Strategy Generation for Distributed Smart Production Systems based on Networks of Timed Automata

Master Thesis
Group: DEIS103F18
Supervisor: Ulrik Nyman
Computer Science
Aalborg University
22nd May 2018

AALBORG UNIVERSITY
STUDENT REPORT

SUMMARY

In recent years Industry 4.0 and smart production have been research subjects of increasing interest. Aalborg University acquired a Modular Smart Production System made by the company Festo in order to support the research conducted at the university. This *learning factory* is used by several students, and this paper is the second of two, which takes a computer science perspective on how to obtain strategies for good uses of such modular factories. I many settings one would want the optimal use of such a factory, but with the modularity the state space is increasing fast. We therefore want to challenge the approach by investigating whether or not a near-optimal solution would have a practical use.

In this paper we present how a Uppaal Stratego model of a factory system can be used to generate strategies, which makes the foundation of code synthesis of distributed programs. We present how Uppaal Stratego models in general can be used to generate strategies for distributed behaviour, as long as one follows a set of modelling principles. The principles define the restrictions that locations and edges have, and applies those principles to the usecase. This includes how decision making is represented in the model, which edges must be controllable, and how hide private data from other processes. If the principles are followed it is trivial to divide the strategies into multiple strategies, which then makes up the foundation of the code synthesis.

The format of the strategies are presented, and parts of an abstract syntax tree (AST), which presents the information in the strategy. We show how the AST can be reduced and provides an implementation of it. Some ideas for reducing the AST is presented, which might have a great impact on the size of the synthesised code. Afterwards we present the overall structure of the PLC logic that controls the Festo system, and what parts need to change in order to utilise the generated strategy.

Strategy Generation for Distributed Smart Production Systems based on Networks of Timed Automata

Martin Kristjansen
Department of Computer Science
Aalborg University, Denmark
mk09@student.aau.dk

1. ABSTRACT

In this paper we present a method for generating strategies for near-optimal uses in a distributed system. The formalism used is networks of timed automata, and the usecase is Festo's Modular Production System. The tool used is Uppaal Stratego, and we utilise its new query format to hide private information, such that all actors in the distributed setting only have access to the information they would in the real system. We demonstrate how one can obtain such strategies by defining a set of principles one's Uppaal Stratego model must fulfil.

Author Keywords

Statistic Model Checking; Uppaal Stratego; Smart Production, Modular Production Systems, Industry 4.0

2. INTRODUCTION

In recent years Industry 4.0 and smart production have been research subjects of increasing interest. Aalborg University acquired a Modular Smart Production System made by the company Festo in order to support the research conducted at the university[1]. This *learning factory* is used by several students, and this paper is the second of two, which takes a computer science perspective on how to obtain strategies on good uses of such modular factories.

In this paper we present how a Uppaal Stratego model of a factory system can be used to generate strategies, which makes up the foundation of code synthesis of distributed programs. We present how Uppaal Stratego models in general can be used to generate strategies for distributed behaviour, as long as one follows a set of modelling principles. The format of the strategies are presented, and we present parts of an implementation of an abstract syntax tree, which presents the information in the strategy.

3. PROBLEM TO SOLVE

The modular smart production system that Aalborg University has acquired is the usecase of this paper. The system is created by the company Festo and the system allows for an easy changing of the setup of a production line. Since the physical setup is easy to adapt, one needs to be able to change the control logic quickly and easily in order to ensure a fast start of production.

We want to support users of such modular production systems by enabling automatic code synthesis. This synthesis is based on the products that the user wants to produce and the configuration of the production line. We then formulate our problem in this paper as:

> *How can one synthesise PLC logic for the Festo system, if the synthesis is based on strategies generated by networks of timed automata?*

This question relates to the topic of our previous paper[2], which contribution was the Uppaal Stratego models of Festo's modular smart production system. The relation between the two papers is explained in more detail in section 6.

4. RELATED WORK

We and other students at Aalborg University have previously worked with modelling the Festo system by using timed automata[3–5]. All three papers are subject to state space explosion, which hinders or limits the practical usability of the papers' results. Other students have used virtual commissioning in order to define how one can better use a given factory configuration[6]. Later, we used Uppaal Stratego in order to mitigate the result of state space explosion. The final result of the paper was a formal model of the Festo system[2]. This model forms the foundation of the strategy generation presented in this paper.

Uppaal Stratego[7] is a branch of the Uppaal family, which implements statistical model checking[8, 9], and is the tool used to define a formal model of the Festo system. The tool is used to obtain near-optimal uses of the system, and an advantage of the tool is that it is not as prone to state space explosion as traditional model checking is[10]. This tool is mostly used if a system contains uncertainties that need to be addressed in the model[11, 12], or to model non-linear systems[13, 14].

As part of the automation process we can include a virtual commissioning system in order to test our PLC logic. Virtual commissioning systems are used to make a virtual model of the production line, plant, or similar systems, which can be used to test the PLC logic. The purpose is to identify errors in the logic faster and before the physical setup is used in production[15, 16]. Although the method can limit the number of errors in production, it is usually not used by smaller or medium sized enterprises, since making the models can be time consuming[15–17]. Virtual models of the Festo smart production system have already been created and used in [6], which saves us the time it would otherwise take to create them.

PLC code synthesis and validation is not a generalised research subject in the sense that much of the research is based on very specific case studies[18, 19]. Generating PLC code is often seen as a scheduling problem, where one wants the shortest cycles. However, with that approach the lines of PLC code

are known but their order is not. Hence, there is no choice between several options of PLC code, but instead it offers an optimal use of predetermined PLC lines.

This paper sets itself apart by combining PLC synthesis with strategies on how to utilise a network of timed automata. The synthesis is not a scheduling problem of pre-determined PLC lines but a method for choosing which PLC logic to use when given a strategy and a factory configuration.

5. SELF CRITIQUE AND REFLECTION

As this paper is the second of two, we have already done a part of the work in order to present a solution for our specified problem. The first paper has been presented and have received some questions and constructive criticism, which we will briefly address in this section, along with how the questions can be answered or how they affect the papers' premises.

5.1 State Space Reduction and Exploration

State space reduction in Uppaal has been an important research subject and several papers have been written to describe the results. One of which is [20] where symmetry was implemented by the use of scalar sets. This made it possible to swap a state with a bisimilar state. The advantage of scalar sets is that it allows the user to define where symmetry appears in the modelling. Another attempt to implement state space reduction has been with discrete partial ordered reduction, as described in [21]. The problem with this approach is that it works well for un-timed systems or systems only using one clock. When multiple clocks are used, the states cannot be reduced in the majority of cases. The two papers are from 2003 and 2006, respectively, and were the latest papers that we could identify, where the goal was to make the Uppaal engine more efficient in terms of state space exploration. We have tried to modify the model to use scalar sets, but we were not successful. As the model has been made, the automata do not have symmetric behaviour. The main reason being that most automata represent a part of the Festo system, at a specific geometric location, and it is therefore not possible to replace one automaton with another within the system. If we were to use scalar sets, then we would need to rethink how the model is structured.

