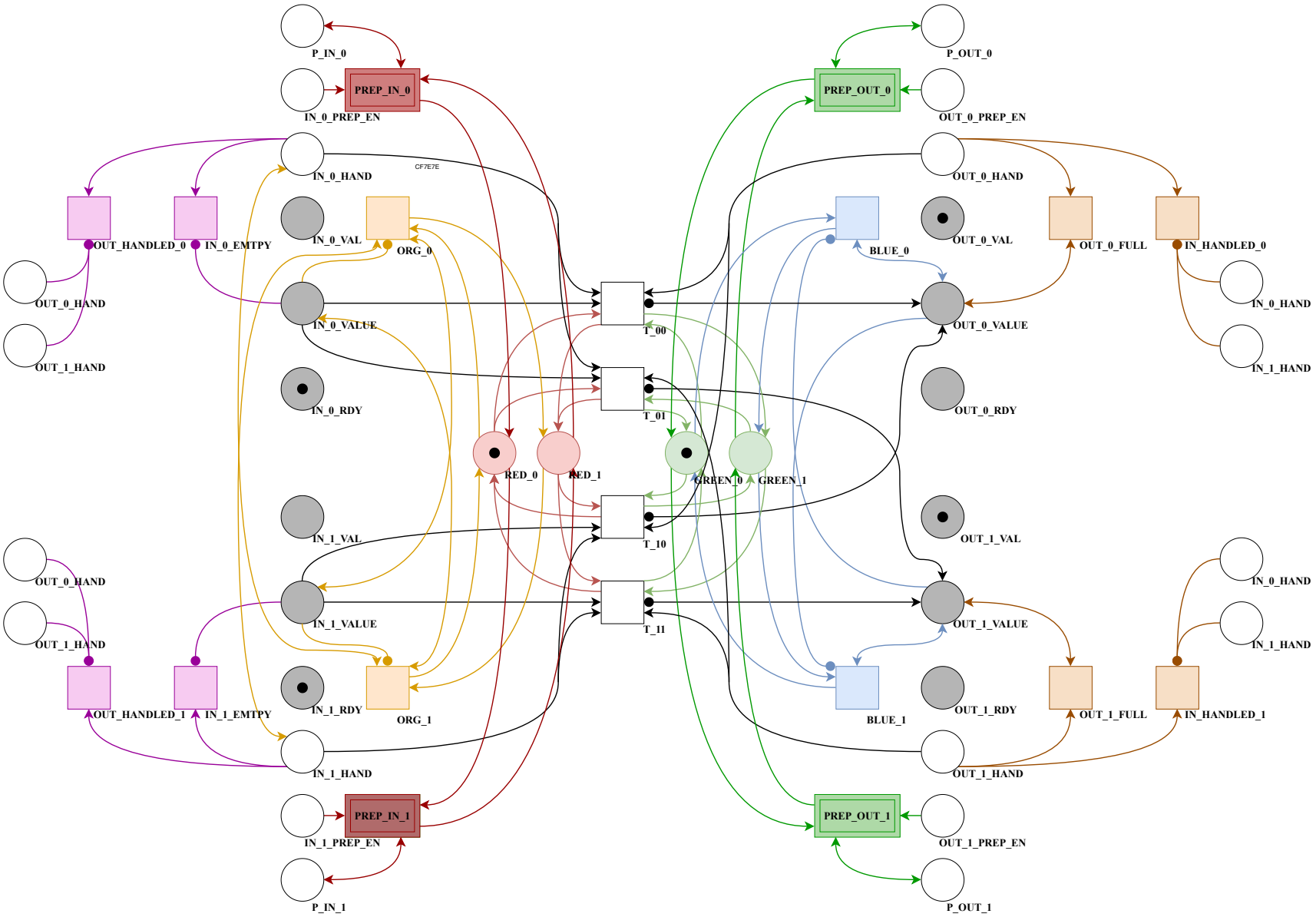
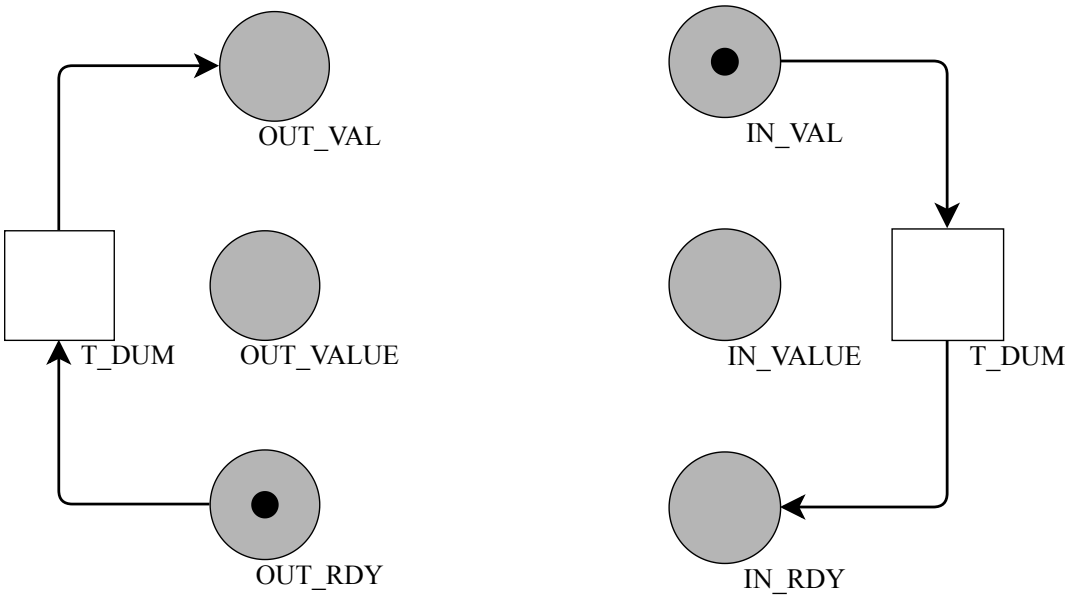


Figure 25: Continuation of Figure 24



(a) Dummy generator module

(b) Dummy consumer module

Figure 26: Dummy modules for unused splitter inputs and outputs

5.3.6 Dummy modules

Dummy modules are used to model unused splitter inputs and outputs. Since no tokens are created or consumed those modules only consist of a single transition which moves the IN_VAL token to IN_RDY in case of a dummy consumer and the OUT_RDY token to OUT_VAL in case of a dummy generator. Additionally they only have one port because they do not connect to any other modules. One might add the missing port to keep the interface uniform, but in this case those 3 added places would be unused by the dummy module itself and are never connected to any other modules. They are not needed.

5.4 Examples for module usage

This Section provides some examples for how all the modules could be arranged to model a belt balancer. Figure 29 shows all possible and allowed connections between two modules. This also shows how a transfer module needs to be added between two productive modules and how dummy modules can be used. Its counterpart in Factorio is shown in Figure 27. The translation of a Factorio Belt Balancer, like in Figure 27, to its corresponding model is rather simple. Our example has three inputs at the left and three outputs at the right side. Each of those inputs translates to one item generator module, in our example the input at the top left becomes item generator 0, the middle one generator 1 and the bottom one generator 2. The same applies for all three outputs, which become an item consumer module each. Since the top input is directly connected to the top output this connection is represented by a single item transfer module at the top. The splitter at the center translates to the module item splitter 0. Since the middle input is connected to the top input of that splitter a transfer module (item transfer 1) is added to connect those ports. The same applies to the loop, the bottom output of the splitter is connected with its bottom input via item transfer 2. And like the others, the connection between the top splitter output and the miiddle Belt Balancer output is modelled with an item transfer module. The connection between the bottom input and output follows the same schematic, except there is no loop. The bottom input and output of the last splitter are unused and therefore a dummy module

