Time-based Modeling and Verification of Flow Systems in UPPAAL

Public version

Wouter Geraedts

w.geraedts@student.ru.nl

20th December 2013

Bachelor's thesis

Radboud University Nijmegen
Institute for Computing and Information Sciences

Supervisor
Prof. dr. Frits W. Vaandrager
Radboud University Nijmegen
f.vaandrager@cs.ru.nl

Second reader
Dr. ir. G.J. Tretmans
Radboud University Nijmegen
tretmans@cs.ru.nl

Abstract

In this bachelor's thesis an approach for Flow Systems in UPPAAL is introduced. This approach is then used to implement a model for an industry use-case: the pre-production machine Lithography Wafer Scanner System by company A. It is during this implementation concluded that this model is not valid. Next, a tool is written to generate Gantt charts from traces of arbitrary timed UPPAAL models. Then, an optimal controller is synthesised for this model. Finally, an extra set of properties is also analysed for the new model. From this it is concluded that UPPAAL enables the time based modelling and analysis of complex real world Flow Systems. However, in order for this to be useful, it is recommended to ensure the validity of models by use of specification compilation.

Concerning this public version

At the request of company A, this bachelor's thesis has been redacted. The relevant figures have been rasterized and censored accordingly. The relevant pieces of text have been censored using uniformly sized black boxes:

- Numeric data has been censored as "[redacted]".
- Words have been censored as "[redacted]".
- Sentences have been censored as "[redacted]".

For certain terms synonyms have been introduced. These synonyms have been colored in a slightly grey hue. A few examples of these synonyms are company A, author A and pre-production machine.

Contents

1. Introduction

Model checkers and other formal verification techniques are used frequently in the hardware industry, but not as much in other areas such as embedded systems [9]. While there surely is an increase in the use of formal specifications and in the use of simulations, testing techniques like unit tests and simulation remain the standard [6]. Using model checkers to check specifications is difficult. Any meaningful specification quickly becomes too complex to verify due to an exponentially growing state space. Especially time-based models suffer from this problem. In this bachelor's thesis a use-case from the industry is investigated. To this end a time-based model in UPPAAL has been created, and some relevant properties of this non-trivial model have been looked at.

1.1. **Problem statement.**

The goal of this thesis is to show that time-based modelling in UPPAAL is ready for use by the industry. In a reflection on the commercialisation of UPPAAL [14], a few examples of use-cases from the industry are mentioned [8][15][11][18]. This thesis contributes to this list, by showing that UPPAAL is usable for Flow Systems in general. Flow Systems are typically machines or processes that have products flowing along several manufacturing steps, akin to an assembly line. Thus, the following will be answered:

> *Is it possible to create an useable timed-based model in UPPAAL of a non-trivial use-case from the industry, and check some meaningful properties from that model?*

There are two aspects to this question. Firstly, a non-trivial use-case from the industry had to be found and correctly modelled as an useable time-based model in UPPAAL. Real machines with complex timing constraint and safety requirements such as the exclusion of deadlocks are considered non-trivial. Furthermore a model is considered useable if engineers can work and adapt the model as required. To this end an approach that is generally usable for this type of systems is defined, and this approach is then used to implement this model. The model is also checked to conform to the guidelines of a *good* model, as defined in "A First Introduction to Uppaal" by Frits Vaandrager [16].

Secondly, some meaningful properties of the model had to be found and explored. Domain experts involved with the use-case surely must have some interesting questions. To be meaningful, these questions must specifically be hard to answer using simulation tools, and specifically well suited to answer through model checking.

For the syntax and semantics of UPPAAL, please refer to "A Tutorial on Uppaal" by Behrmann et al [3].

1.2. **Use-case.**

My supervisor Frits Vaandrager told me about a lithography machine by [redacted], henceforth called "company A". A lithography machine has been modelled by Adyanthaya [2], henceforth called "author A", as part of her master's thesis. Her work is continued in this thesis.

In the other thesis, a model is made in UPPAAL based on a specification in POOSL [1], or Parallel Object-Oriented Specification Language [17]. This model is then checked for:

- The absence of deadlocks.
- Whether or not the machine preserves the order in which it handles its product.
- Whether the machine violates certain time-based logistic rules.

It is useful to expand on the previous work on this model and continue where it was halted. The following future work is listed in the same thesis:

- Extend the model such that it represents all the modules in the wafer flow of the machine. The Atmospheric Wafer Handler is not implemented in the thesis by author A.
- Overcome state space explosion. Difficulties analyzing bigger non-deterministic time ranges are specifically refered to.
- Verify the same properties for a different version of the lithography machine. [redacted] However, the machine is a lot more complex as products on the production line are capable of overtaking each other. This added complexity could prove to be problematic.
- Create an automated translation from the existing POOSL specification to an UPPAAL model.
- For all the analysed properties, generate the ranges of parameters for which the properties are violated.
- Generation of Gantt charts.

In discussions with company A and Radboud University Nijmegen it has been decided that the following properties are of main interest for this research:

Generation of Gantt charts: These charts are very useful in the analysis of traces. Comparing two charts for irregularities is possible: one generated from a simulation from the POOSL specifications and one generated from UPPAAL. In Section 5.1 the implementation of a tool is described that converts UPPAAL traces to an event trace format specifically designed to be used in the generation of Gantt charts.

Controller Synthesis: Optimal throughput of lithography systems is key. This question is answered by synthesising an optimal controller, and comparing the timings of both the controller currently in use and this optimal controller.

Verification of previous results: In order to verify the previous work done, and to compare the new model, the properties analysed by her are analysed again here in this bachelor's thesis. Apart from checking for deadlocks, certain timing requirements are checked which originate from the thermal and quality requirements of the machine. These same requirements are also checked to hold for varying deterministic and non-deterministic timing constraints. Finally, the synthesis of controllers that comply to these properties is also attempted.

2. Background

2.1. **Wafer Scanner System.** In this section, the relevant details of the pre-production machine, are explained. This section is based on the thesis by Shreya Adyanthaya [2], slides by company A [5], and a conversation with Johan Jacobs from company A.

> Company A is one of the world's leading providers of lithography systems for the semiconductor industry, manufacturing complex machines that are critical to the production of integrated circuits or chips.

The pre-production machine is [redacted] studied as a case of a Flow System.

Lithography wafer scanning machines are used in the process to create integrated circuits (ICs) from silicon wafers. In this process (figure 1), wafers are coated with oxides and a photoresist layer. This photoresist is then exposed to ultra violet light from a photomask. All the exposed photoresist and the underlying oxides are then etched away by acid. The rest of the photoresist is then removed, leaving the oxides in the desired pattern. This process may be

repeated several times, for a multilayer Integrated Circuit, until the wafer is diced into separate chips.

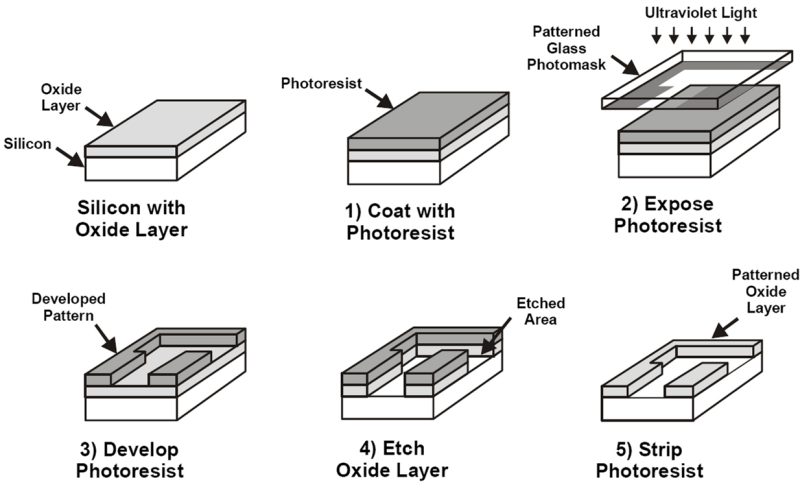

FIGURE 1. Simplified lithography process visualised.

EUV lithography is a next generation lithography technology for the creation of integrated circuits. It uses a lightsource with a wavelength of 13.5nm, and produces features with a resolution of 27nm [redacted] to 22nm [redacted]. The reduction in featuresize is desirable because this results in more integrated circuits per wafer. These circuits are, due to this reduction, also faster and more energy efficient. As EUV light is absorbed by all materials and gases, the wafers are exposed in a ultra-clean vacuum [19]. Furthermore, to prevent undesirable stress in the wafers, the machine has strict thermal requirements. Due to these thermal requirements, and due to the fact that wafers cool down or heat up over time, the machine has strict timing requirements. That is why the machine is divided into three sections, namely the Atmospheric Wafer Handler, the Vacuum Wafer Handler and the (Vacuum) Wafer Stage.

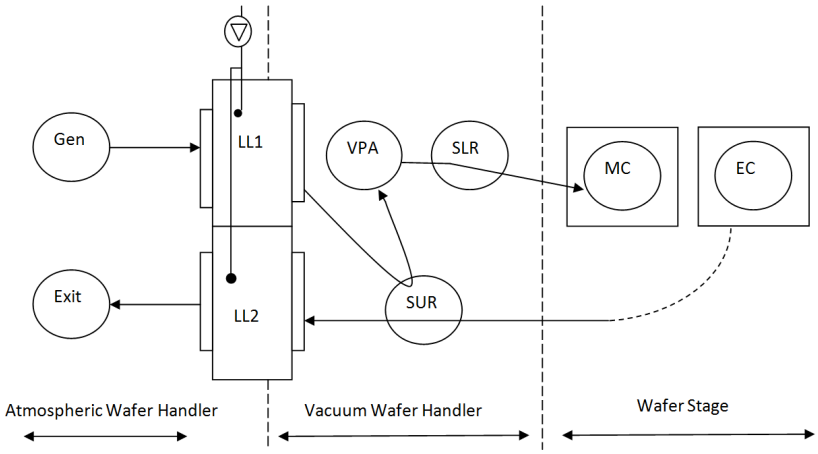

FIGURE 2. Schematic representation of the pre-production machine Wafer Scanner System taken from the thesis by Adyanthaya [2].

The Wafer Scanner System consists of the Wafer Handler and the Wafer Stage. The Wafer Handler handles the ingoing and outgoing wafers, as well as the positioning and thermal normalisation of the wafers in preparation for the Wafer Stage. In the Wafer Stage the positioning is finalised before exposing the wafers.

In the thesis by author A the Atmospheric Wafer Handler is not modelled. The behavior of this part of the machine is completely linear and thus not of interest in this thesis. The components for the rest of the machine, as shown in figure 2, are described here:

Chucks: The system has two chucks, one in the measure position called the Measure Chuck (MC) and the other in the expose position called the Expose Chuck (EC). After executing either the measurement or the exposure, the two Chucks swap their positions and roles. The Measure Chuck (MC) performs various measurements to calibrate the Chuck in preparation of the expose step. It measures the position and orientation using various markings on the wafer, and also the temperature to compensate for any potential stress in the wafer. The Chuck then swaps and exposes the wafer to the image on the photomask. This expose step takes a relatively long time, during which the other Chuck may expel the wafer to the Stage Unload Robot.

Stage Unload Robot (SUR): This robot has two functions: it takes a wafer from Load Lock 1 and puts it on the Vacuum Pre-Aligner. The Stage Unload Robot also takes wafers from the Chuck and puts it in Load Lock 2.

Because the Stage Unload Robot can take wafers from two different sources, there is a possibility of a deadlock. This deadlock occurs if the Vacuum Pre-Aligner, Stage Load Robot and both Chucks are holding wafers, and if the Stage Unload Robot has a wafer bound for the Vacuum Pre-Aligner. That part of the Wafer Scanner System can then not accept any more new wafers, and can not expel the wafers it currently has. To solve this problem a controller is required to control the Stage Unload Robot.