Other questions were related to Uppaal Stratego's ability in handling big models and how close a near-optimal solution is to being optimal. The very short answer to both questions is that we do not have a solid answer. Most papers, which use Uppaal Stragego, have models that consist of 2 or 3 processes, while we can quite easily define a system of 30 or 40 processes. So, we are trying to use Stratego's strategy generation on a larger system than others that we know of having done before us. The second question about near-optimal solutions also does not have a deep answer. In theory, we might be able to calculate the proportionality of having found the best strategy is, or how close we are. However, this calculation depends on the model and whether or not the system has a finite state space.

5.2 Tool of choice

Another Uppaal branch is Uppaal Cora[22], which has been suggested to us as another tool we could have used. Uppaal Cora is developed to find cost optimal traces in systems, and its methods for finding them are different than those of the Uppaal base. However, there are several reasons we chose Uppaal Stratego. One principal reason being that we want to challenge the approaches that try to identify optimal uses. Another more practical one is, that Cora does not have stochastic choices as a part. We do not have any stochastic choices in our models, but if the models were to be used in a real and practical workflow, then uncertainties in e.g. transport time is very likely to be introduced. We therefore consider our models in [2] as a good base for further research and Uppaal Stratego as the more suitable choice compaired to Uppaal Cora.

We have also been asked why we chose to use Uppaal, which uses networks of timed automata, and not a tool that uses a formalism like Petri nets. The tool TimeNET[23] can be used to model coloured stochastic Petri nets, and might also have been a good choice for modelling the Festo system. The tool is under active development with its latest release in August 2017. Coloured tokens would make it fairly easy to model distinctions between stages in a product's development, and we could model the stochastic behaviour in TimeNet. However, if we were to state whether or not e.g. TimeNet would be a good tool, we would need to delve deeper into how strategy generation was implemented as a feature, in order to conclude if its form was usable for our research. Another reason to choose a Uppaal branch over a tool that uses another formalism was that we could base the work on our previous work and experiences in [4] and [5]. However, we cannot disqualify tools like TimeNET, since we have not spent the necessary amount of time to make a proper evaluation of the tools' capabilities.

6. THREE STEP PROCESS

This paper is the second of two papers wherein we investigate the possibility of using statistical model checking in order to obtain a good production plan for a given system configuration. As presented in section 4 and in our previous paper[2], it is often the goal to obtain an optimal solution through analysis of the system. We wanted to challenge this approach in [2] with the Festo system as the usecase, in order to help determine if the optimal solution is the most practical method. In our first paper we presented a Uppaal Stratego model of the Festo system which we will reuse in this paper. The model has been extended to support multiple kinds of products and multiple kinds of work for the applications, and the implications of these changes will be the subject of later sections.

In [2] we presented a three step approach, which is illustrated in fig. 1. The contribution of our first paper is the Uppaal Stratego models, which we will use in this paper to obtain strategies that represent good uses of the production system. These strategies are used to guide the synthesis of PLC code that can be used by a physical production system. In this paper we present the second and third parts of the process in fig. 1, while still making a few, but important, additions to the models.

The paper is organised in this manner: First, we present the modification we have made to the models from our previous paper. Afterwards, we present principles that a Uppaal Stratego model must fulfil in order to obtain strategies useful

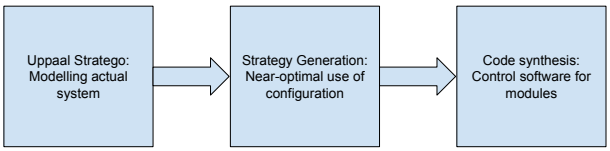

Figure 1. The three step approach of the two papers[2].

in a distributed setting. Then, we present the format of the strategies we obtain by using Uppaal Stratego and how the form should be understood. Thereafter, we look into the structure of the PLC logic Festo has provided with their learning factory, while also describing what logic we can keep and what we need to change. Finally, we specify what still needs to be done in order to make the findings in this paper useful in a practical setting.

7. UPPAAL MODEL MODIFICATIONS

The model we presented in our previous paper was not yet in a state where it was fully usable. We have made some modifications, and in this section we will describe the most important ones and their influence on the system.

The previous five templates[2] have all been updated in one way or another. They are placed in appendix A and are figs. 4 to 8.

7.1 Multiple products and works

The Uppaal model has been modified so it can handle multiple types of work and several kinds of products. The controller can non-deterministically choose any recipe as long the product still is in demand. We have also improved on how we keep track of which pallet has which recipe, since we simply used a boolean value in the old model. Previously, a pallet had either an active product or not, compared to now, where we need to bind a recipe's id to each pallet that has an item.

The concept of multiple works was simpler to implement, since we only changed the global array `works_id` from one to two dimensions. This array keeps track of which work ids a module can conduct and is represented with boolean values. Then, if the module with id `4` wants to check if it can conduct the work with id `2`, it simply checks `works_id[4][2]`, which will be true if the module is able to perform that specific work.

7.2 Channels and Started products

We have added a new channel `complete`, which is used by the `controller` automaton and all `module` automata. A lacking aspect in the old Uppaal model was what the controller would do, if there were no more items to produce. The old model did not define a limit on the number of items to produce, which made it easier to generate a strategy for the desired number of items. However, the strategy could initialise more products than were needed. Let us say we have two pallets in the system and we want to produce a single product. In the real Festo system we would make an order of a single product, which would result in a single product. In the Uppaal model both pallets could start producing a product since there were no guards to hinder the initialisation of a new item. Hence, both pallets could have started an item, even if only a single product is requested. The strategy, therefore, allowed for products to be started but not finished, which did not represent the real world and also introduced waste of resources.

Part of the solution to this problem was to add three elements: A new channel and two new counter arrays. The new channel is called `complete` and is used if the controller cannot start a new product on the requesting module. The channel is used in the transition between `req_incoming` and `idle` in the template shown in fig. 4. All the counters are shown in listing 1, where the new ones are added in order to solve the problem of starting more products than necessary. `started` and `MAX_COUNT` keep track of the number of started products and finished products, respectively. The guards for the two outgoing transitions in `req_incoming` make use of both counters, while the query generates a strategy where `count` equals `MAX_COUNT`.

```
1 int count[RECIPES] = {0, 0};
2 int started[RECIPES] = {0, 0};
3 int MAX_COUNT[RECIPES] = {1, 1};
```

Listing 1. All products counters defined in the model.

7.3 Uncontrollable Edges

As we are using Uppaal Stratego, one can specify if edges in the model are controllable or not. When we generate strategies, we generate choices of which actions to take in a series of states. However, some of the edges might be taken as an effect of the environment or other forces that affect the system we are trying to model. These actions are out of our control, and will therefore not be available in a strategy. The result is that we might not be able to say whether or not an edge ever will be taken or when it is taken, depending on how the model is constructed.

When we first presented our templates in the Uppaal Model, there were no uncontrollable edges. Uncontrollable edges had been necessary if we would have had uncertainties in e.g. transportation time or work time, but these aspects were not included in the design. However, all controllable edges might appear in a synthesised strategy, and it is not all transitions we want reflected in a strategy later on. An example is the edge between `req_pallet` and `read_pallet`, which calls a function as part of its update. Loading the pallet's values is important, but this needs to be done in any circumstance and is therefore not important for a strategy to specify. Thus, we change that edge to be uncontrollable.