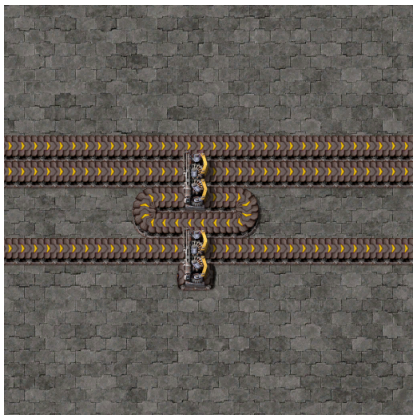


Figure 27: Original Belt Balancer for Figure 29

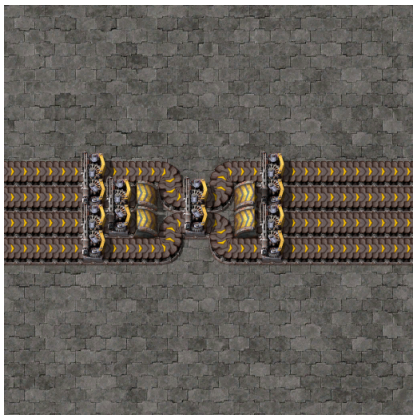


Figure 28: Original Belt Balancer for Figure 30

is needed. Since they count as transfer modules no item transfer module is added between them. Last but not least, the item loopback is added. This does not have any clear equivalent in the original Belt Balancer, it is only needed by the model to work properly.

To sum this translation process up, each Belt Balancer input becomes a item generator module, each output an item consume module and each splitter becomes an item splitter module. The belts between inputs, outputs and splitter are abstracted with the item transfer module and if any input or output port is not used a input or output dummy is added respectively. The item loopback module is added at the end to connect each item consumer module with each item generator module and has no in game equivalent.
Figure 28 together with Figure 30 provides a more complex translation example. But unlike the previous one this is a real, commonly used Belt Balancer and not just a construct for demonstration purpose.

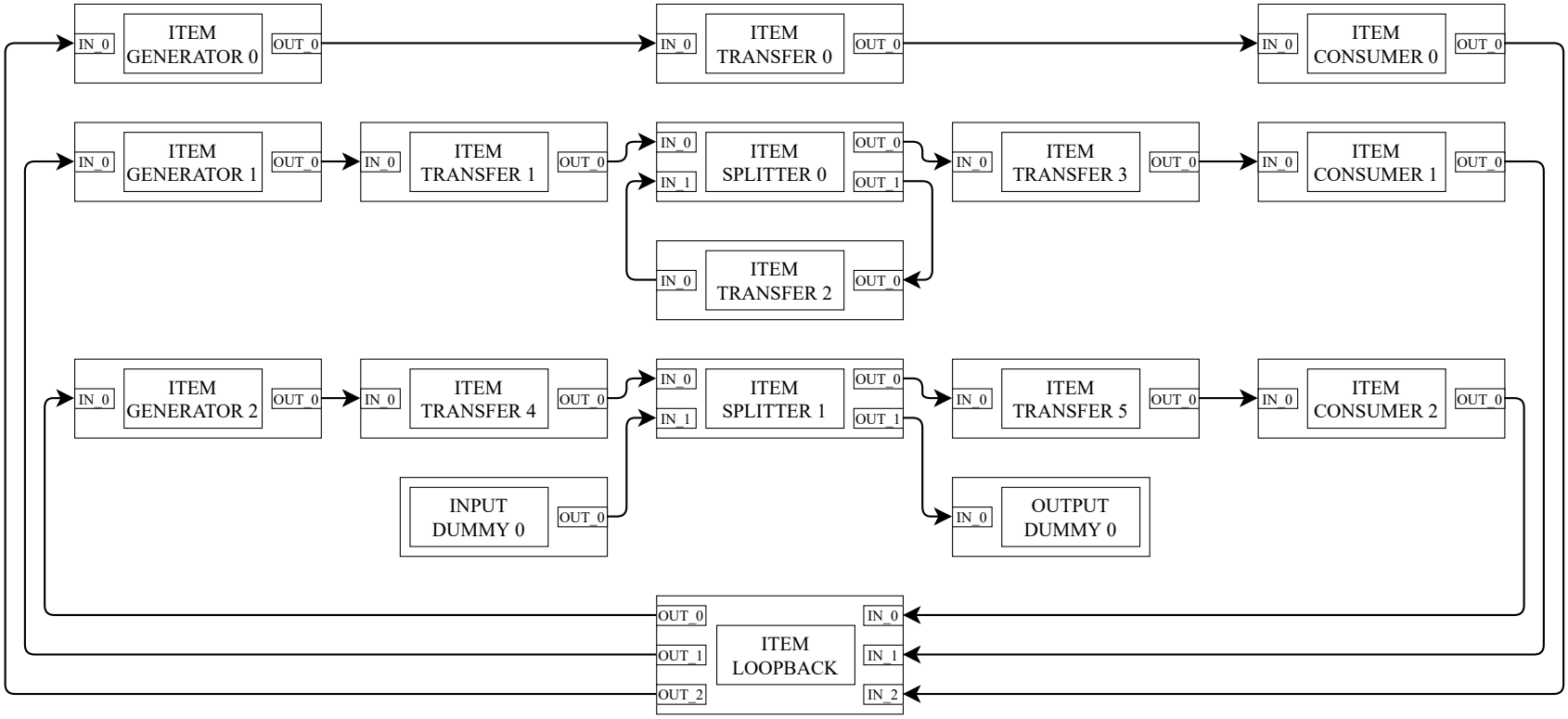


Figure 29: Example Belt Balancer containing all modules with all possible connections. Figure 27 shows the original in Factorio

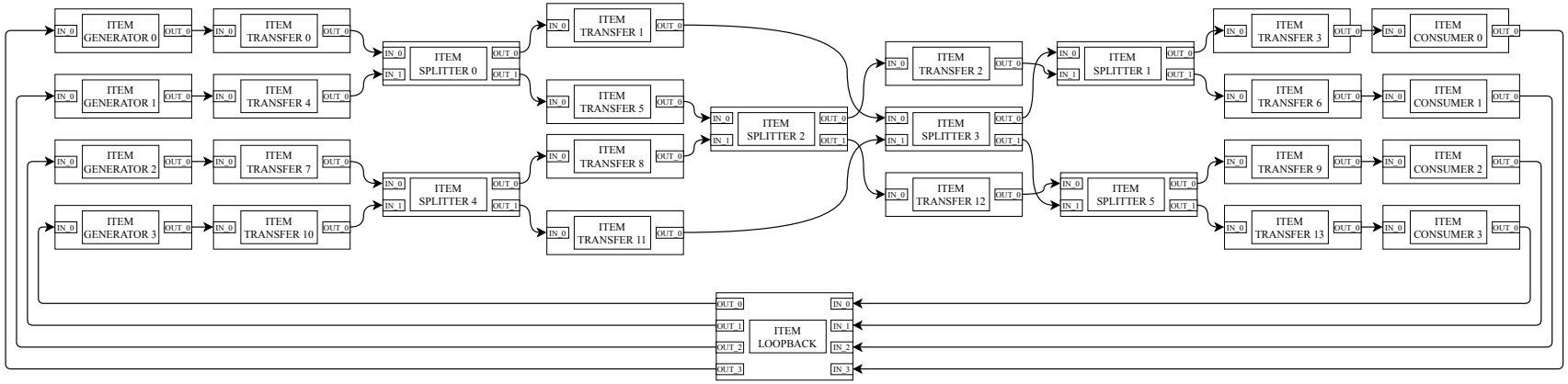


Figure 30: Classical 4 -> 4 Belt Balancer. Figure 28 shows the original in Factorio

6 Properties of Belt Balancer

With the model from Section 5 it is now possible to define certain properties for a Belt Balancer. The goal of this Section is to give definitions in linear temporal logic (LTL) for properties commonly used in the Factorio community. This includes, but is not limited to, whether a given Belt Balancer actually balances its output, how it effects throughput and how well it responds to irregular inputs and outputs.