Vacuum Pre-Aligner (VPA): The Vacuum Pre-Aligner receives the wafer from the Stage Unload Robot, lifting it from the arm using a series of pins. It then aligns the orientation of wafers in the vacuum stage, and heats them up to normalise the temperature. Over time, the wafer cools down. The temperature of the wafer needs to match that

of the Chucks as close as possible when the wafer arrives. Thus the wafer needs to be exposed on the Chuck as quickly as possible, to minimise additional stress in the wafer. It drops the wafer onto the Stage Load Robot by lowering the pins once more.

Load Lock 1 (LL1) & Load Lock 2 (LL2): Because the wafers can only be exposed to EUV light in vacuum, the machine internally has a vacuum chamber. The vacuum has as an additional advantage that the wafers remain thermally normalised for a longer period of time. The Load Locks enable the exchange of wafers from the Atmospheric Wafer Handler to the Vacuum Wafer Handler, and vice-versa. Both the airlocks use the same pump for depressurisation. Thus the airlocks do not have the capability to depressurise at the same time. The depressurisation process takes a relatively long time. The airlocks can vent for pressurisation independently.

Stage Load Robot (SLR): This robot takes a wafer from the Vacuum Pre-Aligner and transfers it to an available Chuck in the measure position. Due to the aforementioned thermal requirements, it is imperative that the wafer is sent to one of the Chucks as quickly as possible.

2.2. **Adyanthaya's model.** Ideally a model is made from a specification [6]. For this bachelor's thesis, the UPPAAL model made by author A was used as a base to distil a better model from it. Model A was derived from a specification by company A written in POOSL.

To determine whether model A is of good quality, the model is checked to the criteria as specified by Vaandrager [16]. Most of these criteria are originally described by Mader, Wupper and Boon [13].

Object of modelling: As described by author A, the object of modelling is the POOSL specification of the pre-production machine as made by company A.

Purpose: author A has checked several properties. For each of these properties she adapted a nearly deterministic timed base model. Thus each of these models has a specific purpose. She checked properties such as but not limited to:

- Deadlocks are excluded.
- The time to exchange wafers is constant.
- The wafers exit the machine in the same order as they entered.

Note that the base model by author A contains clocks only used for the verification of the exchange time property. Thus not all models are purely made for specific purposes.

Traceable: Model A is not very traceable: in some cases it is quite unclear what is encoded. Symbols with names such as G, STOP2 and ABC are common. Furthermore, there is no clear separation of concerns: the Stage Unload Robot and Stage Load Robot encode work done by the Vacuum Pre-Aligner in reality. (Such as MOVEPINSTOWAFERVPA and MOVEPINSFROMWAFERVPA) The paper by author A [2] does not provide clarification on these subjects.

Truthful / valid: During my re-implementation of this model, several inconsistencies were found in model A. In a conversation with Johan Jacobs we concluded that this was indeed the case. More on these inconsistencies in section 4.1.

Simple: In terms of state space the model is very simple: the base model is nearly deterministic. The semantics of the model however are not clear, as the model is not very traceable. As such, it is not clear whether or not some variables are superfluous. Looking at the statistics outputted by UPPAAL, model A could have a smaller state space. UPPAAL explores 1590 states for model A, compared to 1432 states for my re-implemented model with the `ControllerOriginal` (see section 3.4.4).

Extensible and reusable: All templates in the model are parameterised. However, there is no uniform way to chain these templates such that it is possible to just add a new component. Because the pre-production machine is a complex machine, this is inherently difficult. However, in essence the machine is a assembly line. Thus it should be possible to easily add extra steps to this line. In model A this is not easily done.

Interoperability and sharing: The model is informally based on the POOSL specification by company A. The timing constraint have the same names as in the specification. Other than that, no relation to the specification is documented.

To solve these problems, the model A has been re-implemented using a new approach (see section 3.1). The goal of this re-implementation is to optimise the traceability, simplicity and extensibility of the model. Because this means that this new model is not directly based on the source (the POOSL specification) this raises doubts on the validity of the model. These concerns are addressed in section 4.2. In this thesis this model is henceforth referred to as "the new model", whereas the model on which it is based is referred to as "model A".

3. Implementation

In order to successfully implement a model of the pre-production machine wafer scanner system, a simple toy example from previous research is incrementally expanded on. An approach for flow systems introduced in this bachelor's thesis, henceforth called the "*FS-approach*", is used. Whilst implementing the model, the same steps as used by author A [2] are followed. During this process, certain problems with the validity of model A were discovered. When done, the performance of UPPAAL for the new model is considered. Finally, whether or not the new model is valid is put to question.

3.1. Approach for Flow Systems.

To create more traceable and simpler models, the *FS-approach* in UPPAAL is introduced. This approach attempts to achieve a better separation of concerns, by implementing the moving of the partial products between components. In model A this administration for moving wafer is implemented inline for each component, which is more error-prone and less clear.

In the *FS-approach*, specifically for the wafer scanner system, each component has a virtual space for a wafer w. A virtual component, named a Mover (figure 3), takes the wafer from one space and puts it on another space. Because this act of moving will actually cost time, a TimedMover (figure 4) has also been implemented to accommodate this.

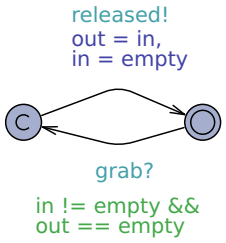

Figure 3. Mover template of the approach/nontimed.xml model.

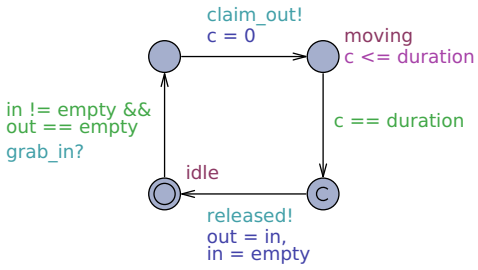

Figure 4. TimedMover template of the [redacted]/third.xml model.

To illustrate which synchronisation channels are necessary, a small UPPAAL model will be used: Given two components, A and B, and a virtual component Mover MAB, there are two synchronisation channels AB_IN and AB_OUT. The channel AB_IN communicates that a wafer can be taken from A by MAB, while the channel AB_OUT communicates that a wafer can be taken to B by MAB. For this synthetic example, these components are implemented as a simple *Sender* (figure 5) and a *Receiver* (figure 6). Because UPPAAL does not support multiple synchronisations on a single transition, the Mover template uses two transitions to synchronise on both channels. To force that both components A and B requesting a move are ready at the same time, the state between these two transitions is marked as *Committed*, meaning that both transitions must be taken right after each other, or result in a deadlock. Both components A and B must wait for the TimedMover to complete the move, thus it uses a broadcast channel AB_RELEASED to synchronise this completion.

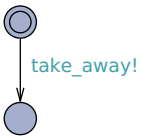

FIGURE 5. Sender template (or component A) of the approach/nontimed.xml model.

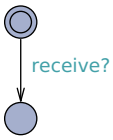

FIGURE 6. Receiver template (or component B) of the approach/nontimed.xml model.

To ensure that the state space for the *FS-approach* is minimal, all synchronisations are *urgent*. A deterministic model is ideal as a starting point, so that non-determinism can be added later if needed to check certain properties. If one wishes to delay these synchronisations (for this non-determinism), it is always possible to add extra non-urgent synchronisations. For traceability it is also nice to explicitly model the location of each wafer. For the wafers and the waferspots, the following definitions are used:

```
const int wafer_count = 1;

typedef int[0,wafer_count-1] wafer_t;
typedef int[-1,wafer_count-1] waferspot_t;

const waferspot_t empty = -1; //Representation for empty wafer-spot
```

The model approach/nontimed.xml in UPPAAL now moves a single wafer from A to B. To illustrate how the approach-model communicates a GraphViz graph has been generated using the TAFLOW program from LIBUTAP (figure 7). LIBUTAP is the LGPL parser library used by UPPAAL for parsing the timed automata stored in a xml-format. The graph shows which automata communicate with each other using synchronisations (in black) and variables (in blue).

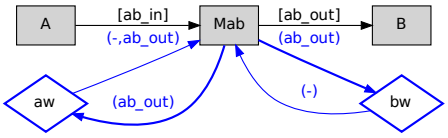

FIGURE 7. Communication in the approach/nontimed.xml model made by using the TAFLOW program from LIBUTAP.

3.2. **Proof of concept.** author A uses a small proof of concept to verify whether UPPAAL enables the implementation of a wafer scanner system. The concept implements the basic aspects of the system: the arrival, processing and delivering of wafers. As shown in figure 8, the model contains two generators, two exits, and two buffers through which the wafers must be passed. Because of these buffers, there is the possibility of a deadlock when M1 contains a wafers headed for M2, and vice-versa.

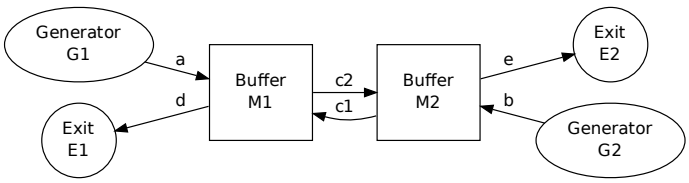

FIGURE 8. Concept for the two-way/deterministic.xml model.

The Generator, as shown in figure 9, models the part of a flow system from which is abstracted. It has an initial starting delay, and generates a wafer if possible once every PERIOD. The GENERATE_WAFER function takes care of all administration of picking the correct wafer to be generated. Here, W is the WAFERSPOT_T for the Generator.

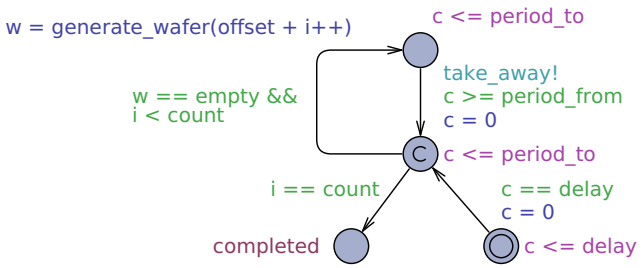

FIGURE 9. Generator template of the two-way/deterministic.xml model.

The exit, as shown in figure 10, models the delivering of the finished product. It continuously consumes wafers when possible, and marks them as “finished”.

Between the generators and exits, are two buffers. These buffers receive wafers and try to pass them on. There are two entry-points for wafers, and thus there are also two paths to process them. As the components in a flow system require time to process the wafers, the buffer-templates model a period required to pass the wafers along.

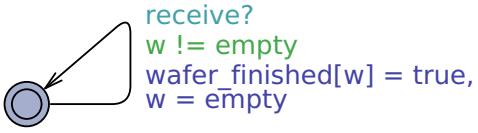

FIGURE 10. Exit template of the two-way/deterministic.xml model.

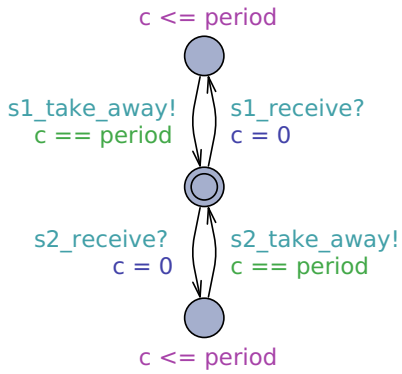

FIGURE 11. Buffer template of the two-way/deterministic.xml model.

Using the Mover template, this machine has been modelled in UPPAAL as two-way/deterministic.xml. By using the TAFLOW program from LIBUTAP, the graph in figure 12 is attained. Looking at these synchronisations, and comparing them to the structure in figure 8, the flow of wafers is clearly visible.