There are several scenarios where changing an edge to be uncontrollable would not be ideal. If there are multiple outgoing edges in a location and we want to control the choice, then all outgoing edges of the given location must be controllable. An example is the `module` template's `read_pallet` location, where we either request the controller for instructions, conduct work, or send the pallet further along. Another pitfall is if the edge synchronises with other processes. If any edge in the transition is uncontrollable, the transition as a whole is uncontrollable. A surprising result of this is, that if even a single receiving edge is uncontrollable the entire transition is uncontrollable, even if the sending process' edge is controllable.

7.4 ModuleBranching Template

One of the biggest changes is that we have added a new template, which represents a module that branches. The reason for us to make a new template is that the old `module` template could not, in its `read_pallet` location, decide which path to send the pallet along. The template made the decision in its `transport` location, but the `module` was not fully in control of which path to use. The `tran` channel is urgent, so it must be used as soon as possible. An effect of this is that if one path is blocked, then the other must be taken, or if both paths are blocked, then the first free one must be taken. The `module` was only in control of the decision if both paths were free. This goes against the idea that the module can make an active decision on which path the send the pallet along.

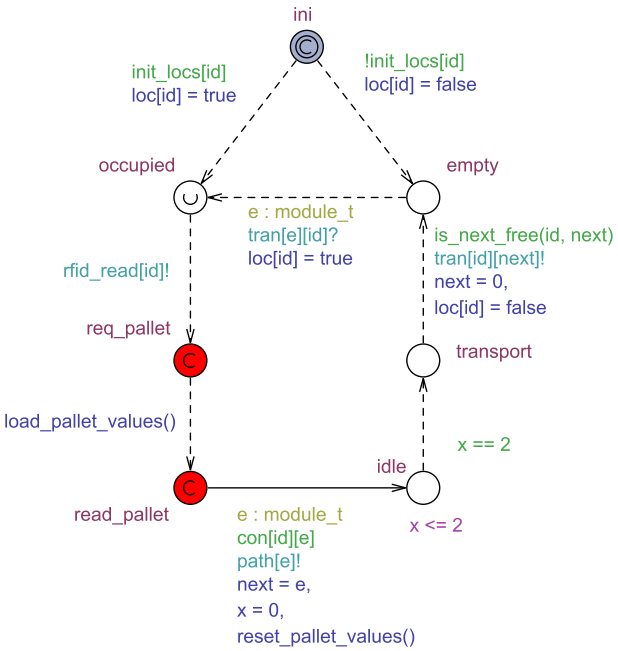

Figure 2. The new ModuleBranching template.

The new template is shown in fig. 2, and at first glance it seems the locations and edges are a subset of the `module` template. The behaviour of the two seems almost identical, but the outgoing edge in `read_pallet` represents behaviour not present in `module`. The edge specifies which path to use and stores that choice in `next`, which is then used by the `tran` channel later. Another important point to stress is that a module in the Festo system either has an application on top, which can conduct work, or it is a branching module. So when we module a branching module, we will do so with a `ModuleBranching` automata rather than with a `Module` automata.

8. STRATEGY CRITERIA AND NEW QUERY FORMAT

One of the open problems we presented in our previous paper was that we need to synthesise control software, which works in a distributed setting. This means that the different modules need to be able to make decisions based on the information stored on a pallet and on information it can request from the MES. The information can e.g. be the state of a given module; is it working, idle, or in an error state.

As part of the previous paper, we explained how we, at the time, did not know how to produce a strategy, which fulfilled those restrictions. Either we obtained a global strategy where the decision to take an action for a specific module was based on the internal state of another module, or we would produce one strategy for each module, which might not give a global near-optimal strategy. However, the upcoming version of Uppaal Stratego has a new way to generate strategies, which would allow us to create a global strategy, where every decision is restricted to a given set of variables and locations in the model. The method is based on *feature selection* in machine learning, where the features to use in a selection are defined by the user. The query has the format shown in eq. (1) and an example is shown in eq. (2).

$$strategy\ S\ =\ minE(goal)\ [<=\ bound]\ \{statevars\}\ ->\ \{pointvars\}\ :<>\ StopCond \tag{1}$$

$$strategy\ S\ =\ minE(time)\ [<=\ 2000]\ \{Process.location,\ k\}\ ->\ \{x,\ y,\ z\}\ :<>\ is\_system\_done() \tag{2}$$

The format contains two sets of features, which will be used in the resulting strategy. Both sets are comma separated lists of variables, but it is the variables in `statevars` that are used first to define what to base a decision on. If the variables in `statevars` are not enough to make a decision, then the variables in `pointvars` are used. The variables in `statevars` should be discrete and not floating point, since a vector is created for each unique combination of the variables. Hence, using floating points would result in an explosion in initial states. Using floats in `pointvars` would help in the splitting between decisions when needed, which does not result in an explosion of combinations.

In the example in eq. (2) there is a variable called `Process.locations`, which is an alias for all locations of all processes. Every location is interpreted as a boolean value, which is true if the process is in that given location. We want to limit the set of locations used to be those that show whether or not a module is working or idle, since the concept of an error state is not represented in the model. However, we want to include the state in the module template, which represent the point in time where a decision between multiple possible transitions is made. Other variables we want the strategy to use are those representing the information on the pallets. We can formulate the criteria as:

- The locations that represent whether a module is working or not, since the MES contains this information.
- The locations in the modules where different edges can be chosen, but we do not know in advance which edge would be optimal.
- The data written on the pallet in question, and no other pallets' information must be visible.

In short, the new query format allows us to restrict what information is accessible for a strategy. This helps us, since whenever a transition can be taken, it can only be chosen based on the information it can access in the physical system, which eases the task of using the strategy as the foundation of the code synthesis.

In fig. 5 the updated `module` template is shown. In order to fulfil our criteria we need to identify the locations representing the information we have access to. Whether or not a module is working is represented in the `working` location. The behaviour in this location is deterministic, but whether or not another module is conducting work can influence the decision made. The decision made results in one of three things: Work is conducted, the module requests the controller for instructions, or a pallet is sent along the conveyor belt. Which action taken is represented by the location `read_pallet`. Also, in order to be able to make these decisions, we need to have access to the information written on the pallet. However, we do not need all six pieces of the information on the pallet but only four of them, as shown in listing 2.

```
1 int glo_carrier_id; // Carrier ID
2 int glo_OPos;       // Order Position (in order)
3 int glo_PNo;        // Position Number (in the recipe)
4 int glo_operation;  // The next operation to conduct.
```

Listing 2. Used pallet values in the the strategies.

The variables we do not use are `glo_ONo` and `glo_resource`. Our model only considers a single order at a time, so there is no need to keep `glo_ONo`, since it would be the same value for all pallets. The latter is the resource value, which indicates which module should conduct work. However, this value is the one hindering the branching of work in the currently implemented Festo system, and we want to let any capable module be able to conduct the work. In order to make sure it is only the current pallet's values that are visible we use global values, which temporarily store the values. When a module starts to read a pallet, a series of committed states are entered, starting with `req_pallet`. The decision made can then also be based on values stored in the global variables, which are reset after a decision is made.