6.1 Used sets and functions

For the next Sections we define the following sets and functions:

$$\begin{aligned} G &:= \text{ Set containing all generator modules.} \\ C &:= \text{ Set containing all consumer modules.} \end{aligned}$$

Those two sets are rather intuitive. G contains all generator modules used in the Petri Net for the Belt Balancer. With this it is possible to argue about the inputs of the given Belt Balancer. C does the same, but for all consumer modules and therefore all Belt Balancer outputs.

$$\begin{aligned} \text{hasFiredPositive} &: G \cup C \to \mathbb{B} \\ \text{hasFiredPositive}(m) &:= \begin{cases} \text{true, if } m \in G \wedge \text{ T\_GENERATED fired} \\ \text{true, if } m \in C \wedge \text{ T\_CONSUMED fired} \\ \text{false, otherwise} \end{cases} \\ \text{hasFiredNegative} &: G \cup C \to \mathbb{B} \\ \text{hasFiredNegative}(m) &:= \begin{cases} \text{true, if } m \in G \wedge \text{ T\_NOT\_GENERATED fired} \\ \text{true, if } m \in C \wedge \text{ T\_NOT\_CONSUMED fired} \\ \text{false, otherwise} \end{cases} \\ \text{hasFired} &: G \cup C \to \mathbb{B} \\ \text{hasFired}(m) &:= \text{hasFiredPositive}(m) \vee \text{hasFiredNegative}(m) \end{aligned}$$

Those functions are used to determine whether a given module caused the last state transition by firing the corresponding transition. For example assume two generator modules $g_1, g_2 \in G$ and during the last step g_1 generated a token. Therefore T_GENERATED, belonging to g_1, fired. In this case hasFiredPositive$(g_1) =$ true. Logically hasFiredPositive$(g_2) =$ false since only one transition fires per step, but hasFiredNegative$(g_2) =$ false as well. The negative variant is not a logical negation of hasFiredPositive, but instead it indicates the decision of a generator module to not create a token during this cycle, which means no item arrived at this Belt Balancer input. The same applies for consumer modules.

$$\text{canFirePositive} : G \cup C \to \mathbb{B}$$

$$\text{canFirePositive}(m) := \begin{cases} \text{true, if } m \in G \wedge \text{ T\_GENERATED can fire} \\ \text{true, if } m \in C \wedge \text{ T\_CONSUMED can fire} \\ \text{false, otherwise} \end{cases}$$

$$\text{canFireNegative} : G \cup C \to \mathbb{B}$$

$$\text{canFireNegative}(m) := \begin{cases} \text{true, if } m \in G \wedge \text{ T\_NOT\_GENERATED can fire} \\ \text{true, if } m \in C \wedge \text{ T\_NOT\_CONSUMED can fire} \\ \text{false, otherwise} \end{cases}$$

$$\text{canFire} : G \cup C \to \mathbb{B}$$

$$\text{canFire}(m) := \text{canFirePositive}(m) \vee \text{canFireNegative}(m)$$

The canFire functions are used to determine, whether a given module could fire during the next state change. Therefore whether all preceding places of T_GENERATED, T_CONSUMED or their counterparts, which are T_NOT_GENERATED and T_NOT_CONSUMED, contain a token.

$$\text{history} : G \cup C \to \mathbb{N}$$

$$\text{history}(m) := \begin{cases} \text{how often T\_GENERATED fired, if } m \in G \\ \text{how often did T\_CONSUMED fired, if } m \in C \end{cases}$$

This history function returns the amount of generated tokens at a given item generator or the amount of tokens consumed by a given item consumer, depending on the type of the module. For all following properties we assume one-safety for the model as a whole. In Section 7.3 we prove this for the item generator module. For all other modules one-safety is assumed. Therefore each place may only contain 0 or 1 token.

6.2 Balancer is never stuck

A Belt Balancer, that stops working after a couple of cycles is not really useful. Therefore the first property describes a Balancer, which is never stuck. A Belt Balancer in Factorio is stuck, if any input is no longer able to receive an item. In terms of the Petri Net model this means, there exists an item Generator which cannot generate a new token. This translates to the following.

$$\exists g \in G : \Diamond\Box\neg\text{canFirePositive}(g)$$

To get the definition of a Belt Balancer, that is never stuck, this formula needs to be negated.

$$\begin{aligned} &\neg(\exists g \in G : \Diamond\Box\neg\text{canFirePositive}(g)) \\ \Leftrightarrow &\forall g \in G : \neg(\Diamond\Box\neg\text{canFirePositive}(g)) \\ \Leftrightarrow &\forall g \in G : \Box\Diamond\text{canFirePositive}(g) \end{aligned}$$

Therefore in order for a Belt Balancer to be never stuck every input has to be able to generate a token always eventually. However this insight did not occur immediately. In fact it originally began with a definition for a Balancer, that is never stuck.

$$\forall g \in G : \Box(\text{hasFiredPositive}(g) \Rightarrow (\Diamond\text{canFirePositive}(g) \wedge \exists c \in C : \Diamond\text{canFirePositive}(c)))$$

In order for a given Balancer to be stuck some token had to be generated in the past at an item generator, but for some reason this token cannot be moved from the generator and therefore blocks the generation of additional ones. The idea is to define a property which guarantees, that every generated token will be moved eventually from its item generator to an item consumer module later on. The removal of the token from the generator module is guaranteed by the implication $\text{hasFiredPositive}(g) \Rightarrow \Diamond\text{canFirePositive}(g)$, because the tokens has to be moved elsewhere eventually in order for $\Diamond\text{canFirePositive}(g)$ to be true. The implication $\text{hasFiredPositive}(g) \Rightarrow \exists c \in C : \Diamond\text{canFirePositive}(c)$ is used to guarantee the arrival of a token at an item consumer eventually, but not its consumption. This can be resolved by later conditions, but since it is possible to eliminate this part of the implication it does not matter.

The expression can be shortened by removing the part about item consumers. This is due to the fact that Belt Balancers, the originals from Factorio and the model, have a limited capacity. In Factorio each belt can only hold on to a limited amount of items at any given time, this is also true for splitter and underground belts. And since each Belt Balancer consists of a limited amount of belts, underground belts and splitter the maximum amount of items, which can be in a given Belt Balancer at any given time, is also limited. The same is true for the model. We assume one-safety for all modules, therefore each place holds no more than one token at any given time. And, much like belts, underground belts and splitter in Factorio, each module in our model can only contain a finite amount of tokens at once, since the amount of places is limited. Additionally the model of a given Belt Balancer consists of a limited amount of modules, therefore the model as a whole can only contain a limited amount of tokens at once. Now, in order to be able to move a newly generated token from an item generator ($\text{hasFiredPositive}(g) \Rightarrow \Diamond\text{canFirePositive}(g)$ holds) there needs to be space in the rest of the model. To provide this space for all generated tokens indefinitely some of them need to be consumed by an item consumer eventually. This means, if $\Box(\text{hasFiredPositive}(g) \Rightarrow \Diamond\text{canFirePositive}(g))$ holds, $\Box(\text{hasFiredPositive}(g) \Rightarrow \exists c \in C : \Diamond\text{canFirePositive}(c))$ has to hold as well for any given $g \in G$. Therefore the second part can be removed from the definition altogether. This results in the following.

$$\forall g \in G : \Box(\text{hasFiredPositive}(g) \Rightarrow \Diamond\text{canFirePositive}(g))$$

In its current state, this property would be false. If an item generator would never create a token the implication would always be true, even if a generated token would never be moved. Therefore we reduce it further and get the following.