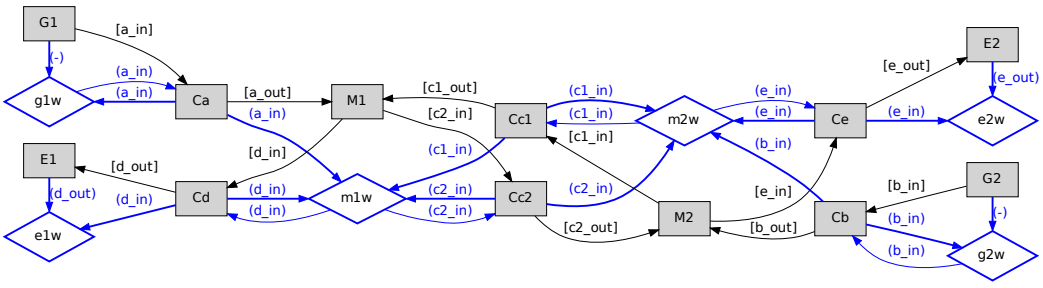

FIGURE 12. Communication in the two-way/deterministic.xml model made by using the TAFLOW program from LIBUTAP.

author A notes that for a specific set of timing constraints (page 22 of [2]) the aforementioned deadlock occurs. This is also the case for the new model. In TWO-WAY/DETERMINISTIC FIXED.XML the same fix as mentioned by author A is applied in order to solve the deadlock.

This proof of concept shows that the *FS-approach* also enables the implementation of the wafer scanner system, while giving a better separation of concerns, a better traceability and more simplicity.

3.3. **Performance considerations.** Generally, UPPAAL models with interesting properties contain some form of non-determinism. For completely deterministic models, running a simulation should suffice.

Some of the properties stated in section 1.2 add non-determinism in the following ways:

- When synthesising an optimal controller, all possible controller behaviours are iterated.
- When checking for non-deterministic timings, all possible combinations of the timing constraints are considered.

For each state UPPAAL includes so-called time delay transitions which globally increment the clock variables. These transitions are only enabled when no automaton is in an *urgent* or *committed* state, and when no automaton currently can take an *urgent* transition. In order to express that the machine does not pause unless explicitly allowed, all synchronisation channels should be marked as *urgent*. Urgent channels can intuitively be explained as that these synchronisations must immediately be executed whenever possible.

This raises the question whether this is allowed: do these synchronisations not take time? The model is expressed in time units of 0.1 seconds. Synchronisations in reality do not take more than 10ms (in general). It should be possible to abstract from this detail. Furthermore, the (virtual) synchronisations part of the Movers do not take time at all. They are not synchronisations in the object of modelling, but virtual aides that implement the extra administration of moving wafers around.

Parallel automata as employed by UPPAAL are modelled as interleaving processes. This interleaving increases state space, as every execution order is considered. An experimental extension has been made for UPPAAL which is capable of optimising the overhead introduced by this interleaving out [10]. However, this extension has not been added to the development snapshot of UPPAAL [3].

The state space of the model is reduced by using the following:

- precise bounded timing constraints (i.e., an activity always takes a singular amount of time).
- these urgent synchronisations.
- committed states whenever the state does not have either a timing constraint of urgent synchronisations.

Finally, the state size can be minimised, by economical usage of variables, and by using bounded types. The performance of the model should then be suited for further checking out the properties as aforementioned in section 1.2.

3.4. **Implementation of pre-production machine.** As a basis for the implementation of the new model of the pre-production machine, the implementation order used by author A has been used. Starting with the two-way proof of concept, components have been added one by one. For each of these components, first a trace was drawn up describing each synchronisation and timing constraint. Each of the trace states were annotated with the semantics of that state, as perceived by the author of this thesis. Using these augmented traces, the component was implemented for the new model. During this process some problems were discovered with model A. More on this in section 4.1. The resulting component has:

- a WAFERSPOT_T W parameter, for the physical location for wafers for this specific component.
- a local clock C, for the timing constraints of this component.

- for each state a name, referring to this added semantics. This greatly improves the traceability of the model, and enables the generation of readable Gantt charts (see section 5.1).
- renamed generic timing constraints, referring to this semantics (and not referring to the timing constraints originally given by company A).
- remodelled synchronisations conforming to the *FS-approach*. All other synchronisations are either removed, or moved to the controller.
- parameters for all the timing constraints and synchronisation channels, so that all templates can easily be reused.

3.4.1. *First iteration.* author A starts out by implementing a system with just the Chucks, Generator and Exit. For the new model, a Chuck Platform was also added, moving the functionality of swapping the Chucks to this new component. Because these Chucks swap, the target destination of the TimedMover also changes. To support this behavior, a new Mover had to be implemented: the DualTimedMover (see figure 13), which specifically selects the component with an empty waferspot to move to. Luckily, this is the only case in which the *FS-approach* could not be applied directly. This behaviour could also have been implemented in the preceding component, and two TimedMovers could have been used instead. This preceding component is the Generator in this iteration, and the Stage Load Robot in the next iteration. Because in reality the Stage Load Robot moves the wafer to the same location every time, the DualTimedMover is a better choice: It is the Chucks that swap from one location to the other. Implementing it in the preceding component would have therefore not been truthful, and would degrade re-usability of the Generator and Stage Load Robot templates.

Moving the wafer from the Chucks to the receiving component is handled by two TimedMovers. Because a TimedMover first synchronises with the sender, and then with the receiver, the DualTimedMover or a similar concept is not required in this case.

In the implementation by author A, the Chucks signal the Vacuum Pre-Aligner to load a wafer, and the Stage Unload Robot to take a wafer from the Chucks. In the new model these signals are sent to a controller instead.

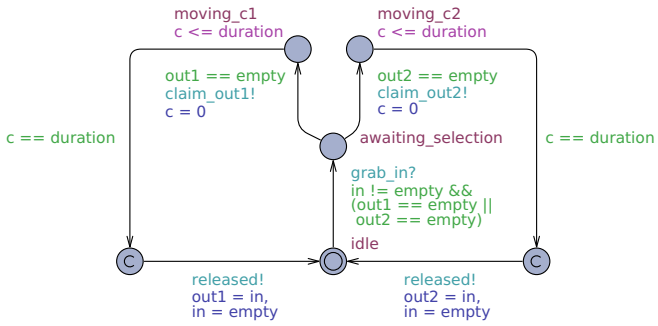

FIGURE 13. DualTimedMover template of the [redacted]/third.xml model.

3.4.2. *Second iteration.* In the second implementation iteration, the Stage Unload Robot, Vacuum Pre-Aligner and Stage Load Robot have been added. While the Vacuum Pre-Aligner and Stage Load Robot do nothing special, the Stage Unload Robot is interesting due to its complexity and because it does some interesting things: The Stage Unload Robot initially

waits in repose at the Vacuum Pre-Aligner. Whilst in repose, the Stage Unload Robot has the option to either:

- take a new wafer from the preceding component and put it on the Vacuum Pre-Aligner. In this case the wafers originate from the Generator. In the final iteration of the new model, the wafers are taken from Load Lock 1.
- take a finished wafer from one of the Chucks, and put it on the following component. In this iteration this component is the Exit, while in the following iteration this is Load Lock 2.

After finishing at either the Vacuum Pre-Aligner or Load Lock 2, the Stage Unload Robot goes back into repose. Choosing between these two options is the crux for the pre-production machine: If too many wafers are taken to the Wafer Stage by the Stage Unload Robot, a deadlock occurs. This choice also determines the processing time of the machine. Thus these decisions are made by the controller in the new model.

In model A these decisions are made inline. The Vacuum Pre-Aligner and Chucks request the Stage Unload Robot to load a wafer onto the Vacuum Pre-Aligner. The Chucks also request the unloading of wafers to the Load Lock 2. When the Stage Unload Robot is done loading the wafer onto the Vacuum Pre-Aligner, it signals that it is ready to load or unload another wafer. In the new model these requests and signals are also rewritten as synchronisations with the controller.

3.4.3. *Third iteration.* In the last implementation iteration, the Load Lock 1 and Load Lock 2 are added. These two components are slightly different from each other. The pump and vent controller by these two airlocks however are exactly the same, apart from their specific timing constraints. Thus for the pump and vent, the same UPPAAL template was used.

3.4.4. *Implementing the controller.* Finally, a controller had to be implemented that conforms to the behavior as implemented in model A. The intuition as distilled from this model is as follows: unloading wafers from the Chucks is done as soon as possible, and has precedence to loading wafers from Load Lock 1. Loading wafers can always be requested, but the request will only be honored if there is no loading in progress. The actual loading is ordered by the controller if there is a wafer standing by in Load Lock 1.

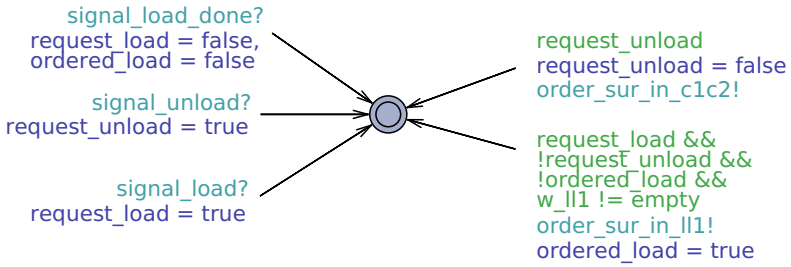

FIGURE 14. Original Controller template `ControllerOriginal`, from [redacted]/third.xml.

4. REFLECTION

4.1. **Validity of model A.** In section 2.2 it is concluded that the traceability, simplicity and extensibility of model A could be improved. Whilst implementing the new model, it turned out that the confidence of the validity of model A could also be improved upon.

Firstly, as already explained earlier, no clear separation of concerns is employed. Some activities seem to be implemented by the wrong components, suggested by their designation as defined in the POOSL specification by company A (see figure 15). Of course the designation of these activities could be interpreted differently, and model A could have implemented these aspects correctly. A clear naming scheme would help in this case.

Implemented by	Should be implemented by	Activities
Stage Load Robot	Vacuum Pre-Aligner	`lowerWaferOnGripperVPA` `movePinsFromWaferVPA`
Stage Load Robot	Chucks	`WSLiftWaferFromGripper` `WSEPinsToWafer`

Stage Unload Robot	Vacuum Pre-Aligner	`centerWaferVPA`
		`movePinsToWaferVPA`
		`liftWaferFromSURGripperVPA`

Stage Unload Robot	Chucks	`WSLowerWaferGripper` `WSFinishUnload`
Vacuum Pre-Aligner	Stage Load Robot	`releaseAndLiftWaferFromVPATableSLR`

FIGURE 15. List of activities for which their designation suggests that they should be implemented in different components.

Secondly, the order in which some activities are executed is wrong. [redacted], the Chucks start loading a wafer, even when it has no intention of grabbing a wafer. Just before the empty swap from the Measure phase to Expose phase is initiated, the activity `WSStartLoad` is executed. In figure 16 this activity is highlighted in red.

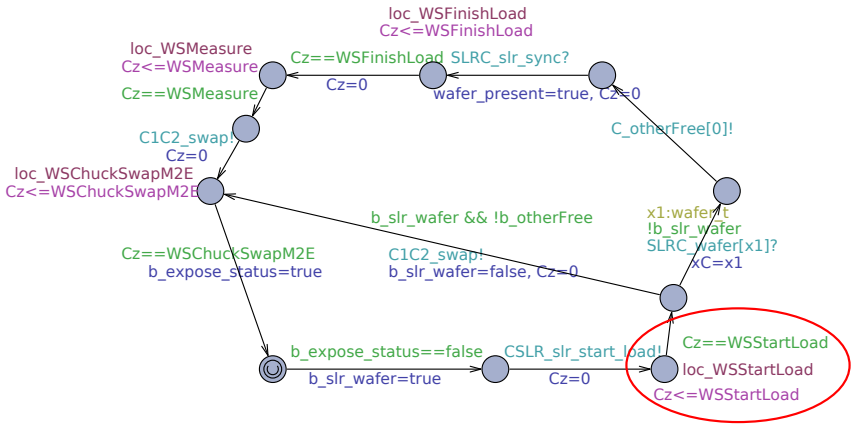

FIGURE 16. Excerpt from Chuck template from model A, with the `WSStartLoad` activity highlighted in red.