Specifically, we chose the variables in `statevars` to be all `read_pallet` locations and the controller's `req_incoming` location. These are the locations wherein each automaton decides which action to perform, and the model is constructed so only a single automaton can make a decision. The decision is then guided by the variables in `pointvars`, if necessary. The variables there are all `working` locations and the four global variables, which represent the values on the current pallet.

8.1 Generalised Criteria

When we have the criteria for the strategy of our Festo system, then we can generalise our criteria even further. The principles we use are applicable in other systems, such that code synthesis might be possible in cases much different than the Festo system.

The first set of principles is of how to model the locations wherein decisions are made, and these decisions must be represented in a strategy.

- All reachable states can at most have a single process that is in a location that represents decision making in the real system.
  - The decision making locations must be committed.
  - All its outgoing edges must be controllable.
  - If any of the edges synchronise, all receiving edges must be controllable too.
- All the decision making locations represent the variables in `statevars`.

The second set of principles is of how to access data, which can affect the decision making.

- A location or variable represents a value, which all other processes have to query at specific points in time.
  - The location or value is directly present in `pointvars`.
- A location or variable represents a value, which one or more processes have to query at specific points in time.
  - A location can set a value, which represents whether the location is active of not, via the update specification of all ingoing edges. The value is then altered back by all outgoing edges.
  - Whenever a process needs a value, that not all other processes can access at all times, the values are loaded into global variables. These global variables are then present in `pointvars`.

If we take a look at `statevars` and `pointvars` for our system, we can see that they fulfil those criteria. The variables in `statevars` are all `read_pallet` locations and the controller's `req_incoming` location. They represent decisions we want to have control over, and all their outgoing edges are controllable, and all receiving edges are controllable.

Then `pointvars` consists of all `working` locations, since they can be queried by all modules at any time, and the global variables of the pallet values. The pallet values are not accessible for all modules at all times, so they are loaded when a process enters `read_pallet` and is reset when leaving that location.

9. STRATEGY GENERATION AND FORMAT

As we mentioned in [2] we did not succeed in generating strategies as a result of a bug. Thanks to the help of Aalborg University now using an experimental version of Uppaal Stratego. This version does not contain the previous bug, but it is not available to the public yet.

As the bug has been corrected in our version of UPPAAL, we are now able to produce strategies, where we optimise the time usage of the production plan. In this section we present how the strategies are retrieved from Uppaal Stratego and how these strategies are formatted in JSON.

In reality, Uppaal Stratego is made up of several programs, where `verifyta` is one of them. This program is the verifying part of Uppaal, and the frontend of Uppaal calls it in

```
{
  "version": <version>,
  "type": <type>,
  "representation": <rep>,
  "actions": <actions>,
  "statevars": <statevars>,
  "pointvars": <pointvars>,
  "locationnames": <loc_names>,
  "regressors": <regressors>
}
```

Listing 3. Format of top level of JSON output.

order to run the verification of queries against a given model. `verifyta` has a command line interface, which one can use in order to work without the GUI. Below is an example of the queries we ran:

```
./verifyta -learning-method
1 -print-strategies ./output
distributed_festo.xml distributed_festo.q
```

The command uses the model defined in an xml-file and executes the queries from the q-file against the model. The query is formatted as described in section 8, and the complete query is part of the zip-file that accompanies this paper. If any of the queries obtain a strategy, then the strategy is printed to a specified output folder. The learning strategy is a splitting strategy, which results in rules based on binary decision trees. However, the leaves can contain several edges that can be taken, all associated with a weight. The greater the weight, the better it would be to take that edge.

The output of the strategy generation is in a JSON format, where the top level is shown in listing 3. The first three entries are constants and therefore do not provide new information between different strategies. `actions` is a map from integer to edge, such that all controllable edges have been given unique ids. `statevars` and `pointvars` are just two lists of the variables of the same name that we specified in the queries used to generate the strategy. `locationsnames` contains all the names of the locations in all instances in the system. Finally, `regressors` contains all the rules that we need to follow, and it will be these rules that are the basis of the code synthesis.

The entry of `regressors` is another JSON object, which entries are state vectors, where each represent the specific regressor for that state. Listing 4 shows an example of a regressor, which uses several rules to determine which action to take. The entry is a string concatenation of comma separated integers, which are enclosed in parentheses. All the integers represent the value of a variable in `statevars`, so the first integer represent the first value in `statevars`. Hence, the entry has the same number of integers as the number of elements in `statevars`, but only a single variable is 1, since we have constructed our model by the principles in section 8. Looking up the state results in a new JSON object, and the important field here is `regressor`. In our example the statevars is not enough to determine which action to take, so `pointvars` are used. `var` represents which variable in `pointvars` to use,

```
"(0,0,0,1,0,0,0,0,0,0,0,0,0)":
{
  "type":"point->act->val",
  "representation":"simpletree",
  "minimize":0,
  "regressor":
  {
    "var":2,
    "bound":0.5,
    "low":
    {
      "52":1,
      "53":49.9476
    },
    "high":
    {
      "var":0,
      "bound":0.5,
      "low":
      {
        "52":1,
        "53":17.8008
      },
      "high":
      {
        "52":35.2158,
        "53":13.9321
      }
    }
  }
}
```

Listing 4. Rules example.

and bound is the splitting value. In this example variable 0 is glo_0Pos. The comparison operator is <=, so if glo_0Pos equals 0.5 or less, the path along low is used. In that case there are no more levels to traverse through, and we have two actions with a weight. The first number, which is always an integer, can be used in the map actions to determine which edge is represented, and the second value is the weight. If the comparison returns false, then the high path is taken, and a new comparison would be made. This time with variable 1, which is glo_carrier_id, and the bound would also be 0.5. However, no matter what the result of the comparison is, the traversal will end.

We then process the JSON format in order to obtain a useful strategy. As the basic structure has been presented we can begin describing how to divide it into distributed programs. We can also use the strategy to determine the sequence in which the products are initialised.

9.1 Product Sequence

As we have obtained a strategy, it can be used to simulate a run of the system. This means that whenever an action is taken, it is taken according to the rules as defined by the strategy. We can use this to obtain the sequence in which the products would be started, since the Festo system must know the sequence that products are ordered, in advance.

Uppaal Stratego gives one the option to track variables and locations, such that we can get a description of how they changed over time. In our case, we would track the array started, which is updated every time a new product is initialised. The query format is as shown in eq. (3), which states that we want to track the values of a set of given variables, and the simulation is run under a given strategy. The integer in the query states the number of runs to perform. We have not implemented a parsing of such a simulation, but it is necessary if one is to produce two or more kinds of products within a single order. A parsing would then tell us in which sequence the products were started in.

$$simulate\ int\ [bound]\{< variables >\}\ under\ Strategy \quad (3)$$

10. DIVIDING A STRATEGY INTO MULTIPLE STRATEGIES

In order to make sense of a strategyâĂŹs format, we need some overall guidelines, so we are able to use it in a distributed setting. It seems the main problem is that all the behaviour is described in a single strategy, but this is actually a minor issue. The Uppaal models are made in such a way, that all reachable states have at most have one single module that makes a decision. No two modules can be in the read_pallet location at the same time. Therefore, an entry in regressors can at most include actions for one module. Each regressor then represents the entire decision tree of one module or the controller. The entry representing the controller is only needed during the simulation that obtains the order sequence and not in the code synthesis, so it can be disregarded.