$$\forall g \in G : \Box\Diamond\text{canFirePositive}(g)$$

With the proper definition of a Belt Balancer, that is never stuck, it is now possible to add various premises to guarantee this stuck free behaviour in certain environments. For example, one variant could define, that every item generator, that can generate a token, will do so eventually. Another example would be a property which guarantees, that only item consumers actually consume items. We will give two examples for this. The first one guarantees stuck free behaviour under optimal conditions. Therefore every item generator, that can generate a token will do so. This means if T_GENERATED and T_NOT_GENERATED can fire T_GENERATED will always be chosen. The same applies to all item consumer modules, T_CONSUMED is prioritized over T_NOT_CONSUMED.

$$\begin{aligned}&(\forall m \in G \cup C : \Box(\text{canFirePositive}(m) \Rightarrow \neg\text{hasFired}(m)\,\mathcal{U}\,\text{hasFiredPositive}(m)))\\ \Rightarrow&\forall g \in G : \Box\Diamond\text{canFirePositive}(g)\end{aligned}$$

This is achieved by preventing $\text{hasFired}(m)$ to become true until $\text{hasFiredPositive}(m)$ becomes true. The second variant is the opposite, it guarantees at least one item generator and item consumer at any given time,

which create or consume a token eventually, if they can.

$$\begin{aligned}&(\Box\exists g \in G : \Diamond(\text{canFirePositive}(g) \Rightarrow \text{hasFiredPositive}(g)))\\ \wedge&(\Box\exists c \in C : \Diamond(\text{canFirePositive}(c) \Rightarrow \text{hasFiredPositive}(c)))\\ \Rightarrow&\forall g \in G : \Box\Diamond\text{canFirePositive}(g)\end{aligned}$$

This can result in the extreme case of having only one item generator with a token and only one item consumer, where it could go. Therefore if a Belt Balancer is never stuck with this condition, every of its inputs has to be connected to every output. Otherwise there would exist an input-output pair with which it would be stuck, if they are the only one to create and consume tokens. Assume a Belt Balancer with two inputs and two outputs. The top input is connected to both outputs, but the bottom one is only connected with the bottom output. This Balancer would not hold. If only the bottom input generates tokens and only the top output consumes them, then the Belt Balancer would be stuck since there is no way for the tokens from the bottom input to reach an output, which would consume them. Therefore the Belt Balancer fills up with tokens and stops to accept tokens from the bottom input eventually.

6.3 Balanced output

The next step is to define whether the output of a given Belt Balancer is actually balanced. There have been 3 steps for defining an output balanced Belt Balancer. The first one restricts under which circumstances $\text{canFirePositive}(c)$ for an item consumer $c \in C$ is positive.

$$\Diamond\Box(\forall c \in C : \text{hasFiredPositive}(c) \Rightarrow (\forall m \in C \setminus \{c\} : \neg\text{canFirePositive}(c)\,\mathcal{U}\,\text{hasFiredPositive}(m)))$$

If this holds, after consuming a token each item consumer may not be able to do so again, until every other item consumer consumed a token too, therefore $\text{hasFiredPositive}(m) = \text{true}$. But instead of enforcing this immediately we allow a initialization phase by enforcing this property eventually. Most Belt Balancers do not work immediately since the delay between different inputs and outputs is different or internal belt loops need to receive items first. If for example the path from the bottom input to the bottom output contains one splitter and the path from the top input to the top output 3, the items will reach the bottom output faster than the top one and the output is temporarily unbalanced.

This formula implies a fixed output pattern, which means there is a fixed order at which each consumer consumes a token. Assume a Belt Balancer with 3 consumer modules a, b, c. Item consumer a removed the first token, and therefore $\text{canFirePositive}(a)$ has to be false until $\text{hasFiredPositive}(b)$ and $\text{hasFiredPositive}(c)$ have been true. After that, module b consumes the second token and has to wait for a and c. Since a is still waiting for c, module c has to be the next one. After consuming its token, module c has to wait for a and b. Now module a is the only one which can consume the next token, and after that b is the only one to do the same and the cycle repeats. While such a fixed output pattern may be desirable in certain situations, it does not cover all valid Belt Balancers. It would be perfectly fine for the pattern to change after a, b and c consumed their token. For example $a, b, c, c, a, b, a, c, b$ would be fine, every output only consumes a token after everyone had its turn. One way to allow those pattern changes is to track the amount of tokens consumed by each item consumer, which is done by $\text{history}(c)$ as mentioned in Section 6.1.

$$\Diamond\Box(\forall c_1, c_2 \in C : c_1 \neq c_2 : |\text{history}(c_1) - \text{history}(c_2)| \leq 1)$$

This modified property holds if the amount of items consumed between two item consumer modules is less than or equal to 1. This allows for variations of the output pattern while guaranteeing a balanced output. With

this the output has to be balanced, because if the property above holds, $\Diamond\Box(\exists n \in \mathbb{N} : \forall c \in C : \text{history}(c) = n \vee \text{history}(c) = n + 1)$. Therefore there are only two possible results for history(c), either its n or $n + 1$. If this would not be the case, there has to exist a $c' \in C$, such that either history$(c') < n$ or $n + 1 <$ history(c'). In case of history$(c') < n$ the property for the Belt Balancer could not hold, since for a $c \in C$: history$(c) = n+1$ the condition $|\text{history}(c) - \text{history}(c')| \leq 1$ would not be true.
The same applies for the case $n + 1 <$ history(c'). Now the property does not hold because there is some $c \in C$: history$(c) = n$ with which $|\text{history}(c) - \text{history}(c')| \leq 1$ is false. In order for the property to hold in future states as well, item consumers c with history$(c) = n + 1$ may not be able to consume a token until all other outputs c' with history$(c') = n$ have consumed one. Otherwise there would exist a consumer c with history$(c) = n + 2$ and a consumer c' with history$(c') = n$ which would violate $|\text{history}(c) - \text{history}(c')| \leq 1$. And this all together results in a Belt Balancer with a Balanced output.

This definition has a problem. Previously the diamond operator was enough to cover the initial period, but since we are now comparing the total amount of consumed tokens this no longer works. Assume a Belt Balancer with two outputs. If during initialization the top output consumed three tokens and the bottom one zero, and after that they output in an alternating fashion, the property above would not hold since the difference between them would always be greater one. But this should still be seen as a valid Belt Balancer since the output is balanced after the initial couple of tokens. Our approach for this holds, if there exists some $x_n \in \mathbb{N}$ for each $n \in C$, such that the following holds.

$$\forall c \in C : \Diamond\Box(\forall m \in C : |(\text{history}(c) - x_c) - (\text{history}(m) - x_m)| \leq 1)$$

The idea for this approach is to determine the amount of tokens consumed during the initialization period for each output and subtract those from their respective history. This normalizes the history for each output to the point, where the Belt Balancer actually starts to balance its output.