Third, the general convention for moving wafers used in model A is odd. The following sequence of activities is employed, in the Stage Load Robot and Stage Unload Robot primarily, but also in the Chucks and in the Load Locks:

- **Prepare** the transfer. This activity includes moving towards the receiving or sending component in the case of the Robots.
- **Synchronise** both components and transfer the ownership of the wafer by setting the ownership variable of the receiving component to the identifier of the specific wafer.
- **Start** to transfer the wafer. This activity most likely includes the preparation stages of the transfer which can only be executed after the actual transfer.
- Here some other activities may be executed, such as lowering a wafer onto the gripper of a Robot.
- **Take** or **give** the actual wafer.
- Perform some other post-transfer activity. In the case of the Vacuum Pre-Aligner and Stage Load Robot, the eccentricity between both components is measured.
- **Finish** the transfer. This activity most likely implies returning to a pre-transfer repose state. For a Robot this means moving away from the docking location in front of a component.
- **Synchronise** the completion of the transfer with both components, signalling that they are free to continue with other activities.

There are two problems with this: The ownership of the wafer is given long before the wafer is actually transferred. It would make more sense to transfer the ownership immediately after the actual transfer completed. Also, the ownership is not transferred, but copied. The receiving component assigns the wafer identifier to its local ownership variable. Unfortunately, the sending component keeps the value of its ownership variable.

Lastly, the deadlock property verified by author A does not work as intended. She added a location to the Exit template which will be entered once the machine has finished successfully. Then the property `A[] (not deadlock)` is checked. This property yield '`not satisfied`', because a deadlock occurs once the machine finishes successfully.

To fix this, a transition was added to this "finished" location (see figure 17), making sure that the machine does not go into a deadlock when finished. This property and this accompanying fix to the Exit template does not exclude a livelock. The intention of this property is to exclude the failure of the machine. Adding a name to the Exit location which symbolises that the machine has finished, the property `A<> Exit.finished` can be checked. It happens that this property is satisfied, given proper modifications to the Exit template. This stronger property excludes deadlocks, livelocks, and specifies precisely how the machine should behave.

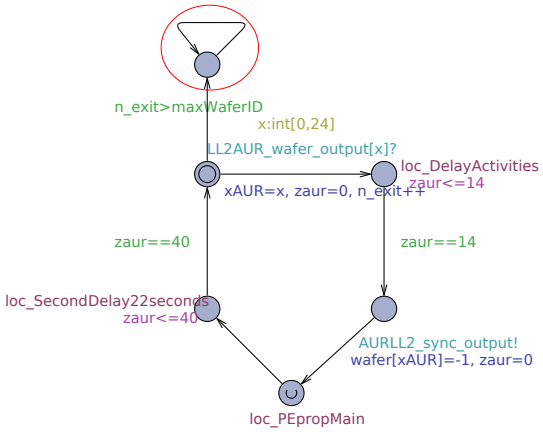

FIGURE 17. Exit template from model A, with the deadlock-averting location highlighted in red.

These problems could (partially) be attributed to a misinterpretation of the semantics of the model by the author of this bachelor's thesis. Acknowledging these concerns, and with possible solutions for these concerns prepared, the quality of the new model can now be inspected.

4.2. **Quality of the new model.** In order to improve the traceability, simplicity and extensibility of model A, the new model was made. Also an attempt was made to improve the validity of the new model. By using the same guidelines as used in section 2.2, the quality of this model is inspected.

Object of modelling: Model A, of which the object of modelling is the POOSL specification of the pre-production machine as made by company A. Also, some adaptations were made based on the solutions as described in section 4.1.

Purpose: For the purpose of optimisation, several models have been made for this bachelor's thesis. In these models, the same properties can be checked, for varying conditions. There are models with the timings as defined by company A, a range of deterministic timers set by an Initializer automaton and with non-deterministic timers varying whilst executing. State space is not wasted as in the thesis by author A, because all properties can be checked with either no overhead, or using a tailormade Observer which can be turned off. See section 5.3 for more on these Observers and on the varying timings.

Traceable: By assigning a name to nearly every automaton location, and a generic name to every timing constraint, variable and synchronisation, the traceability of the new model is significantly improved. With the new model it is possible to generate readable Gantt charts (section 5.1).

Truthful / valid: In the new model the issues outlined in section 4.1 are partially resolved. The Stage Load Robot is still responsible for lowering and lifting the wafers and the Stage Unload Robot also still performs the centering of the wafer for the Vacuum Pre-Aligner. Only the activities for which there was a fair amount of certainty that they should be moved, were moved.

Generally the validity of the model is quite dubious. The model is informally established based on a model which is also informally established on the original specification. Furthermore, the timings of these two models vary heavily: a trace for 10 wafer takes 3938 time units for model A, whilst the same run takes 4734 time units for the new model. In the thesis by author A is it implied that the timings generated from the model have been checked to the timings as used in the POOSL specification. It can be concluded that the new model is invalid.

Simple: Using the *FS-approach* introduced in section 3.1, a better separation of concerns is achieved. As the moving of wafers between components is implemented by the *FS-approach*, the components become simpler and therefor easier to understand.

By using the conventions as introduced in section 3.3, the state size and state space is minimised. The state space of the final model even grows linear in respect to the number of wafers for each run.

Extensible and reusable: By employing the *FS-approach*, all components use a uniform interface to connect to other components. Thus it is easily possible to swap, remove or add new components to the productionline in the model.

Furthermore, as explained in section 3.4.4, it is possible to use a different controller. This is mainly useful for controller synthesis (see section 5.2).

Interoperability and sharing: No formal relation to model A or the specification by company A has been given. Ideally, the new model would have been generated directly from the specification.

From this it can be concluded that the new model improves on the traceability, simplicity and extensibility of model A. However, the validity and the interoperability of the model could be improved.

It is possible to informally ascertain the validity of simple models. With complex models of big machines such as the pre-production machine, this is possible by comparing Gantt charts for example. To get truly valid models, the only option is to generate them directly from the specification by use of a compiler [6].

Note that validity of the model was never a goal in this bachelor's thesis. The *FS-approach* introduced in this thesis could easily be used in further work for that purpose.

5. Analysis

5.1. **Gantt charts.** In her thesis, author A writes that she would like to have Gantt charts to be generated from UPPAAL traces. These charts are a useful tool, to be used in debugging but also in verification of the model. It should be possible to implement this such that Gantt charts could be generated from any time-based UPPAAL model.

Since version 4.1.15 of UPPAAL, the beginnings of a special Gantt chart view can be seen in the `ConcreteSimulator`. This view is not yet functional, and is thus not yet suitable for the analysis purposes as required for this thesis.

Adyanthaya suggests looking at libutap for the implementation of this feature. There is a single example program in this library, `tracer.cpp`, which proved to be very useful. This piece of code has the ability to read a compiled model and a `.xtr`-trace, and outputs a readable trace. With this program as a basis the search for suitable tools to visualise Gantt charts started.

ResVis was selected: a tool by the Eindhoven University of Technology that reads the Octopus-format and visualises the traces contained as Gantt charts. ResVis enables fast navigation in these charts and provides some performance analysis tools.

A tool was made that converts traces from time-based models in UPPAAL to the Octopus-format. This tool, named UPPAAL2OCTOPUS, is released under the LGPL-license. The source-code is available on GitHub at `https://github.com/Wassasin/uppaal2octopus`.

5.1.1. *Requirements on the model.* To generate Gantt charts for a time-based model, it is necessary to include an immutable clock in the model to reference the time. UPPAAL2OCTOPUS uses the global CLOCK C as the reference for time. Furthermore, to have a meaningful Gantt chart it is required to annotate the locations in the model with proper names. Thus UPPAAL2OCTOPUS ignores locations which do not have a name defined.

5.1.2. *Intermediate format.* Internally UPPAAL compiles a model to be run in a Virtual Machine. This intermediate format representation can be output by the UPPAAL VERIFYTA-server, using the following command:

```
$ UPPAAL_COMPILE_ONLY=1 verifyta model.xml > model.if
```

To generate traces for Gantt charts, more information is required than contained in the `.xtr`-trace files. The trace files reference to the identifiers in the compiled model. This compiled model contains the following:

layout: A list of constants, variables, clocks and locations used in the model. Variables list their minimum and maximum values, and for locations their name and flags are listed.

instructions: A list of instructions used by the model. Describes how expressions are calculated, including functions used in the model. The instructions are written in a form of assembly language.

processes: The concrete automata instantiated for the model. Including the initial location of the process and the name of the process.

locations: The states of the various automata in the model, their flags (initial, urgent, committed, etc.) and invariant expression.

edges: Transitions between the states in the various automata, including the guard, update and synchronisation expressions.

expressions: Expressions such as guards, updates and invariants in use by the model. Contains a textual representation of the expression, and lists which variables the expression concerns.

Ideally the Gantt charts should contain the names of the states referenced to in the trace, along with the relevant timing constraints which can be found in the invariants of those states. These two pieces of information can be extracted from this file, for a given trace of locations. However, as explained in section 5.1.4, two bugs complicate this.

5.1.3. *`.xtr`-trace files.* The trace files as generated by the UPPAAL GUI program are not human readable and do not give a complete representation of the trace. All identifiers used in this files refer to the layout elements of the compiled UPPAAL model, as described in section 5.1.2. The file describes trace states and transitions, alternatingly.

A state begins with a series of identifiers, separated by newlines and terminated with a single dot. These refer to the location identifiers for all the processes in that state. This is followed by a set of clock constraints, each a set of three integers i, j and their value, followed by a dot. The least significant bit of the value represents the strictness of the constraint, whereas the rest is bitshifted and used as the difference between the clocks referenced to by i

and j. Finally, a set of integers is given, followed by a dot. These values represent the value of each of the variables in the current trace state.

A transition is represented by a pair of integers for each process in a model, followed by a dot. These integers refer to the identifiers of locations x and y, such that it encodes that for the current trace transition and for the current process transition $x \to y$ is taken.

A clock constraint should be read as data structures called Difference Bounded Matrices, or DBMs, which offer a canonical representation for this timed constraint system. This data structure represents a weighted directed graph, where the vertices correspond to clocks (with a single *constant* clock "$\mathtt{t}(0)$") and the weights of the edges represent the bounds on the differences between pairs of clocks. UPPAAL generates a minimal and canonical representation for these constraints [12]. This is why not all clock constraints are directly represented but may indirectly be inferred from the other constraints.

Ideally the constraint $c - \mathtt{t}(0) \leq t$ is directly read from the value of $w(c, \mathtt{t}(0))$, where $w(x, y)$ returns the weight value for the edge $x \to y$. If this constraint is not available, the constraint can be inferred by finding a path $c \twoheadrightarrow \mathtt{t}(0)$ in the DBM graph. In UPPAAL2OCTOPUS this is done by a breadth-first search over this graph. Given a path $x_0 \twoheadrightarrow x_n$, the t in the constraint $x_0 - x_n \leq t$ can be calculated as following:

$$t_{x_0 \twoheadrightarrow x_n} \geq \sum_{i=0}^{n-1} w(x_i, x_{i+1})$$

Now that the value for the clock `c` can be read from the `.xtr`-trace, all that is required to generate Gantt charts is available to UPPAAL2OCTOPUS.

5.1.4. *Intermediate format bugs.* While implementing the `.xtr`-trace parser, two problems in the UPPAAL VERIFYTA-server were encountered that have yet to be resolved:

(1) Invariants are not correctly represented in models in the intermediate format as generated by the UPPAAL VERIFYTA-server. Instead of a proper expression for the invariant such as "`c <= delay`", the compiled model references to the incorrect expression "c". Ideally the Gantt charts would be annotated with the invariant of each state, to refer back to the relevant timing constraint which the invariant describes. Due to this bug, this is currently not possible when parsing the `.xtr`-trace format. The relevant bug report can be found here: `http://bugsy.grid.aau.dk/bugzilla3/show_bug.cgi?id=555`.

(2) If a location in a UPPAAL model contains a function in its invariant, this location is incorrectly represented in the intermediate format as generated by the UPPAAL VERIFYTA-server. Instead of an entry as a location in the layout-section of the intermediate format, it is listed as a constant with the value "return". The relevant trace state can subsequently not be outputted by UPPAAL2OCTOPUS, without interpreting this entry as a location. This results in gaps in the Gantt charts for models with functions in invariants, such as the models with nondeterministic timings described in section 5.3. The relevant bug report can be found here: `http://bugsy.grid.aau.dk/bugzilla3/show_bug.cgi?id=556`.