The decision trees of the regressors are solely based on information that the modules have access to. As described in detail under the criteria in section 8, the modules have access to the other modules' state and the values of the pallet in question. Therefore, we do not need to make alterations in order to make them usable in a distributed setting, which was a serious concern in our previous paper[2]. In order to check which module to bind a regressor to, we use the statevars map in the top level of the JSON and the state in which the rule is used. The list of integers in the rule represents the value of the statevar in the same position. As we only have locations in statevars, the values are boolean and represent whether or not a process is in the given location.

Let us say we had a module process called module_0, and the first variable in statevars was module_0.read_pallet. Then, if the first value in the state representation in a rule is 1, then the regressor is specifying how module_0 decides which action to take. If we had $\mathbb{N}$ modules in total we would end up with $\mathbb{N}$ regressors in total, one for each module.

11. DEFINING AN ABSTRACT SYNTAX TREE

For implementing an Extended Backus-Naur form (EBNF) of the JSON format we use SableCC[24], which is a compiler compiler written in Java. We do not present the EBNF in this paper, but the grammar is included in the zip-file that accompanies this paper. Instead, we present parts of the generated abstract syntax tree (AST), in order to present what information we need and how to process the information.

```
1 strategy =
2     [actions]:transition+ [statevars]:state+
3     [pointvars]:point* [regressors]:regressor_list+;
```

Listing 5. Root node of AST.

Listing 5 shows the specification of the root node, which we chose to call strategy. It has four fields; actions, statevars, pointvars, and regressors. All four fields is a list of nodes, where actions and regressors must have at least one element, while the other two can be empty. Listing 6 shows the nodes that specifies the rules that drive the decision making of modules. A regressor_list node has a state and zero or more rules. The strategy generation might result in some states where we do not take actions, since we never reach that state. However, the state is still shown in the strategy, but does not have any rules associated with it.

```
1 regressor_list
2     = [state]:integer+ rule*;
3
4 rule
5     = {base} [action]:integer [value]:weight
6     | {splitting} [var]:integer [value]:boundd
7                   [low]:rule+ [high]:rule+;
```

Listing 6. Regressor and rule nodes of AST.

However, if there are rules, then there is either exactly one splitting node or one or more base nodes. base just contains an action and its weight, and the node is also a leaf node in the AST. splitting represents the splitting on a given

variable in `pointvars` with a specified bound. Other `rule` nodes are then referred to by `low` and `high`, which can either be one new `splitting` node or one or more `base` nodes.

12. REDUCING THE AST

The generated strategy has a form where only one of the processes in the model can take an action in any given state. There might be multiple options with different weights to choose between, and we want to reduce those options to a single one. The weight is insignificant, if there is only one option to take, since the weight is used to compare options.

Listing 7 shows an example of a regressor, which can be reduced significantly, if we purge options that do not have the highest weight. At lines 48 and 49 the action 52 would be chosen, and that same action would be chosen at lines 53 and 54. This results in a new situation, which also can be reduced. The `high` field on line 42 refers to a splitting, where both `low` and `high` refer to the same action. The `splitting` node can be replaced with a new `base` node, which has 52 as its action with weight 1.

If we apply the same kind of reduction to the `low` field on line 9, the result would be a new `base` node with the action 52 and weight 1. If we then were to rewrite the JSON in listing 7 to match that, the result is shown in listing 8. Reducing the AST will result in simpler programs, where it is required to make fewer boolean checks, since we only want the option, which has the highest weight. As we are able to map the values for `actions`, `statevars`, and `pointvars`, we are also able to translate the reduced example to something useful. The behaviour of the reduced AST is illustrated in algorithm 1, in order to give a sense of how one can understand a rule.

```
palletValues ← ReadPalletVaules ();
if palletValues.0Pos <= 2 then
    TransportPallet ();
else
    ConductWork (palletValues.operation);
end
```

Algorithm 1: Pseudocode of the reduced regressor.

13. FLOW OF PLC LOGIC

In this section we present how the logic is structured. This provides an understanding of the logic from a generalised perspective, which helps us when defining how to modify the flow in the automated code synthesis.

13.1 Tasks

Each PLC runs two programs called tasks. Each task is executed every 20ms, which is the time interval chosen by Festo, but it can be changed if needed. Global variables are defined and both tasks can access and modify these, and the variables are not reset when termination of a task is done. Hence, one task can change a global variable, which can affect the behaviour of the other task.

The first task, called `VisuProgram`, stores the input of the HMI (Human Machine Interface) into the global variables.

```
1  "(0,0,0,1,0,0,0,0,0,0,0,1)":
2  {
3      "type":"point->act->val",
4      "representation":"simpletree",
5      "minimize":0,"regressor":
6      {
7          "var":0,
8          "bound":2,
9          "low":
10         {
11             "var":1,
12             "bound":0.5,
13             "low":
14             {
15                 "var":2,
16                 "bound":0.5,
17                 "low":
18                 {
19                     "52":1,
20                     "53":22.9844
21                 },
22                 "high":
23                 {
24                     "52":1,
25                     "53":7.42188
26                 }
27             },
28             "high":
29             {
30                 "var":2,
31                 "bound":0.5,
32                 "low":
33                 {
34                     "53":3
35                 },
36                 "high":
37                 {
38                     "53":5
39                 }
40             }
41         },
42         "high":
43         {
44             "var":1,
45             "bound":0.5,
46             "low":
47             {
48                 "52":38.1069,
49                 "53":5.93848
50             },
51             "high":
52             {
53                 "52":22.4414,
54                 "53":18.7471
55             }
56         }
57     }
58 }
```

Listing 7. Regressor which is not reduced.

```
"(0,0,0,1,0,0,0,0,0,0,0,1)":
{
    "type":"point->act->val",
    "representation":"simpletree",
    "minimize":0,"regressor":
    {
        "var":0,
        "bound":2,
        "low":
        {
            "53":1
        },
        "high":
        {
            "52":1
    }
}
```

Listing 8. Regressor which is reduced.

This way, a human on the factory floor can adjust the settings, reset the system, or intervene if an error occurs.

The second task, called `MAIN`, is the one that controls the belt, reads the RFID chip of pallets, and communicates with the MES. This task is much more extensive than the first one, and most of the synthesising will be of this task.

Since each task needs to be run every 20ms the control structures are limited compared to those of a high-level programming language. The code must have terminated before the deadline has passed, so loops and recursion are just two examples or constructs that are discouraged. The control logic in `VisuProgram` is not subject to change in our code synthesis. We have no intention of changing the end-users options for interacting with the factory when it has started. However, we want to change how the modules interact with the MES and how the pallets are operated, so we will present the structure of `MAIN`.

13.2 MAIN task

The main task consists of parts that we want to change, in order to match the behaviour that the generated strategy dictates. The task consists of two primary parts; declarations and the program itself. The declarations consists of variables that state the module's IP address, the TCP ports used when communicating with MES. We to not want to change any of that information or how it is structured.