The third step is to generalize this definition even further. With the previous approach we eliminated the need for a fixed output pattern, but it still does not cover all valid Belt Balancers. Assume a Balancer with 3 outputs which outputs three items at the top, three items at the middle and three items at the bottom. If this pattern is repeated indefinitely this would be a valid Belt Balancer, but since the difference between two outputs is temporarily greater than one it would be rejected by all current definitions. One might be tempted to just increase the allowed maximum difference, but to truly cover all valid Balancers this limit needs to be infinite. This in turn would render this definition useless, since it would hold for any given Balancer construct. Assume a Balancer with two outputs and all incoming items are routed to the top one. Since the limit is infinite this construct would hold, despite not balancing its output at all. Defining a maximum limit for each Belt Balancer individually, like in the following definition, would not work either. For a consumer $n \in C$, $x_n \in \mathbb{N}$ is defined as in the previous approach.

$$\exists l \in \mathbb{N} : \forall c \in C : \exists x_c \in \mathbb{N} : \Diamond\Box(\forall m \in C : |(\text{history}(c) - x_c) - (\text{history}(m) - x_m)| \leq l)$$

Again, assume a Belt Balancer with two outputs. First, the top output receives an item, then the bottom one. After that, the top one gets two, the bottom one gets two. Then three at the top, three at the bottom, and so on. This would be a valid Balancer since the output is always balanced eventually. But there would not exist a $l \in \mathbb{N}$ such that the property above holds. While it is not possible to create such a Balancer in Factorio without being able to either construct infinitely large Balancers or combining a Balancer with Factorios Turing complete circuit system it still gave inspiration for a generalised definition for a balanced output. Each $x_n \in \mathbb{N}$ for a consumer $n \in C$ is defined as in the previous approaches.

$$\forall c \in C : \exists x_c \in \mathbb{N} : \Box\Diamond(\forall m \in C : (\text{history}(c) - x_c) = (\text{history}(m) - x_m))$$

With this the previously used limit is removed entirely and instead an always eventually occurring, balanced state is defined. But there is still a problem if someone wants to actually prove this property for a given Belt Balancer with, for example, a model checker. While a quantor over a finite set like C can be resolved to a series of logical AND and OR operations, this is not always possible in the case of an infinite set like $\mathbb{N}$. At least this would result in an infinitely long expression which cannot be proven by a model checker. Therefore it is desirable to remove the $\exists x_c \in \mathbb{N}$ part from the definition. One alternative is the following.

$$\Box\Diamond\left(\Box(\forall g \in G : \neg\text{hasFiredPositive}(g)) \Rightarrow \Diamond\Box\left(\forall c \in C : \text{history}(c) \geq \left\lfloor \sum_{i=1}^{|C|} \cdot \frac{1}{C} \right\rfloor\right)\right)$$

This approach does not restrict the output pattern at all. Instead it checks, whether at any given point if no new items are generated at any input, the amount of consumed items at all outputs is distributed equally. This also eliminates the problem with the initial phase, during witch the balancer does not distribute items evenly since this also happens if no new items are generated at the input and the rest inside the balancer is slowly transferred to an output. This approach also handles rather abstract balancers like the previous one, which increases the amount of items consumed at each output after each cycle.

6.4 $a \to b$ Belt Balancer

With a proper definition of a balanced output it is now possible to define the first Belt Balancer property, which is widely used in the Factorio community. A Balancer with a inputs and b outputs is considered a valid $a \to b$ Belt Balancer if it distributes items from its inputs to its outputs evenly under optimal conditions (as defined in 6.2).

$$(\forall m \in G \cup C : \Box(\text{canFirePositive}(m) \Rightarrow \neg\text{hasFired}(m)\,\mathcal{U}\,\text{hasFiredPositive}(m))) \Rightarrow$$
$$\Box\Diamond\left(\Box(\forall g \in G : \neg\text{hasFiredPositive}(g)) \Rightarrow \Diamond\Box\left(\forall c \in C : \text{history}(c) \geq \left\lfloor \sum_{i=1}^{|C|} \cdot \frac{1}{C} \right\rfloor\right)\right)$$

This is the weakest definition of all properties and is therefore the minimum required for a Belt Balancer in order for it to be actually usable.

6.5 Throughput limited $a \to b$ Belt Balancer

The definition of a throughput limited $a \to b$ Belt Balancer is a weaker version of its throughput unlimited variant, which will be described in Section 6.6. In order to be a valid throughput limited $a \to b$ Balancer, the Balancer has to full fill three properties.

First, it has to be a valid $a \to b$ Balancer, as defined in Section 6.4.

Second, every input has to be connected with every output, which can be defined like this.

$$\forall A \in \mathcal{P}(G) \setminus \emptyset : \forall m \in A \cup C : \Box(\text{canFirePositive}(m) \Rightarrow \neg\text{hasFired}(m)\,\mathcal{U}\,\text{hasFiredPositive}(m))$$
$$\Rightarrow \forall c \in C : \Box\Diamond\text{canFirePositive}(c)$$

Therefore in order for every input to be connected with every output, every output has to receive an item eventually as long as at least one input receives items.

The last property is maximum throughput. This means depending on a and b the Belt Balancer transfers the maximum amount of items given optimal conditions (as defined in 6.2).

$$\begin{aligned}&(\forall m \in G \cup C : \Box(\text{canFirePositive}(m) \Rightarrow \neg\text{hasFired}(m)\,\mathcal{U}\,\text{hasFiredPositive}(m)))\\ \Rightarrow&(|G| \leq |C| \Rightarrow \forall g \in G : \Diamond\Box(\neg\text{hasFiredNegative}(g)))\\ \wedge&(|G| \geq |C| \Rightarrow \forall c \in C : \Diamond\Box(\neg\text{hasFiredNegative}(c)))\end{aligned}$$

The maximum amount of tokens transferable per processing cycle is $\max(|G|, |C|)$. Each item generator and consumer module can only process one token per processing cycle. Therefore the Balancer can only move $|G|$ tokens per processing cycle, if $|G| \leq |C|$ because only $|G|$ tokens can be generated each cycle. The same applies for the case $|G| \geq |C|$. Since each consumer can only consume one token per processing cycle it is only possible for up to $|C|$ tokens to be distributed by the Balancer in each cycle. In order to guarantee maximum throughput each item generator module has to generate a token per cycle in case of $|G| \leq |C|$ and each item consumer has to consume a token per cycle in case of $|G| \geq |C|$. Together with the premise, which guarantees each generator and consumer fires positive if it can, this can be expressed by $\Diamond\Box(\neg\text{hasFiredNegative}(m))$, since T_NOT_GENERATED or T_NOT_CONSUMED can only fire if T_GENERATED and T_CONSUMED can not.

If all three properties hold, which means the given Balancer is a valid $a \rightarrow b$ Belt Balancer, each input is connected with each output and it achieves maximum throughput given optimal conditions, the Balancer is considered a throughput limited $a \rightarrow b$ Belt Balancer.

6.6 Throughput unlimited $a \rightarrow b$ Belt Balancer

A throughput unlimited $a \rightarrow b$ Belt Balancer is a special case of the throughput limited variant, which also clears up the reason for their names. A throughput limited Belt Balancer guarantees maximum throughput under optimal conditions, therefore it transfers the maximum amount of items possible if all inputs and outputs are used. But this does not guarantee maximum throughput if for example only 2 of 3 inputs and 2 of 4 outputs are used, which means all other inputs and outputs never generate or consume items. In those cases the throughput of the Belt Balancer may be limited which results in the name throughput limited $a \rightarrow b$ Belt Balancer. The unlimited version guarantees maximum throughput even if only a subset of inputs and outputs are used. For this we define an additional property.

$$\begin{aligned}&\forall g \in \mathcal{P}(G) \setminus \emptyset : \forall c \in \mathcal{P}(C) \setminus \emptyset : ((\forall m \in g \cup c : \Box(\text{canFirePositive}(m) \Rightarrow\\ &\neg\text{hasFired}(m)\,\mathcal{U}\,\text{hasFiredPositive}(m)))\\ \Rightarrow&(|g| \leq |c| \Rightarrow \forall m_g \in G : \Diamond\Box(\neg\text{hasFiredNegative}(m_g)))\\ \wedge&(|g| \geq |c| \Rightarrow \forall m_c \in C : \Diamond\Box(\neg\text{hasFiredNegative}(m_c))))\end{aligned}$$

This property is pretty much the same as the maximum throughput property from Section 6.5, except it guarantees maximum throughput for all combination of subsets of inputs and outputs. Therefore a Belt Balancer with this property can never be the bottleneck of a system. However it is not guaranteed, that the output is balanced if only a subset of inputs or outputs is actually used, in those cases the throughput is just unlimited.