Because UPPAAL is not open source software, this bug can not be fixed without contacting the authors. But there is another solution available.

5.1.5. *Human readable format.* Traces can also directly be generated by the UPPAAL VERIFYTA-server. These traces are in a human readable format. The previously mentioned bugs do not

appear to affect the generation of these traces. Also, it is much easier to parse, because it is human readable and because the values and bounds for all clocks and variables are directly outputted. Human readable traces can be generated by the UPPAAL VERIFYTA-server, using the following command:

```
verifyta -y -t2 model.xml query.q 2> model.trace
```

Contrary to `.xtr`-trace traces, a trace in the human readable format can not be exported from or imported by the UPPAAL GUI program. Furthermore, it does not resolve the symbols of guards, updates and synchronisations in its output, but maintains the symbols as defined in the templates.

A parser for this human readable format has been implemented in UPPAAL2OCTOPUS, so that the program now supports both formats.

5.1.6. *Octopus format.* Files in the Octopus format represent event traces. Originally conceived to represent event traces for printers, UPPAAL2OCTOPUS generates event traces from any timed UPPAAL model. The file contains a list of records, each on a separate line. Each record in the format contains the following tab-separated fields:

jobId: The identifier of a job the resource is working on as a string. As there is no relevant concept of a job in UPPAAL, UPPAAL2OCTOPUS abuses this field to refer to the current state a process is in.

At first the string "*process.location*" was used for this field. In this case ResVis erroneously reports "entry is missing corresponding start/end entry" for some entries, resulting in these entries being missing from the Gantt chart. Prepending this string with the unique locationId from the compiled UPPAAL model fixed this problem. A bug has yet to be filed concerning this problem.

pageNumber: For each job this pageNumber must be unique. There is also no relevant concept for a page in UPPAAL. That is why this field is not used in UPPAAL2OCTOPUS. To satisfy this uniqueness requirement the current locationId is outputted as the pageNumber.

scenario: There is also no relevant concept for a scenario in UPPAAL. Because the field has no use in the current context, UPPAAL2OCTOPUS outputs the constant string "UPPAALtrace" as the scenario.

resource: A resource is akin to a UPPAAL process. Thus the name of the process of the current trace element is outputted here.

eventId: An event corresponds directly to an UPPAAL event. Each event must have a "start" and "end" record. For each resource the event identifiers must be unique.

startEnd: Either the strings "start" or "end".

timeStamp: Floating point representation of the time. UPPAAL2OCTOPUS directly outputs the lower bound of the constraint for clock `c` in this field.

label: These strings can be viewed in ResVis by enabling labels on event trace files. (`ctrl-l`) Originally UPPAAL2OCTOPUS outputted the invariant of the current location as the label. However, due to a bug (see section 5.1.4) in UPPAAL this is not possible. The current location (same as the jobId) is therefor outputted as the label.

The files generated by UPPAAL2OCTOPUS are successfully read by ResVis. The Gantt charts drawn by ResVis are quite usable, as they show all the trace states, their time spans and their names. Thus UPPAAL2OCTOPUS can be considered to be a success.

5.2. **Controller Synthesis.** When modelling a machine such as the pre-production machine, it is sensible to split the environment and the controller. This results in the separation of the characteristics of the object of modelling, and the strategy employed by that object.

In the case of the pre-production machine, the controller only determines whether the Stage Unload Robot should take a wafer from Load Lock 1 or take a wafer from one of the Chucks. Some controllers can be considered incorrect, as there is a potential deadlock situation during which the Wafer Handler is full and the Stage Unload Robot can not unload any wafers from the Wafer Handler. The throughput of the machine depends on the behaviour of the controller, apart from the timing characteristics of the machine. An optimal controller is desired which maximises the throughput of the machine. In this case, a controller is considered optimal when for all possible runs the resulting trace uses a minimal amount of time. This is not the only possible optimality requirement imaginable. For example, a requirement stating that the amount of time a wafer is waiting at a Chuck to be removed by the Stage Unload Robot must be minimal could also be considered. Here, only maximising the throughput is considered.

5.2.1. *Fully non-deterministic controller.* As UPPAAL can compute the fastest trace for which a property holds, it is possible to derive how long a specific run should take for any optimal controller. This can be done by creating a fully non-deterministic controller, which can generate any possible behaviour. As there are only the two relevant channels to control the Stage Unload Robot, this controller (figure 18) consists of two transitions synchronising these channels without any additional conditions.

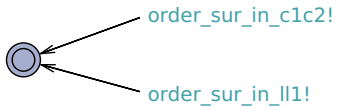

FIGURE 18. Non-deterministic Controller template `ControllerFree`, from [redacted]/third.xml.

UPPAAL can generate various traces for any given property. For example, the trace that takes the least amount of time can be generated by selecting the `fastest`-option in the Diagnostic Trace menu (figure 19). To get the fastest trace for a specific successful run UPPAAL can be asked whether the following property holds:

```
E<> deadlock and forall (x : wafer_t)wafer_status[x] == exited
```

The system declarations of the [redacted]/third.xml model contain an immutable clock `c` to keep track of how long a trace is. This clock can simply be read out from this trace, to know how fast the run made by an optimal controller is. If for another controller all possible runs take the same amount of time as the fastest successful trace of the fully non-deterministic controller, that controller is also optimal.

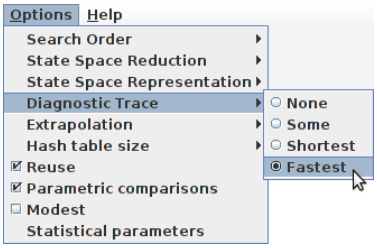

FIGURE 19. Menu option for a fastest Diagnostic Trace generation in UPPAAL.

5.2.2. *An optimal controller.* The fastest trace for the fully non-deterministic controller need not be logical: UPPAAL could generate a seemingly random trace. The trace however might give inspiration on a logical deterministic controller which might also be optimal.

For a run with 6 wafers, the fully non-deterministic controller outputs the following given a fastest trace:

1	order_sur_in_ll1
2	order_sur_in_ll1
3	order_sur_in_ll1
4	order_sur_in_c1c2
5	order_sur_in_ll1
6	order_sur_in_c1c2
7	order_sur_in_ll1
8	order_sur_in_c1c2
9	order_sur_in_ll1
10	order_sur_in_c1c2
11	order_sur_in_c1c2
12	order_sur_in_c1c2

This sequence of synchronisations suggests that having at most three wafers at the Stage Unload Robot, Vacuum Pre-Aligner, Stage Load Robot and Chucks is an optimal strategy. Thus, a controller with this optimal strategy would:

(1) Order the Stage Unload Robot to take three wafers from Load Lock 1 to the Vacuum Pre-Aligner.
(2) Alternatingly order the Stage Unload Robot to take a finished wafer from the Chucks to Load Lock 2 and then a new wafer from Load Lock 1 to the Vacuum Pre-Aligner, until there are no more new wafers available.
(3) Order the Stage Unload Robot to take the remaining three wafers from the Chucks to Load Lock 2.

This controller is implemented in `ControllerOptimal`, and shown in figures 20 and 21.

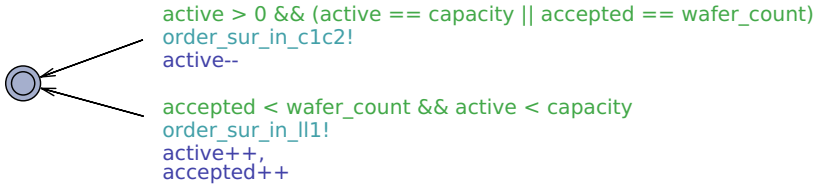

FIGURE 20. Optimal Controller template `ControllerOptimal`, from [redacted]/third.xml.

```
const int capacity = 3;

int[0,capacity] active = 0;
int[0,wafer_count] accepted = 0;
```

FIGURE 21. Declarations for the `ControllerOptimal` template, from [redacted]/third.xml.

By checking the trace time for this model both with the fully non-deterministic controller and optimal controller for a sufficient long run we can convince ourselves that `ControllerOptimal` is in fact optimal. This is at least the case for $n \leq 10$, as can be seen in figure 22.

5.2.3. *Optimality of the original controller.* Now that an optimal controller has been implemented, the question remains whether the controller in use by company A is also optimal. This controller has been implemented in `ControllerOriginal`, and is explained earlier in section 3.4.4. By extracting the fastest traces for all runs $n \leq 10$, it can be checked whether the controller by company A is optimal.

n	$t_{\texttt{free}}$	$t_{\texttt{optimal}}$	$t_{\texttt{original}}$	`diff`
1	1477	1477	1477	+0
2	1837	1837	1838	+1
3	2201	2201	2201	+0
4	2561	2561	2562	+1
5	2925	2925	2925	+0
6	3285	3285	3286	+1
7	3649	3649	3649	+0
8	4009	4009	4010	+1
9	4373	4373	4373	+0
10	4733	4733	4734	+1

FIGURE 22. Table describing the timings for the fastest traces for all the controllers.

As shown, the controller in use by company A is *not* optimal, albeit with a very small (and constant) margin. For any run $n \geq 1$ there is a delay of alternatingly 0 and 1 time units. To see why this is the case, the timings of each controller are first defined inductively:

$$
\begin{aligned}
t_{\texttt{free}}(1) &= 1477 \\
t_{\texttt{free}}(n+1) &= t_{\texttt{free}}(n) + 360 + 4((n+1) \bmod 2)
\end{aligned}
$$

$$
t_{\texttt{optimal}}(n) = t_{\texttt{free}}(n)
$$

$$
\begin{aligned}
t_{\texttt{original}}(1) &= 1477 \\
t_{\texttt{original}}(n+1) &= t_{\texttt{original}}(n) + 360 + 4((n+1) \bmod 2) + n \bmod 2
\end{aligned}
$$

Using the Gantt charts produced in section 5.1 (figures 23 and 24) the moment at which the traces of `ControllerOptimal` and `ControllerOriginal` diverge can be found. Though the traces differ from $t = 0$ onwards, the traces happen to resynchronise after processing every wafer. In `ControllerOptimal` the Stage Unload Robot immediately gets the order to move to Load Lock 1, whilst in `ControllerOriginal` this only happens later on. When the last wafer is about to be unloaded from a Chuck, the Stage Unload Robot will still have to move to the Chuck, while in `ControllerOptimal` it is already standing by. In `ControllerOriginal` the order for the Stage Unload Robot to move is signalled just after the Chucks are ordered to swap. During the swap, the Stage Unload Robot is moving towards the Chuck and will initialise the transfer of the wafer afterwards. The timings differ depending on which Chuck is used last.

For runs with an odd amount of wafers, the Stage Unload Robot needs to wait 3 time units for the transfer with Chuck 1 whilst it is in repose. For both the `ControllerOptimal` and `ControllerOriginal`, the Stage Unload Robot has to wait for the Chuck. The behaviour of the Chuck is the same for both controllers, and therefor the delay is 0 time units.

For runs with an even amount of wafers, the Stage Unload Robot is 1 time unit slower in preparing the transfer than Chuck 2. During this time Chuck 2 is standing by, waiting for the unloading. This delay results in the 1 time unit difference between both controllers, because the `ControllerOptimal` has prepared this transfer beforehand, and `ControllerOriginal` has not.