The program then is constructed out of several function blocks as shown in fig. 3. In this example there are 8 function blocks, each with its own declarations and flow. The `AppModul` function block contains information on how to use the application that a module has on top. This function is fairly simple, since it only conducts work, if the pallet states that the current module is next in the sequence to conduct work.

The other function block we need to alter is `StopperWithMesh`. This block is responsible for two things; reading the pallet and requesting instructions from

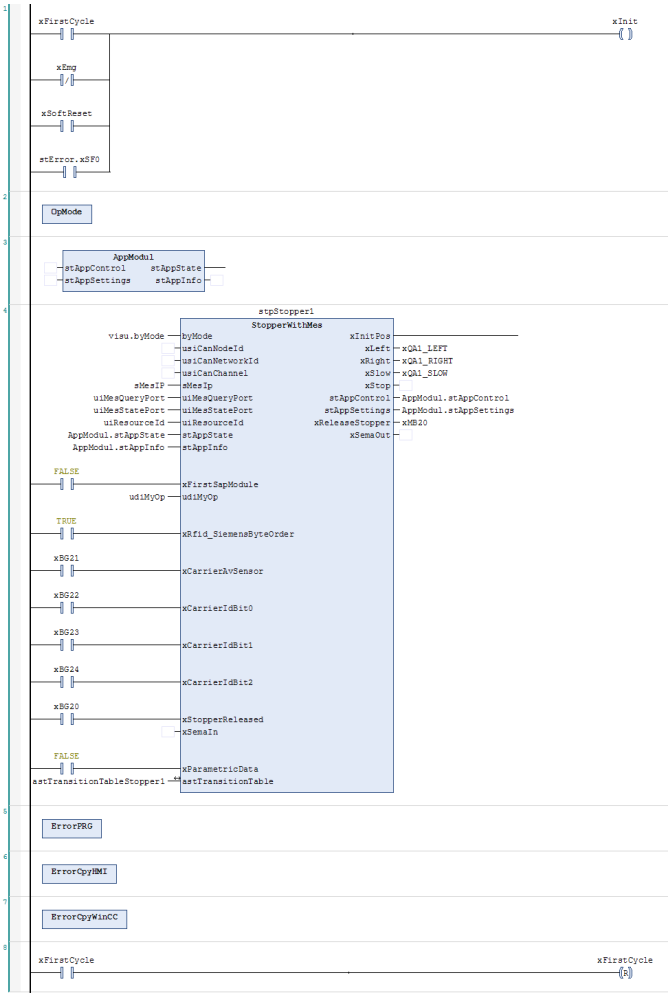

Figure 3. PLC flow of a MAIN program.

MES. However, it requests instructions from MES no matter what information is written on the pallet. Therefore, we want to split this into two function blocks; one that can read the pallet, and one that can request the MES. The functionality of the latter function block will then be used by the `AppModul`, since its decision making can depend on the state of other modules.

13.3 PLC format

The editor used for the Festo implementation is CodeSys[25], which can export and import PLC logic in a format defined by OpenPLC[26]. OpenPLC is an industry standard in an xml format. The format can be used to specify Structured Text, Ladder Logic, device configuration and more.

We, therefore, suggest that the code synthesis uses this format as the target platform, since it will be usable for the Festo system, but the logic would be readable by any editor that accepts OpenPLC.

14. CONCLUSION AND FUTURE WORK

In this paper and our previous paper, the goal was to have a fully automated process, which took a Uppaal Stratego model of a Festo system in order to generate a strategy and test the code in either virtual commissioning or on the physical system. We have not succeeded in implementing code generation. However, not only have we defined how useful strategies can be generated for the Festo system, but the same principles and model criteria can be used in other systems, such that Uppaal Stratego can be used to generate near-optimal strategies in other distributed systems.

We have also implemented an EBNF using SableCC and have suggested some strategy reduction principles as described in section 12. In addition we have outlined what parts of the PLC code we need to generate and what to reuse, and we have identified a standardised format in which to generate the code.

We therefore consider the primary contribution of this paper to be the principles of how to obtain a Uppaal Stratego model, which can be used to generate strategies, that are useful in distributed settings.

14.1 Implementing Code Generation

The end goal of our two papers was to have an automated process, which could generate PLC logic. This is not completed, but we have defined how generated strategies can be used for such synthesis. We therefore have the tools needed to complete the implementation of code synthesis.

14.2 Automated Uppaal generation from Experior Models

The Uppaal Stratego model has to be manually adjusted, when changing the setup of the system. In [6] virtual commissioning models of the Festo system was developed, and these models can be used to automate generation of a Uppaal Stratego specification. The specific virtual commissioning software is Experior, and the model is saved in a zip-like structure, where the setup is stored in clear text. It is therefore possible to use a Experior model to generate a Uppaal Stratego file, which represents the specific setup.

14.3 Simulating Product Orders

We also need to implement the simulation of the sequence of products, as described in section 9.1. Without this, the PLC logic might not be able to utilise the system when multiple kinds of products are produced, since the sequence is part of the generated strategy.

14.4 Fine Tuning a Model through Testing

In general the Uppaal Stratego model must be validated through testing. The testing would primarily be of the Festo system's defined timing constraints, but also to validate the correctness of the code generation itself. For this we think that using virtual commissioning would be an ideal platform for such fine tuning.

Acknowledgement

There are several people I would like to thank, and without whom the thesis would not have been possible.

Marius Mikucionis and Peter Gjøl Jensen from Aalborg university's Computer Science department, who have shared their knowledge of Uppaal Stratego with me. Also, Casper Schou and Steffen Mortensen from Aalborg university's Robotics and Automation department, since their knowledge of the Festo system, PLC logic, and virtual commissioning have been very valuable.

I would also like to thank my good friends Jens Christian Laursen and Aske Dybbro Andersen for continues proof reading, and to keep challenging my research with intelligent questions. Finally, I want to thank my girlfriend Jannie for all the support and encouragement, and our son Sebastian to help me to put my mind of the paper from time to time.

15. REFERENCES

1. M. Nardello, O. Madsen, and C. Møller, "The smart production laboratory: A learning factory for industry 4.0 concepts," in *Joint Proceedings of the BIR 2017 pre-BIR Forum, Workshops and Doctoral Consortium co-located with 16th International Conference on Perspectives in Business Informatics Research (BIR 2017), Copenhagen, Denmark, August 28 - 30, 2017.* (B. Johansson, ed.), vol. 1898 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2017.
2. M. Kristjansen, "Aiding smart production with statistical model checking," 2017.
3. M. Claus Jensen and A. Brandborg, "Optimizing modular factory configurations: Using timed automata and tabu search," 2016.
4. M. Kristjansen, "Modelling of festo production system," 2017.
5. M. Kristjansen, "Supporting parallel production in festo manufacturing system," 2017.
6. C. Blad, E. Straznickas, S. Ganeswarathas, and S. Koch, "Virtual commisioning of a reconfigurable manufacturing system," 2017.
7. Aalborg University & Uppsala University, "Uppaal stratego home," 2015. Note: accessed the 20-12-2017.
8. C. Baier and C. Tinelli, eds., *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, vol. 9035 of *Lecture Notes in Computer Science*, Springer, 2015.
9. A. David, K. G. Larsen, A. Legay, M. Mikucionis, and D. B. Poulsen, "Uppaal SMC tutorial," *STTT*, vol. 17, no. 4, pp. 397–415, 2015.
10. A. Boudjadar, A. David, J. H. Kim, K. G. Larsen, M. Mikucionis, U. Nyman, and A. Skou, "Statistical and exact schedulability analysis of hierarchical scheduling systems," *Sci. Comput. Program.*, vol. 127, pp. 103–130, 2016.