6.7 Universal $a \rightarrow b$ Belt Balancer

The next logical step is to define a Balancer, which is capable to balance its output even if not all inputs and outputs are used. Those are known as universal Belt Balancers since they can be used universally. In order to

convert all previously defined properties to their universal counterpart we substitute each occurrence of G and C with X and Y respectively. Additionally the following is added as a condition to all properties.

$$\forall X \in \mathcal{P}(G) \setminus \emptyset : \forall Y \in \mathcal{P}(C) : (\text{substitued variant of previousely defined property})$$

While a Balancer with universal properties is rather convenient, it is usually difficult do design those. Additionally they require a lot of space and a large amount of splitters, which translates to high material cost in the game. Therefore there exists a weaker variant of the universal property. A universal $(x - a) \to (y - b)$ Belt Balancer has a inputs, b outputs from which at least x inputs and y outputs have to be used in order for the properties to hold.

$$\forall X \in \{A : A \in \mathcal{P}(G) \setminus \emptyset \wedge x \leq |A| \leq a\} : \forall Y \in B : B \in \mathcal{P}(C) \setminus \emptyset \wedge y \leq |B| \leq b :$$
$$(\text{substitued variant of previousely defined property})$$

Those properties are not universal, but they can handle up to $(a - x)$ unused inputs and $(b - y)$ unused outputs which makes them more flexible than their normal counterparts.February

7 Verification

This Section explains how Petri Net properties can be verified. Since the previously proposed properties are rather complicated we will focus on verifying one-safety, more specifically one-safety for the item generator module. We will use two methods. The first one uses a modified version of deadlocks, while the second one takes the more classical approach of proving a set of place invariants, which in turn guarantee one-safety.
To verify more complex properties tools like model checkers should be used, which will be discussed in Section 7.4.

7.1 Preparation

At the moment the Petri Net uses inhibitor arcs to simplify the model. But to perform the actual verification we need to unfold those arcs by adding complementary places. Figure 31 shows the unfolded form of the item generator module. The IN_VALUE place has been split into IN_VALUE and its inverted counterpart IN_VALUE_NEG, the same happened with LOCAL_EN and OUT_VALUE. Additional arcs have been added as well where needed. The item generator module on its own cannot do anything, it is stuck with its initial marking. Therefore we need to add transitions to model the behaviour of modules at its input and output. This is achieved by adding two transitions at the input and two transitions at the output (Figure 32). We assume each module at the input and output always follow the protocol specified in 5.2, which is why T_IN creates a token at IN_VALUE only if this place is empty (therefore IN_VALUE_NEG needs to contains a token) and if it can consume one from IN_RDY. Additionally it creates a token at IN_VAL to activate the generator module. T_IN_NOT models the case of the variant, which does not generate a token at the value place of the input port. T_OUT and T_OUT_NOT do the same for the output.

This simulation of input and output in its current form is to optimistic. It creates the token at IN_VAL immediately after consuming one from IN_RDY, but the protocol allows some downtime between consuming the token from IN_RDY, generating a token at IN_VALUE and creating the token at IN_VAL. To represent this properly additional transitions and places are added at the input and output (see Figure 32). In preparation for the verification all transitions have been annotated with a shorter name.

7.2 Modified deadlocks

One Method to prove one-safety uses a modified version of deadlocks [3]. The original definition of a deadlock $Q \subseteq P$ is $\forall t \in T : (\exists p \in Q : p \in t^\bullet) \Rightarrow (\exists q \in Q : q \in {}^\bullet t)$. This guarantees, if a token is added to on of the places in Q, there exists at least one place $q \in Q$, from which a token is removed. With this the total amount of tokens in all places in Q cannot rise, if none of them has a token to begin with $(\Box((\forall q \in Q : |q| = 0) \Rightarrow (\Box(\forall q \in Q : |q| = 0)))$. In other words, the upper limit for the total amount of tokens in Q is zero, if all places in Q do not contain a token. Therefore if a deadlock does not contain a token initially, it is one-safe.

But this is rather unpractical, since a initially unmarked deadlock cannot do very much, because places without tokens cannot activate transitions. Therefore it would be nice to have a property, which provides an upper limit for the total amount of tokens even if they are initially marked. The following property is inspired by the original definition of deadlocks.

$$Q \subseteq P \text{ is deadlock-mod} := \forall t \in T : |\{p \in Q : p \in {}^\bullet t\}| \geq |\{q \in Q : q \in t^\bullet\}|$$

This guarantees, that the amount of tokens consumed in Q by t is greater than or equal to the amount of tokens created in Q by t. With this the total amount of tokens in Q can never rise. Therefore if this property holds, the initial amount of tokens in Q is the upper limit. If the total amount of tokens is one or zero during the initialization, all places in Q are one-safe. With this it is now possible to prove one-safety for an entire Petri Net. If there exists some A, such that $A \subseteq \mathcal{P}(P) : (\forall Q \in A : Q \text{ is deadlock-mod}) \wedge (\bigcup_{Q \in A} Q = P)$ holds, and if the total amount of tokens in each set $Q \in A$ is smaller than or equal to one, the entire net is one-safe. One possibility to prove this for the item generator is shown in Figure 33. Each orange circle is a set $Q \in A$, such that the modified deadlock property holds. Additionally they contain only one token each during initialization and together they contain all places of the Petri Net. Therefore the item generator module is one-safe.

7.3 Place invariants

The more classical approach is to find a set of invariants [3], which together prove one-safety for the Petri Net. In order to prove invariants we need the transition matrix N. In case of the item generator module it is defined as follows:

N :	a	b	c	d	e	f	g	h	i	k	l	m	n	o
IN_NOT	1	1	0	0	0	0	0	0	0	0	0	0	0	0
IN_01	0	0	1	-1	0	0	0	0	0	0	0	0	0	0
IN_02	0	0	0	1	-1	0	0	0	0	0	0	0	0	0
IN_VAL	0	1	0	0	1	-1	0	0	0	0	0	0	0	0
IN_VALUE	0	0	0	1	0	0	0	0	0	0	0	0	0	0
IN_VALUE_NEG	0	0	0	-1	0	0	0	0	0	0	0	0	0	0
IN_RDY	-1	0	-1	0	0	0	0	0	1	0	0	0	0	0
GLOBAL_EN	0	0	0	0	0	1	0	0	-1	0	0	0	0	0
LOCAL_EN	0	0	0	0	0	1	-1	-1	0	0	0	0	0	0
LOCAL_EN_NEG	0	0	0	0	0	-1	1	1	0	0	0	0	0	0
OUT_NOT	0	0	0	0	0	0	0	0	0	0	0	0	1	-1
OUT_01	0	0	0	0	0	0	0	0	0	1	-1	0	0	0
OUT_02	0	0	0	0	0	0	0	0	0	0	1	-1	0	0
OUT_VAL	0	0	0	0	0	0	0	0	1	-1	0	0	-1	0
OUT_VALUE	0	0	0	0	0	0	1	0	0	0	-1	0	0	0
OUT_VALUE_NEG	0	0	0	0	0	0	-1	0	0	0	1	0	0	0
OUT_RDY	0	0	0	0	0	-1	0	0	0	0	0	1	0	1

Additionally the initial marking M_0 is defined in the same format:

$$M_0 = (0,0,0,0,0,1,1,0,0,1,0,0,0,1,0,1,0)^T$$

With the transformation matrix and the initial marking defined we now need to find suitable place invariants. Since IN_VALUE and IN_VALUE_NEG are inverse to each other, |IN_VALUE| + |IN_VALUE_NEG| = 1 should be true. This results in the following calculation for the invariant i:

$$\begin{aligned} i &= (0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0) \\ \vec{0} &:= (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0) \\ i \cdot N &= \vec{0} \\ i_0 = i \cdot M_0 &= 1 \end{aligned}$$