Thus this delay is:

$$
\begin{aligned}
\texttt{delay}_{n\ \texttt{even}} &= \texttt{prepareTakeSURFromWS\_startFromLL2} + \texttt{startTakeSURFromWS} \\
&\quad - \texttt{WSChuckSwapM2E} - \texttt{WSStartUnload} \\
&= \blacksquare + \blacksquare - \blacksquare - \blacksquare = 1 \\
\texttt{delay}_{n\ \texttt{odd}} &= \texttt{prepareTakeSURFromWS\_startFromLL2} + \texttt{startTakeSURFromWS} \\
&\quad - \texttt{WSChuckSwapM2E} - \texttt{WSStartUnload} + \text{synchronisation} \\
&= \blacksquare + \blacksquare - \blacksquare - \blacksquare + \blacksquare = 0
\end{aligned}
$$

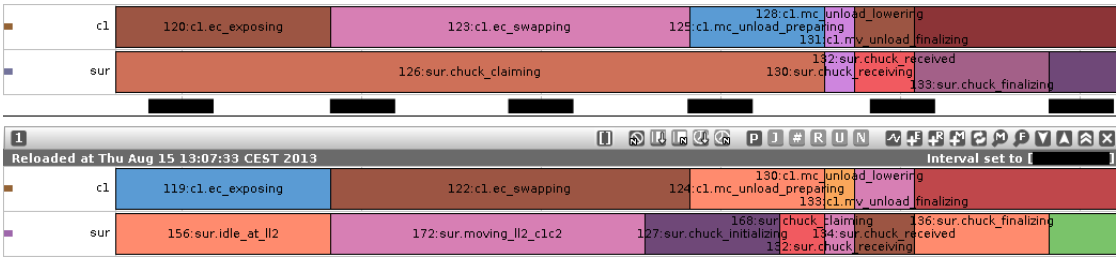

FIGURE 23. Part of two ResVis Gantt charts with `ControllerOptimal` (above) and `ControllerOriginal` (below) for a run with 9 wafers, from [redacted]/third deadlock.xml.

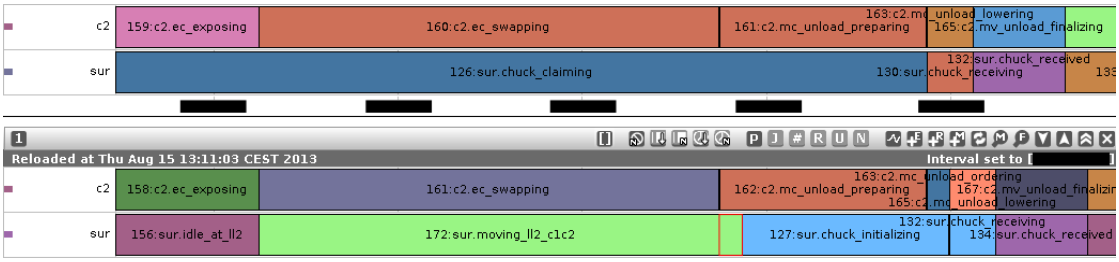

FIGURE 24. Part of two ResVis Gantt charts with `ControllerOptimal` (above) and `ControllerOriginal` (below) for a run with 10 wafers, from [redacted]/third deadlock.xml. Highlighted in red is the 1 time unit constant delay.

5.2.4. *Conclusion.* Using the fully non-deterministic controller `ControllerFree` an optimal controller `ControllerOptimal` was synthesised. For $n \leq 10$, the duration for the fastest traces for the models using `ControllerFree`, `ControllerOptimal` and `ControllerOriginal` have been measured.

From this it is concluded that the controller in use by company A is not optimal for $n \geq 1$. An argument is also given why the controller is not optimal for all non-tested runs $n > 10$. This is assuming no requirement is violated by `ControllerOptimal`, and the signals can be given by the controller at any time.

the signals given by the controller can be given at any time, and no requirement is violated in that process.

While the controller is strictly not optimal, it only has a constant overhead of 1 time unit for all runs with an even number of wafers, compared to an optimal controller. From this it can be concluded that the controller in use by company A is *almost* optimal.

5.3. **Verification of results.** Besides testing for deadlocks, author A also looks at other properties of interest. For the purpose of verification, the same properties have been tested for the new model for both controllers.

In order to check these properties, Observers were implemented: these automatons go into a *satisfied* or *violated* state when appropriate. This separates the measuring specifically for these

properties from the actual model. Because checking for these properties might significantly increase the state space size, these automatons can be disabled when necessary for efficiency. Also, because these Observers are separate from the actual model, the model will function regardless of the property being violated. Finally, these Observers should set themselves to *violated* or *satisfied* as quickly as possible. This way, UPPAAL can ignore certain branches earlier, minimising the state space. To this end, these Observers are set to have a higher priority compared to other processes. As soon as the Observer is able to claim it is done, it's always processed first. Unfortunately UPPAAL is not able to analyse properties concerning deadlocks for these models containing process priorities. So, in order to check for deadlocks, the Observers must be turned off, removing this ordering from the model.

5.3.1. *Parking Position Property.* The Stage Load Robot is capable of holding on to a wafer, after picking it up from the Vacuum Pre-Aligner, before putting it on one of the Chucks. The wafer has a temperature conditioned to [redacted] °C when taken from the Vacuum Pre-Aligner. The temperature of the wafer changes from unloading onwards, imposing stresses in the wafer. A maximum of ±[redacted]mK change in temperature is acceptable. This means that the wafer may stay in this parking position for up to [redacted] time units. If a controller strategy results in a wafer staying in this position longer, this Parking Position Property is violated.

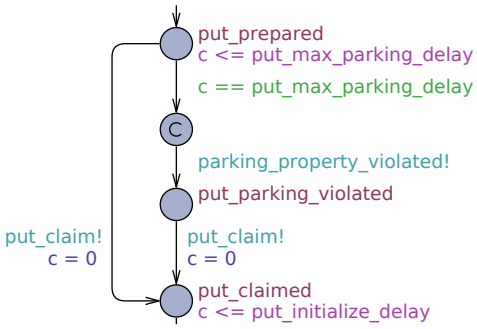

FIGURE 25. Cutout from the Stage Load Robot highlighting where the Parking Property is checked, from the [redacted]/third.xml model.

For efficiency, the actual measuring of the violation of this property is done within the Stage Load Robot. When violated, this automaton sends a signal (figure 25) to the ObserverParking-Position (figure 26). The property is considered satisfied, if no such signal was received, and the machine has successfully processed all wafers.

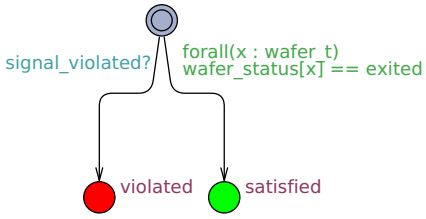

FIGURE 26. ObserverParkingPosition template of the [redacted]/third.xml model.

5.3.2. *Post-exposure Time Property.* After being exposed, the image captured on the wafer slowly dissipates and gets blurred. To prevent degradation, the wafer is baked in a machine outside the wafer scanner. In order to make sure all wafers have a satisfactory quality, the time between the exposure and baking should be constant for each wafer.

Because this baking machine is not modelled in the new model, the wafers are only considered up to the point they exit the pre-production machine. Thus the property states that the time each wafer requires to reach the Exit from the moment of exposure should be equal.

This property is difficult to check: the required time should be measured for each wafer, and compared. Because these measurements occur after one another, multiple clocks will be required to store the time. Storing pairs of clocks and comparing the differences between these pairs is not possible. This is not possible because UPPAAL only allows a limited set of operations on clocks:

- Only upper bounds ($<$, $\leq$, $==$) may be checked on invariants.
- Clocks may be compared to another clock or to integer values.
- A difference between of clocks may be compared to integer values.
- Clocks may be assigned an integer value, or the value of another clock.
- Comparison with a clock does internally not yield a boolean, and thus can not be used in functions.

Ideally these clocks should be *stopped* from increasing, like a stopwatch. Adding this to UPPAAL would extend it from supporting Linear Timed Automata to Linear Hybrid Automata [4]. Reachability analysis for these Linear Timed Automata however is undecidable, and therefor state space exploration will not always terminate. (Stopwatches have been implemented in version 4.1 of UPPAAL [7] using overapproximation, which was not known to the author of this thesis at the time that these properties were analysed.)

A way to implement this is to store these clocks as integer values. Assigning clocks to integer variables directly does not make sense. UPPAAL represents these clocks as bounds: a pair of two integers depicting the lower and upper bound of the time in a specific state. Storing either the lower bound, the upper bound or both bounds is a possibility. A C-style function to fetch one of these would be ideal, as it would not increase the complexity of the model too severely. However, because it is not possible to do clock comparisons in these functions, this is not an option.

The only solution to this problem is a hack that author A also uses: By using a SELECT-statement, and comparing the clock with all possible values for the integer type, the lower bound is gained. This has several problems: UPPAAL internally creates a transition for each possible value of this SELECT-statement. For the integer type, this would result in 65536 transitions to be generated. In this case UPPAAL is not able to instance the model for verification, as it produces a segmentation fault. However for this usecase a lower N can be used: in the new model the type SMALLTIME_T is specified as $\texttt{int}[0, N]$, where N is the max value for any clock. For 10 wafers, $N \approx 5000$ is sufficient.

These extra transitions, along with the extra integer values stored, result in a much more difficult model to compute. To minimise the state size, the same integer $t[x]$ is used for measuring both the exposure time and the exit time. Still, because of the extra computational load, it is nice to turn off this ObserverPostExposure when it is not used.

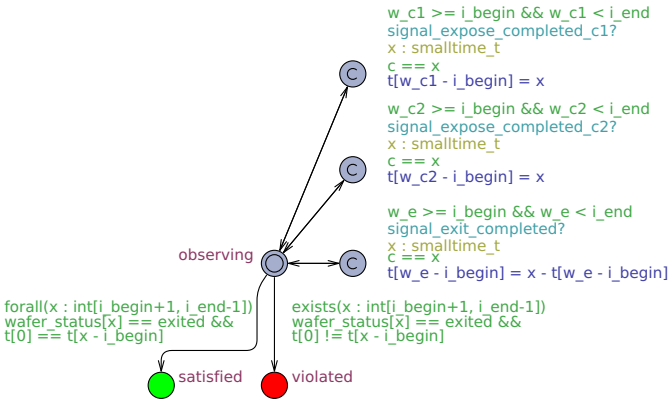

FIGURE 27. ObserverPostExposure template of the [redacted]/third.xml model.

5.3.3. *Wafer Exchange Time Property.* In order to analyse throughput of the machine, the time between loading and unloading the Chucks is measured. Specifically from just before preparing to unload to just after loading has finished. The first five wafers of a batch are ignored for this property. For the rest of the wafers, this time should be constant.

Because this property is not specific for a certain wafer, but for a pair of wafers, measuring the Wafer Exchange Time in UPPAAL is a bit harder than the Post-Exposure property. First, when a Chuck unloads a wafer, the time is recorded for that Chuck. Second, when that Chuck loads a new wafer, the interval is computed and stored for that loaded wafer. Because the first two wafers do not have a corresponding unloaded wafer, measuring this property for those elements does not make sense.

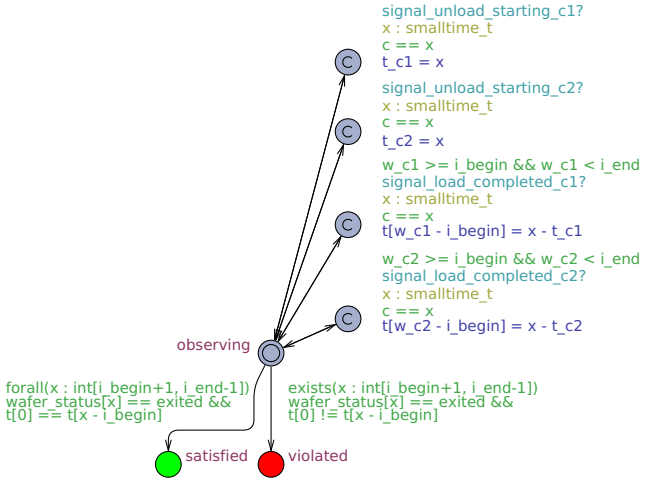

FIGURE 28. ObserverWaferExchange template of the [redacted]/third.xml model.