11. A. B. Eriksen, C. Huang, J. Kildebogaard, H. Lahrmann, K. G. Larsen, M. Muniz, and J. H. Taankvist, "Uppaal stratego for intelligent traffic lightsâŃę," in *12th ITS European CongressEuropean Congress and Exhibition on Intelligent Transport Systems and Services*, ERTICO-ITS Europe, 2017.

12. K. G. Larsen, M. Mikucionis, and J. H. Taankvist, "Safe and optimal adaptive cruise control," in *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings* (R. Meyer, A. Platzer, and H. Wehrheim, eds.), vol. 9360 of *Lecture Notes in Computer Science*, pp. 260–277, Springer, 2015.

13. K. G. Larsen, M. Mikucionis, M. Muñiz, J. Srba, and J. H. Taankvist, "Online and compositional learning of controllers with application to floor heating," in *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings* (M. Chechik and J. Raskin, eds.), vol. 9636 of *Lecture Notes in Computer Science*, pp. 244–259, Springer, 2016.

14. W. Ahmad and J. van de Pol, "Synthesizing energy-optimal controllers for multiprocessor dataflow applications with uppaal stratego," in *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I* (T. Margaria and B. Steffen, eds.), vol. 9952 of *Lecture Notes in Computer Science*, pp. 94–113, 2016.

15. P. Hoffmann, R. Schumann, T. M. A. Maksoud, and G. C. Premier, "Virtual commissioning of manufacturing systems A review and new approaches for simplification," in *European Conference on Modelling and Simulation, ECMS 2010, Kuala Lumpur, Malaysia, June 1-4, 2010* (A. Bargiela, S. Azam-Ali, D. Crowley, and E. J. H. Kerckhoffs, eds.), pp. 175–181, European Council for Modeling and Simulation, 2010.

16. C. G. Lee and S. C. Park, "Survey on the virtual commissioning of manufacturing systems," *J. Computational Design and Engineering*, vol. 1, no. 3, pp. 213–222, 2014.

17. R. Drath, P. Weber, and N. Mauser, "An evolutionary approach for the industrial introduction of virtual commissioning," in *Proceedings of 13th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2008, September 15-18, 2008, Hamburg, Germany*, pp. 5–8, IEEE, 2008.

18. H. Flordal, M. Fabian, K. Åkesson, and D. Spensieri, "Automatic model generation and plc-code implementation for interlocking policies in industrial robot cells," *Control Engineering Practice*, vol. 15, no. 11, pp. 1416–1426, 2007.

19. M. Dahl, K. Bengtsson, P. Bergagård, M. Fabian, and P. Falkman, "Sequence planner: Supporting integrated virtual preparation and commissioning," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 5818–5823, 2017.

20. M. Hendriks, G. Behrmann, K. G. Larsen, P. Niebert, and F. W. Vaandrager, "Adding symmetry reduction to uppaal," in *Formal Modeling and Analysis of Timed Systems: First International Workshop, FORMATS 2003, Marseille, France, September 6-7, 2003. Revised Papers* (K. G. Larsen and P. Niebert, eds.), vol. 2791 of *Lecture Notes in Computer Science*, pp. 46–59, Springer, 2003.

21. K. B. Holleufer, J. B. Rosenkilde, and M. Toft, "Discrete partial order reduction for uppaal," 2006.

22. Aalborg University & Uppsala University, "Uppaal cora home," 2014. Note: accessed the 04-05-2018.

23. Technische Universität Ilmenau, "Timenet home," 2017. Note: accessed the 04-05-2018.

24. ÃĽtienne Gagnon, "Sablecc home." Note: accessed the 01-03-2018.

25. CodeSys, "Codesys home," 2018. Note: accessed the 03-05-2018.

26. OpenPLC, "OpenPLC home." Note: accessed the 22-03-2018.

APPENDIX

A. MODIFIED UPPAAL MODELS

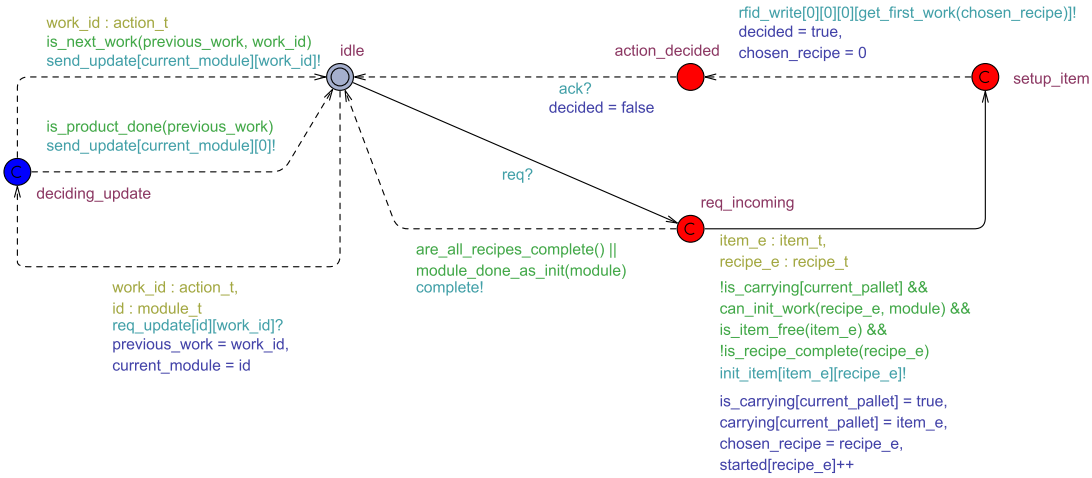

Figure 4. The updated controller template.