Since $i \cdot N = \vec{0}$ and $i \cdot M_0 = 1$ are true, the invariant |IN_VALUE| + |IN_VALUE_NEG| = 1 holds as well. Therefore the places IN_VALUE and IN_VALUE_NEG are one-safe. The same can now be done for the pairs LOCAL_EN, LOCAL_EN_NEG and OUT_VALUE OUT_VALUE_NEG, since they are inverse to each other too. In order to prove the rest of the places two additional invariants are needed. Those can be found by analysing the the movement of the token from IN_RDY. This token is transferred from its initial place to either IN_NOT or IN_01. In the case of IN_01, it is moved to IN_02 next and both paths meet at IN_VAL. From there it moves to GLOBAL_EN and back to IN_RDY. With this in mind the total amount of tokens in those places should always be one. Therefore the following invariant k should hold:

$$\begin{aligned} k &= (1,1,1,1,0,0,1,1,0,0,0,0,0,0,0,0,0) \\ k \cdot N &= \vec{0} \\ k_0 = k \cdot M_0 &= 1 \end{aligned}$$

Since the calculation for k holds, the place invariant |IN_NOT|+|IN_01|+|IN_02|+|IN_VAL|+|GLOBAL_EN|+|IN_RDY| = 1 is true and those places are one-save as well. The same applies for the place invariant

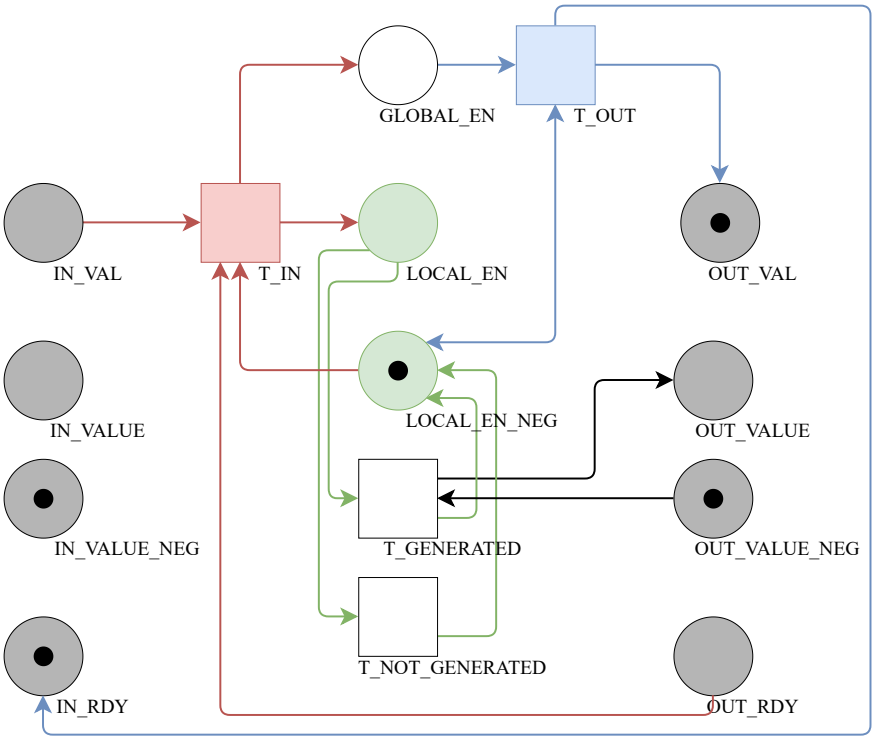


Figure 31: Item generator module without inhibitor arcs

$|\text{IN\_GLOBAL}| + |\text{OUT\_NOT}| + |\text{OUT\_01}| + |\text{OUT\_02}| + |\text{OUT\_VAL}| + |\text{OUT\_RDY}| = 1$, the calculation for this is the same. Since all places have been covered and are determined one-safe with those invariants, the entire item generator module is one-safe.

7.4 Model checker

To verify more complex properties we used PROMELA [7] for modelling with SPIN [8] [9] as interpreter. To translate a given Petri Net to PROMELA each place is converted to a channel and each transition to a process as explained in [10]. Additionally this translation can be expanded in order to handle inhibitor arcs. For a transition $t \in T$ each place $p \in {}^\bullet t$ translates to the condition $(\text{len}(p) > 0)$, because every place in ${}^\bullet t$ has to contain at least one token in order for t to fire. On the other side all places $i \in t_{\text{inhib}}$ have to contain 0 tokens to activate t. Therefore they can be translated to $(\text{len}(i) == 0)$. With this it is unnecessary to unfold the inhibitor arcs in the Petri Net. To give an example for this we translate the Petri Net shown in Figure 34 to PROMELA. The code is shown in Listing 1. We decided to use channel with a size of two, every place in the Petri Net only contains up to two tokens at any given time.

In order to actually verify a Net with PROMELA an interpreter is needed. We tried to use SPIN for this, but unfortunately it is not capable to verify our models. With SPIN each process receives and 8 bit wide index which limits the maximum amount of processes to 256. Therefore only a Petri Net with 256 or less transitions can be verified. The same applies to channel, which have a 8 bit wide index as well. With those limitations to the amount of transitions and places it is not possible to verify reasonably sized Belt Balancers with SPIN. For example the Belt Balancer shown in Figure 28, which is commonly used and rather small, consists of 296 places and 271 transitions. Therefore it cannot be verified by SPIN any more.