5.3.4. *Conformance of existing controllers.* author A has tested these properties for what she calls "deterministic timings", and "non-deterministic timings". An initializer automaton is used for the "deterministic timings", which sets the following timing constraints to specific ranges:

- prealigning by the Vacuum Pre-Aligner.
- measuring by the Chucks.
- exposing by the Chucks.

In the thesis by author A the timing constraints for which these properties has been tested are not mentioned explicitly. When looking at model A, the following constraints are used:

Model \ Constraint	prealign	measure	expose
Originally	[redacted]	[redacted]	[redacted]
Trace extraction model	[redacted, redacted]	[redacted]	[redacted, redacted]
Wafer Exchange models	[redacted, redacted]	[redacted]	[redacted]

FIGURE 29. Timing constraints as used in the models distributed with the thesis by author A, in time units (where 1 time unit equals 0.1 seconds).

Later in the thesis by author A she notes that a deadlock occurs with a expose time of [redacted] time units, and a pre-align time of [redacted] time units. In the models distributed along her thesis, this specific expose timing constraint is not tested. These prealign and expose constraints are also not near their original values. The range as used in the Trace extraction model by author A has been extended a bit for the new model to include this specific set of constraints as well. Finally company A had a request to test whether the machine works even if they change the timings a little bit. Thus a test has also been added for timing constraints with a variance of at most 10 time units. Because testing a wide range of these constraints is difficult for UPPAAL, these ranges could not be extended to encompass all values. Thus, the unification of these timing constraints has been tested in the new model. This has been implemented in the Initializer template, as shown in figure 30.

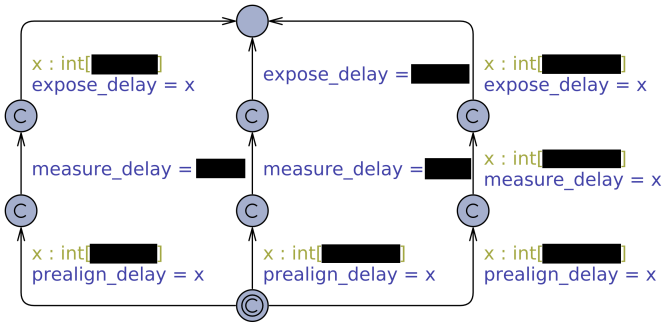

FIGURE 30. Initializer template of the [redacted]/third initializer.xml model.

author A also checks these properties for so-called "non-deterministic timings". She varies the timings of the prealign and expose phases vary during execution between 1 and 100 time units. Again, these new timings do not include the original values as provided by company A. As UPPAAL has difficulty with computing the model for just a varying expose timing constraint, computing the model with both varying prealign and varying expose timing constraints is not feasible. A choice has to be made: either both constraints may vary, but a lower variance must be chosen, or they do not vary simultaneously. For the new model these

constraints are simultaneously set to a variance of at most 5 time units. With these parameters the limits of what UPPAAL can compute within a reasonable amount of time and with a reasonable amount of memory are reached.

The `ControllerOriginal` and `ControllerOptimal` in the new model do not conform to all of these three properties. In order to test these properties, UPPAAL is asked whether or not a state is reachable for which the property is satisfied. In the case of the initializer automaton, if UPPAAL finds a state for which such a property holds then there is a combination of timing constraints which result in a controller being able to satisfy that property. In the case of the non-deterministic timings there is a sequence of timing constraints for which the controller happens to satisfy that property. If the controller does not satisfy the property in a static run, this non-deterministic trace might give clues as how to adapt the controller to work.

Model \ Property	Deadlock	Parking Position	Post Exposure	Wafer Exchange
author A	excluded	satisfied	satisfied	violated
Static and Original	excluded	violated	violated	violated
Static and Optimal	excluded	satisfied	violated	violated
Initializer and Original	excluded	satisfied	violated	violated
Initializer and Optimal	excluded	satisfied	violated	satisfied[1]
Nondeterministic and Original	excluded	violated	violated	violated
Nondeterministic and Optimal	excluded	satisfied	violated	violated

As model A satisfies the Post Exposure Time property, but the controllers of the new model do not, it can be concluded that the new model differs significantly in its behaviour. At least one of the models can thus be concluded to be incorrect.

5.3.5. *Continued Controller Synthesis.* `ControllerFree` is not capable to conform to the properties for the given timing constraints. To conform, the controller should park the wafer before handing it to the Chucks. In model A this is done by letting the Load Lock 1 hold on to the wafer, before the Stage Unload Robot takes it to the Vacuum Pre-Aligner. To synthesise a controller which has the desired behaviour, the non-deterministic `ControllerFree` should wait some time before it orders the Stage Unload Robot to move a wafer. Because this results in a much bigger state space, it is necessary to limit the amount of time the controller can wait. For this purpose `ControllerFreeDelayed` was implemented (figure 31), which defines an upper bound on the waiting time.

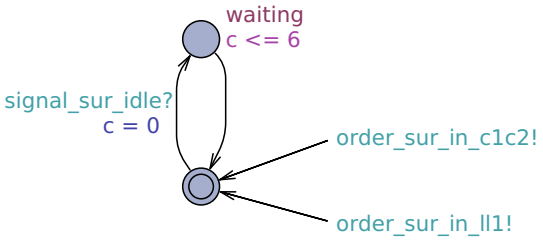

FIGURE 31. `ControllerFreeDelayed` template of the [redacted]/third.xml model.

[1]for example, measure = [redacted], expose = [redacted] and prealign = [redacted]

Looking at the traces generated by `ControllerOptimal`, the amount of time the controller should wait is 6 time units. UPPAAL is able to compute these properties for this bound of 6 time units within a reasonable amount of time, for runs up to 7 wafers. This controller is capable of finding strategies to satisfy all properties. However, it is not capable of finding a strategy that satisfies all properties simultaneously. It can be concluded that these properties are mutual exclusive, given this model along with its timing constraints, given a waiting time of up to 6 time units, and given the capabilities the controller currently has.

6. Conclusion

An approach for Flow Systems in UPPAAL has been introduced. This *FS-approach* has been used to implement a model for the pre-production machine by company A. This is considered to be a non-trivial use-case. This implementation greatly improved upon the traceability, simplicity and extensibility compared to the previous implementation. However, it is concluded that the new model is not valid.

Next, a tool was written to generate Gantt charts from traces of arbitrary timed UPPAAL models. These Gantt charts are great for visualising traces. They enable the comparison with traces generated from other models, code or from measurements in an actual machine.

Then, an optimal controller was synthesised for the new model. This controller completes 1 time units faster for runs with an odd number of wafers compared to the original controller in use by company A. Thus, the controller in use by company A can be considered to be almost optimal.

Following, a set of extra properties as specified by Adyanthaya (or author A) was also analysed for the new model. This analysis gave insight into how much the new model actually differs from model A. Also, for the new model, it is concluded that the properties as specified by author A are mutual-exclusive.

Thus, the new model can be considered useable, augmented by the Gantt charts. Useful properties for this model were examined in detail. However it is advisable to ensure that models are valid, otherwise this exercise would be pointless for industry applications. For the creation of models for such complex machines, it is suggested to use some form of compilation in order to generate these models from the specification.

6.1. Further challenges.

Model generation: In order to ensure the validity of the model, it would be nice to apply the *FS-approach* specified in section 3.1 in an automated fashion. Ideally, to translate the POOSL specification to a timed UPPAAL model (as described in [20]), a compiler would have to be made. It would then only be necessary to check the validity of this compiler.

These generated models would also have to be readable by humans. As UPPAAL models contain information on how to layout the graphs, GraphViz could be used to suggest a layout.

Stopwatches: At the time the models for this thesis were created, it was not known to the author that the development version of UPPAAL (v4.1 and newer) supports the use of stopwatches. In section 5.3, a hack was used to implement Observers, involving the storage of clock values as integers. The work done in this chapter could be revisited, by in stead using these stopwatches as described in [7]. This will probably result in the properties being easier to verify by UPPAAL, paving the way for larger ranges of timing constraints to be verified.

References

[1] The POOSL website. http://www.es.ele.tue.nl/she/index.php?select=32.

[2] Shreya Adyanthaya. Formal Time-based Wafer Flow Modeling and Verification. Eindhoven University of Technology, 2011. MSc Thesis.

[3] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.

[4] Franck Cassez and Kim Larsen. The impressive power of stopwatches. In *CONCUR 2000–Concurrency Theory*, pages 138–152. Springer, 2000.

[5] [redacted], 2013.

[6] Edmund M Clarke, Orna Grumberg, and Doron A Peled. *Model checking*. MIT press, 1999.

[7] Alexandre David, Jacob Illum, Kim G Larsen, and Arne Skou. Model-based framework for schedulability analysis using uppaal 4.1. *Model-Based Design for Embedded Systems*, pages 93–119, 2009.

[8] Alexandre David and Wang Yi. Modelling and analysis of a commercial field bus protocol. In *Real-Time Systems, 2000. Euromicro RTS 2000. 12th Euromicro Conference on*, pages 165–172. IEEE, 2000.

[9] Orna Grumberg and Helmut Veith. *25 years of model checking: history, achievements, perspectives*, volume 5000. Springer, 2008.

[10] John Håkansson and Paul Pettersson. Partial order reduction for verification of real-time components. In *Formal Modeling and Analysis of Timed Systems*, pages 211–226. Springer, 2007.

[11] Anders Hessel and Paul Pettersson. Model-based testing of a wap gateway: An industrial case-study. In *Formal Methods: Applications and Technology*, pages 116–131. Springer, 2007.

[12] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[13] AH Mader, H Wupper, and M Boon. The construction of verification models for embedded systems. 2007.

[14] Paul Pettersson. Formal methods applied in industry-on the commercialisation of the uppaal tool. In *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual*, pages 450–451. IEEE, 2011.

[15] Davor Slutej, John Håkansson, Jagadish Suryadevara, Cristina Seceleanu, and Paul Pettersson. Analyzing a pattern-based model of a real-time turntable system. *Electronic Notes in Theoretical Computer Science*, 253(1):161–178, 2009.

[16] F.W. Vaandrager. A First Introduction to Uppaal. http://www.mbsd.cs.ru.nl/publications/papers/fvaan/handbookuppaal/, 2013.

[17] JPM Voeten. *POOSL: An object-oriented specification language for the analysis and design of hardware/software systems*. Eindhoven University of Technology, Faculty of Electrical Engineering, 1995.

[18] Aneta Vulgarakis, Cristina Seceleanu, Paul Pettersson, Ivan Skuliber, and Darko Huljenic. Validation of embedded systems behavioral models on a component-based ericsson nikola tesla demonstrator. In *Quality Software (QSIC), 2011 11th International Conference on*, pages 156–165. IEEE, 2011.

[19] Christian Wagner and Noreen Harned. Euv lithography: Lithography gets extreme. *Nature Photonics*, 4(1):24–26, 2010.

[20] Jiansheng Xing, Bart D Theelen, Rom Langerak, Jaco van de Pol, Jan Tretmans, and JPM Voeten. From poosl to uppaal: Transformation and quantitative analysis. In *Application of Concurrency to System Design (ACSD), 2010 10th International Conference on*, pages 47–56. IEEE, 2010.

Appendix A: List of UPPAAL models

approach/nontimed.xml: Basic case of model using the *FS-approach*. Used in section 3.1.

approach/timed.xml: Version of previous model using timed constraints.

two-way/deterministic.xml: Application of the *FS-approach* on the two-way proof of concept with deterministic timings used in section 3.2.

two-way/deterministic fixed.xml: Version of previous model with the deadlock-problems as described by author A fixed.

[redacted]**/first.xml:** First iteration of the implementation of the pre-production machine Wafer Scanner containing the Chucks, Generator and Exit (section 3.4.1).

[redacted]**/second.xml:** Second iteration of the implementation of the pre-production machine Wafer Scanner to which the Stage Unload Robot, Vacuum Pre-Aligner and Stage Load Robot have been added (section 3.4.2).