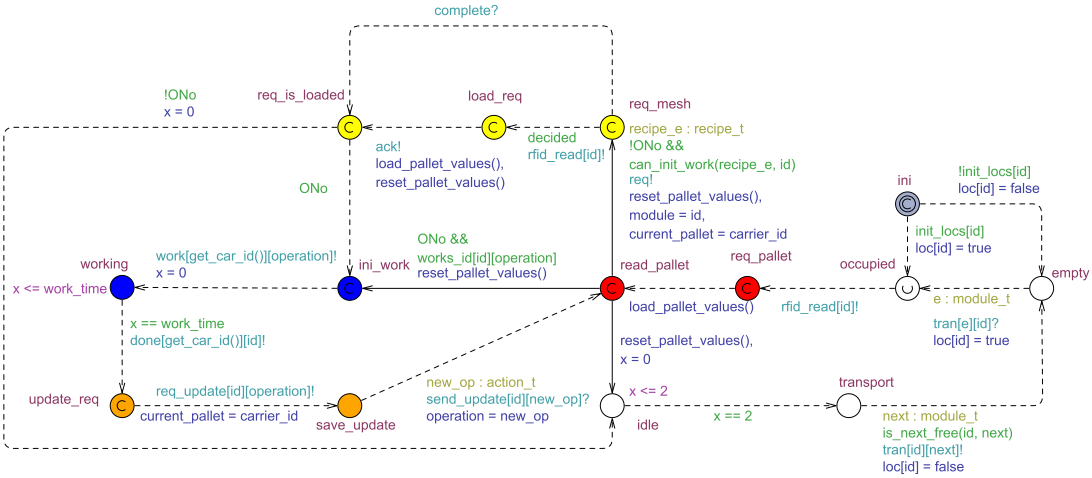

Figure 5. The updated module template.

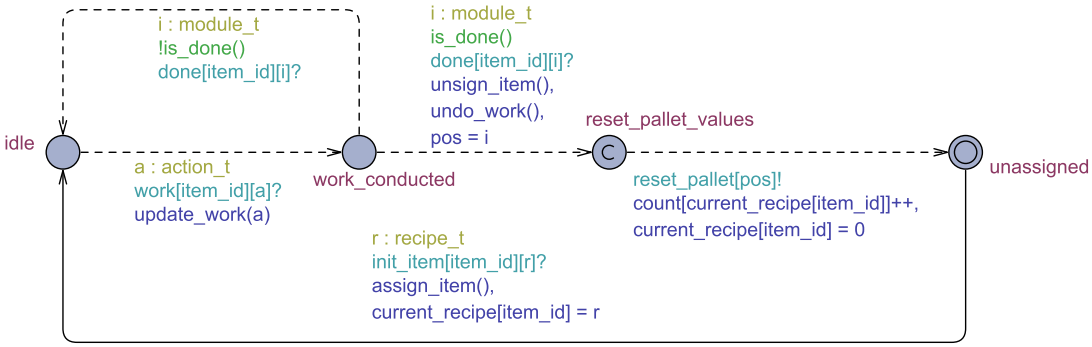

Figure 6. The updated item template.

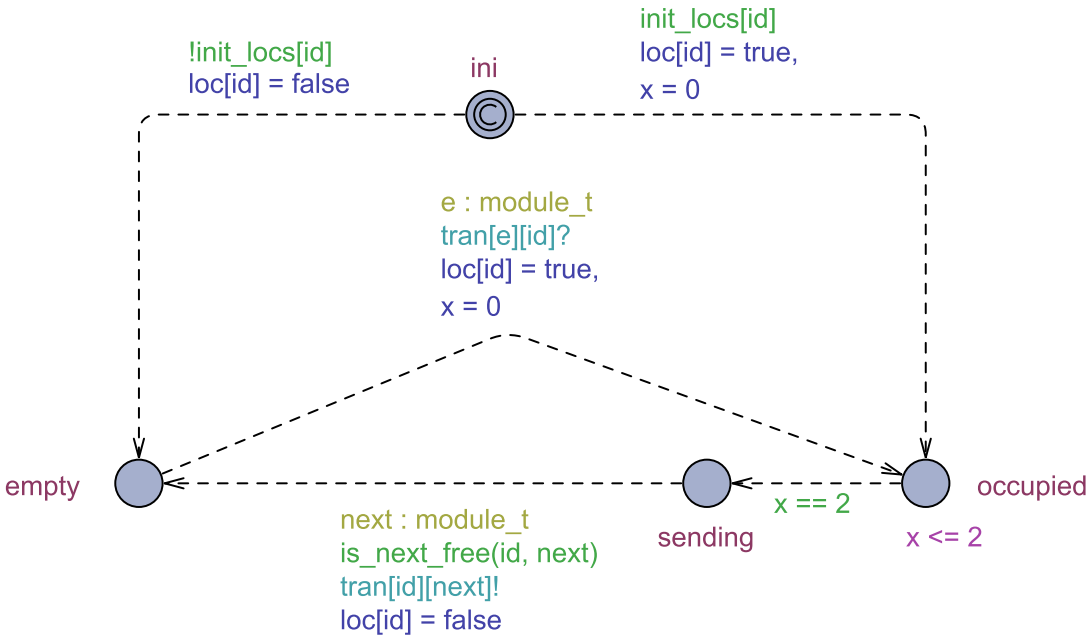

Figure 7. The updated belt template.

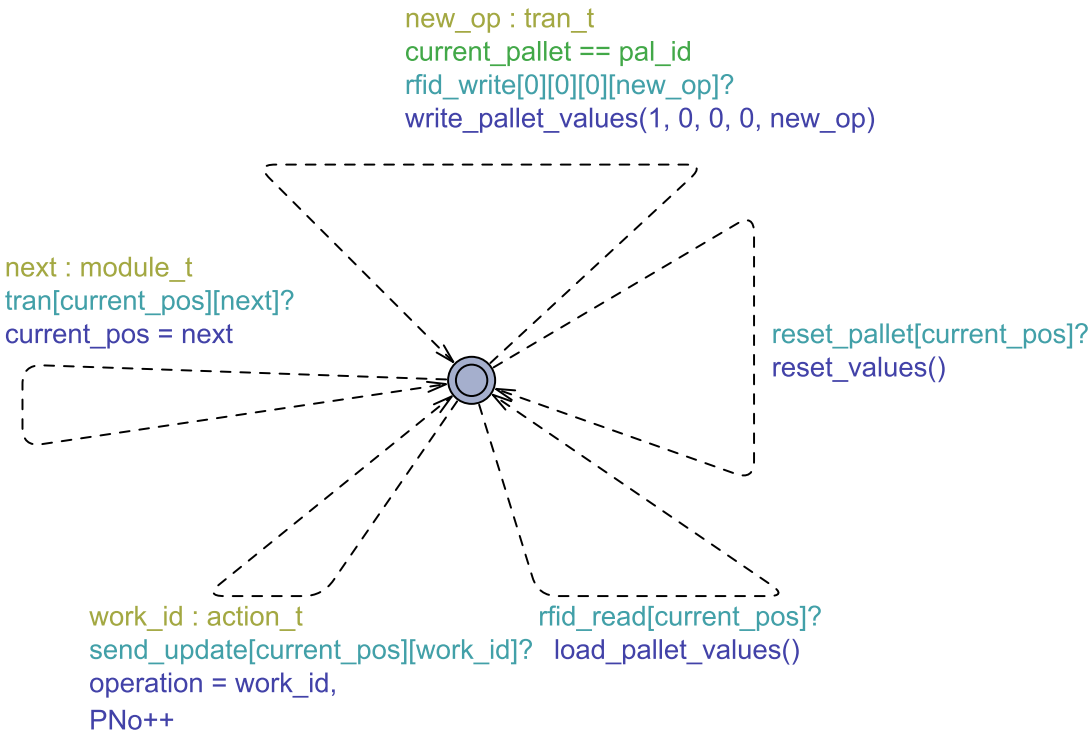

Figure 8. The updated pallet template.