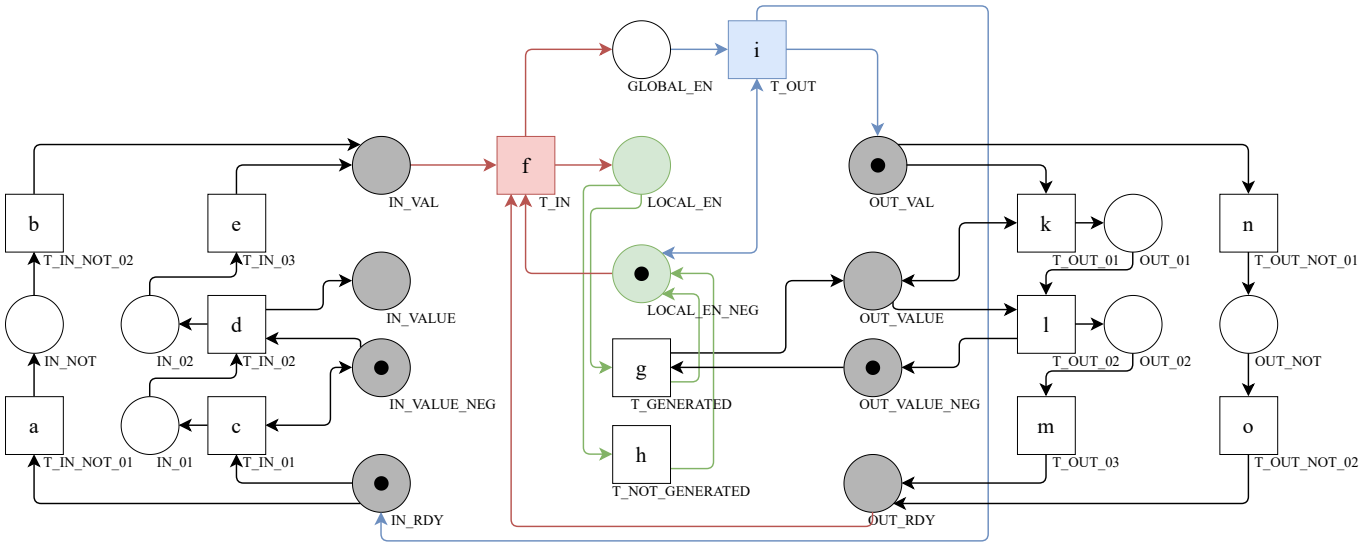


Figure 32: Improved and annotated version of Figure 17

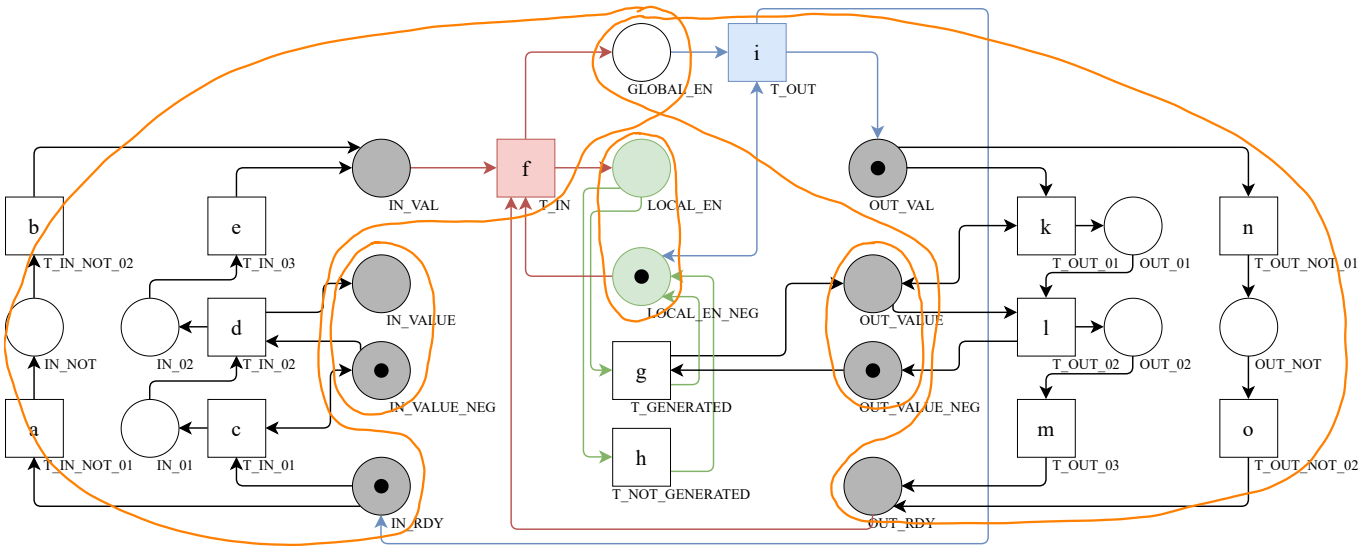


Figure 33: Item generator module with marked deadlock-mod sets

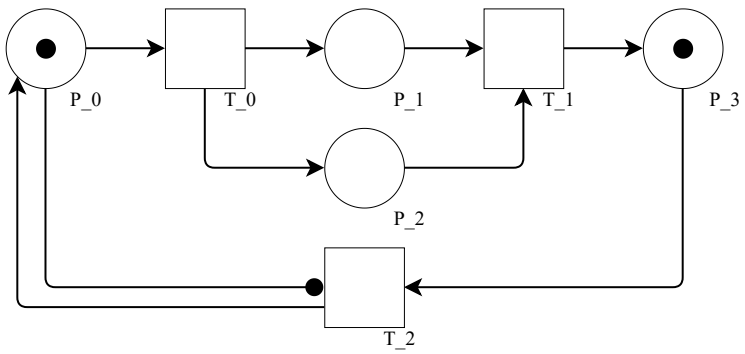


Figure 34: Petri Net represented by Listing 1

```
chan P_0 = [2] of {bit };
chan P_1 = [2] of {bit };
chan P_2 = [2] of {bit };
chan P_3 = [2] of {bit };

proctype T_0 ()
{
    do
    :: atomic
    {
        // check whether P_0 contains
        // at least one token
        (len(P_0) > 0) ->
        // consume a token from P_0
        P_0 ? _;
        // add a token to P_1 and P_2
        P_1 ! 0;
        P_2 ! 0;
    }
    od;
}

proctype T_1 ()
{
    do
    :: atomic
    {
        (len(P_1) > 0 && len(P_2) > 0) ->
        P_1 ? _;
        P_2 ? _;
        P_3 ! 0;
    }
    od;
}

proctype T_2 ()
{
    do
    :: atomic
    {
        // check whether P_3 contains
        // at least one token and
        // whether P_0 is empty
        // due to the inhibitor arc
        (len(P_3) > 0 && len(P_0) == 0) ->
        P_3 ? _;
        P_0 ! 0;
    }
    od
}
```

```
init
{
    // create initial marking
    P_0 ! 0;
    P_3 ! 0;

    // start petri net simulation
    atomic
    {
        run T_0();
        run T_1();
        run T_2();
    }
}
```

Listing 1: PROMELA code for 34

8 Conclusion

In the previous chapters we described a possible model for Belt Balancers consisting of multiple Petri Net modules, which can be arranged to model the behaviour of a Factorio Belt Balancer. Additionally properties commonly used in the Factorio community were defined via LTL, which can be verified with a modelling language like PROMELA. The translation from Petri Net to PROMELA works as well, but due to internal limitations the actual verification with SPIN does not, if the presented translation is used.

This could be solved by using an alternative verification tool like GreatSPN [11]. This Petri Net validation tool is a promising alternative due to its ability to verify LTL, CTL and CTL* [12] properties. But unlike PROMELA it cannot handle inhibitor arcs, those need to be unfolded. Furthermore this project could be expanded by increasing the scope of the verification. Right now we only focus on a small part in a complex and diverse logistical system. Instead of only verifying a Belt Balancer, a whole assembly line could be analysed to find bottlenecks and other causes of inefficiency. There are other logistical systems like logistical drones and trains as well, which come with their own unique set of problems to solve. A proper integration of those model checking and verification tools into the game is a sensible next step as well, which could be achieved by using Factorios modding API [13]. To sum this up we want to mention a seminar [14], which presents more Factorio related problems involving multiple scientific disciplines. Its authors also develop an interface to communicate with the game via external optimisers.

References

[1] Wube Software Ltd. *Factorio Website*. Aug. 2020. URL: https://www.factorio.com.

[2] Carl Adam Petri. “Kommunikation mit Automaten”. PhD thesis. Universität Hamburg, 1962.

[3] Wolfgang Reisig. *Petrinetze : Modellierungstechnik, Analysemethoden, Fallstudien*. 2010.

[4] Fred Kröger and Stephan Merz. *Temporal Logic and State Systems*. 2008.

[5] Amir Pnueli. “The temporal logic of programs”. In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 1977, pp. 46–57.

[6] ARM. *AMBA AXI and ACE ProtocolSpecification*. Oct. 2020.

[7] Gerard J. Holzmann. “Design and Validation of Protocols”. In: *Tutorial Computer Networks and ISDN Systems* 25 (1990). PROMELA proposal, pp. 981–1017.

[8] *SPIN Website*. Jan. 2021. URL: http://www.spinroot.com.

[9] Gerard J. Holzmann. “The model checker SPIN”. In: *IEEE Transactions on Software Engineering* 23.5 (1997). SPIN manual, pp. 279–295.

[10] Gerald C. Gannod and Sandeep Gupta. “An automated tool for analyzing Petri nets using Spin”. In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. 2001, pp. 404–407.

[11] Elvio Gilberto Amparore et al. “30 years of GreatSPN”. In: *Principles of Performance and Reliability Modeling and Evaluation*. 2016, pp. 227–254.

[12] Edmund M. Clarke and E. Allen Emerson. “Design and synthesis of synchronization skeletons using branching time temporal logic”. In: *Logics of Programs*. 1982, pp. 52–71.

[13] Wube Software Ltd. *Factorio modding API*. Feb. 2021. URL: https://lua-api.factorio.com/.

[14] Kenneth N. Reid et al. *The Factory must grow: Automation in Factorio*. 2021.