[redacted]**/third.xml:** Third iteration of the implementation of the pre-production machine Wafer Scanner to which the Load Lock 1 and Load Lock 2 have been added (section 3.4.3). Also the ObserverParkingPosition, ObserverPostExposure and ObserverWaferExchange used in section 5.3 have been added. In this implementation, either the `ControllerFree`, `ControllerOriginal` or `ControllerOptimal` can be used. Because the Observers use an execution ordering, the deadlock property cannot be checked for this model.

[redacted]**/third deadlock.xml:** Version of previous implementation without the Observers, so that the deadlock property can be verified.

[redacted]**/third initializer.xml:** Version of the `third.xml` model with deterministic timings implemented by an Initializer automaton.

[redacted]**/third initializer deadlock test.xml:** Version of the `third deadlock.xml` model with deterministic timings implemented by an Initializer automaton.

[redacted]**/third nondeterministic.xml:** Version of the `third.xml` model with non-deterministic timings.

bugreports/invariants.xml: Model demonstrating a bug in UPPAAL which incorrectly represents a timing invariant, as described in section 5.1.4.

bugreports/invariants-function.xml: Model demonstrating a bug in UPPAAL which incorrectly represents an invariant containing a function call, as described in section 5.1.4.

Appendix B: Gantt charts

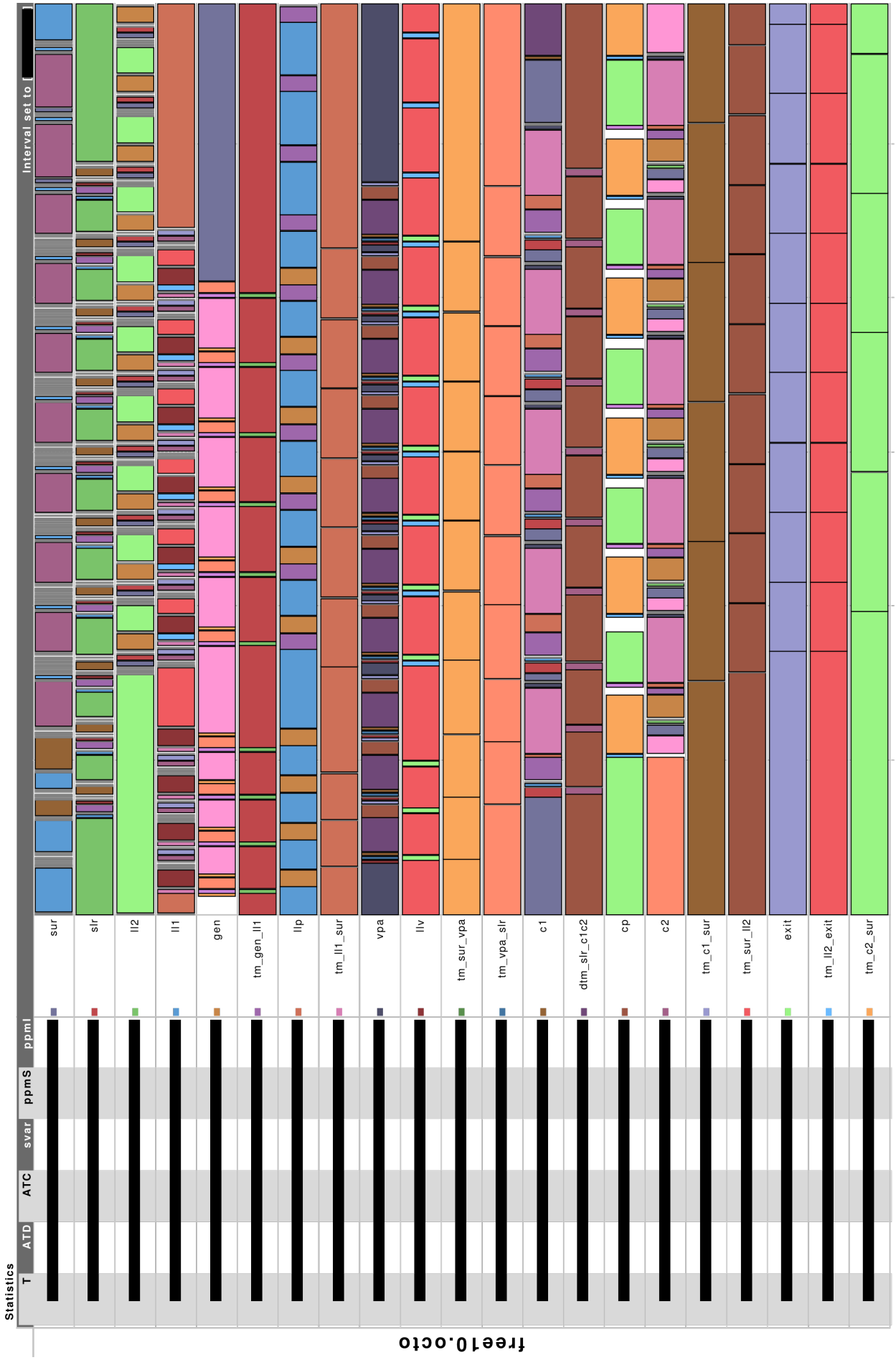

Figure 32. ResVis Gantt chart of an optimal trace for a run with 10 wafers, from `ControllerFree`.

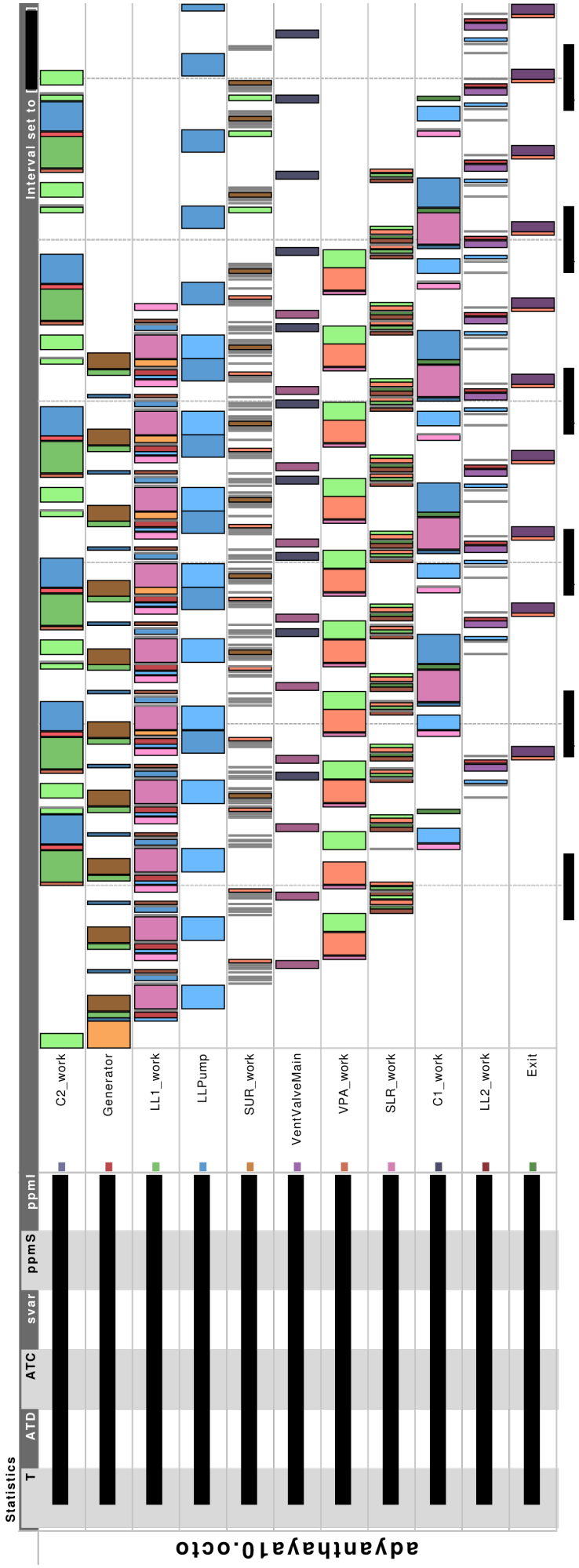

FIGURE 33. ResVis Gantt chart of a trace for a run with 10 wafers, from model A.

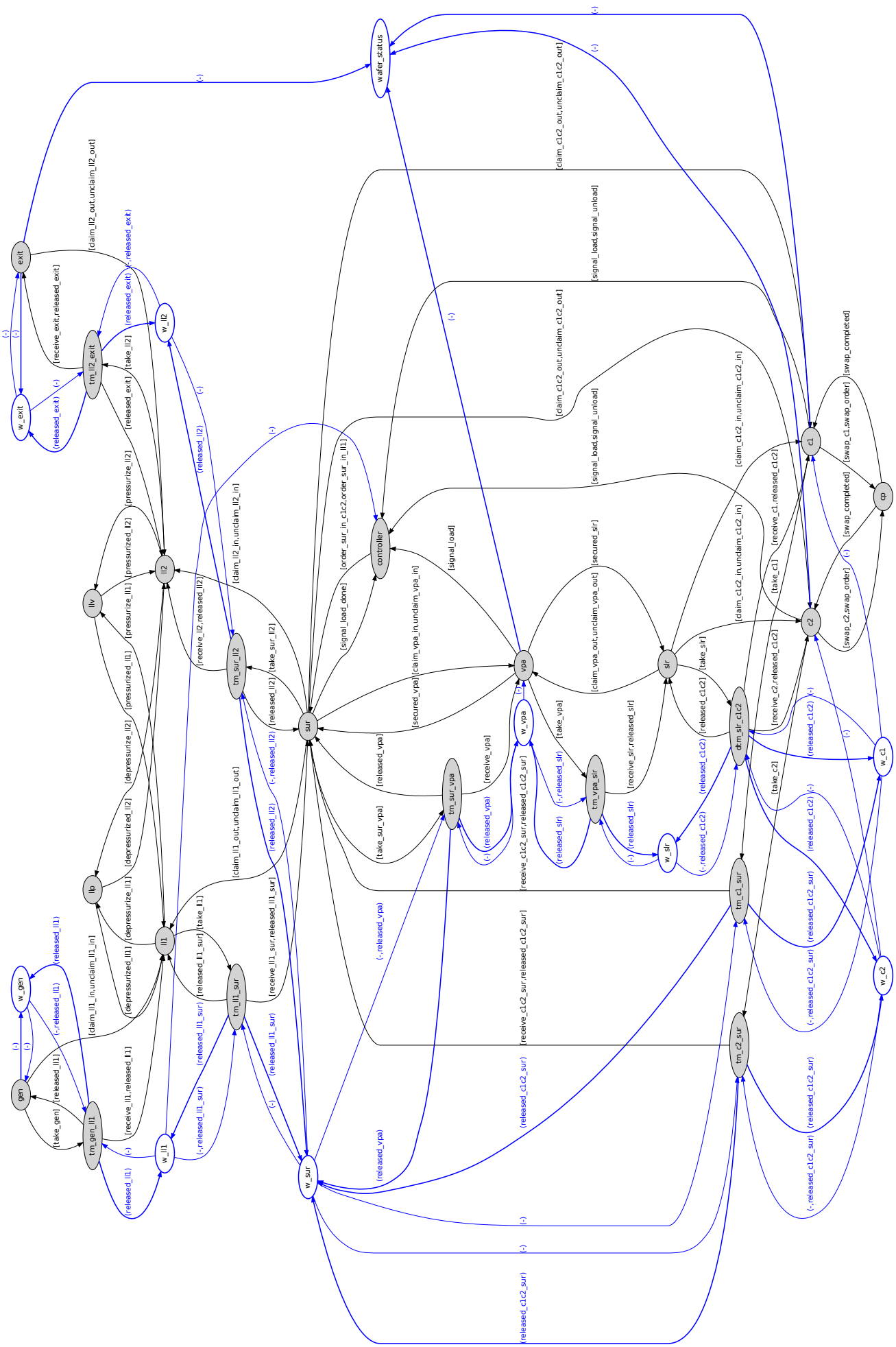

Figure 34. Communication in [redacted]/third.xml with `ControllerOriginal` by the TAFLOW program from LIBUTAP.